TRANSFORMAÇÃO DE MODELOS ORM-UML

Ambrósio Alves Soares

Dissertação para a obtenção do grau de Mestre em

Engenharia Informática

Trabalho efetuado sob a orientação de:

Prof.ª Paula Ventura Martins

Setembro de 2014

TRANSFORMAÇÃO DE MODELOS ORM-UML

Ambrósio Alves Soares

Dissertação para a obtenção do grau de Mestre em

Engenharia Informática

Trabalho efetuado sob a orientação de:

Prof.ª Paula Ventura Martins

Setembro de 2014

TRANSFORMAÇÃO DE MODELOS ORM-UML

Declaração de autoria de trabalho

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

(Ambrósio Alves Soares)

A Universidade do Algarve tem o direito, perpétuo e sem limites geográficos, de arquivar e publicitar este trabalho através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, de o divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Dedicatória

"Dedico esta dissertação aos meus Pais, irmãos, a minha Esposa e a minha filha como forma da minha imensa gratidão ..."

Agradecimentos

Primeiramente a **Deus** todo-poderoso, por me dar existência

Aos meus pais Ambrósio Albino Soares e Maria Matilde Alves Lampeão Soares pelo carinho e apoio incondicional ao longo da minha vida. Amo-vos muito.

Aos meus irmãos Norberto (Betinho), Alcindo (Docas) e Edite, por serem quem são, meus irmãos de verdade.

À minha esposa Elsa de Paula Soares, presença incontornável na minha vida, e em especial a minha filha Matilde (*Puka*), que à adoro tanto.

À minha avó Corália e avô Lampeão pelos sábios ensinamentos e aconselhamentos, que Deus os tenha no esplendor da luz perpétua.

O meu profundo reconhecimento de gratidão aos Professores Dr. António Rosado e a Professora Dr.ª Paula Ventura Martins, que tão sabiamente assistiram e teceram considerações de base para que este trabalho se tornasse possível.

Aos defuntos da família Soares e Lampeão, e de todos amigos, que Deus os tenha e os guarde em Paz.

A todos que de forma direta ou indireta forneceram elementos para que este trabalho se efetivasse, vai o meu sentimento de apreço, reconhecimento e gratidão

Resumo

O crescente avanço das tecnologias nas diversas áreas, como a engenharia de software, permitem o desenvolvimento de aplicações baseado em modelos independentemente de linguagens e plataformas.

Estes modelos compõem a base das novas arquiteturas no desenvolvimento de software como a ***Model Driven Architecture*** **(MDA)** do ***Object Management Group*** **(OMG)**, onde os principais fundamentos são a representação formal destes modelos e os mecanismos de transformação de um modelo em outro.

Contudo, a modelação e respetivas ferramentas não são um assunto novo. Como tal, sistemas complexos podem ser mais facilmente entendidos e geridos através de modelos que permitem uma abstração da realidade. A ***Object-Role Modeling*** **(ORM)** e a ***Unified Modeling Language*** **(UML)** são exemplos pragmáticos de abordagens de modelação, cujo conceito do ponto de vista sistemático é composto por partes, cada uma das quais tendo suas próprias metas e relações.

Entre os desafios emergentes, relacionados com a variedade de abordagens, está a necessidade de um mecanismo de transformação entre modelos ORM e modelos da UML. Este trabalho propõe e implementa este mecanismo de transformação entre modelos ORM e modelos UML.

A transformação recebe como entrada um ficheiro que representa um esquema conceptual ORM, e processa a conversão. Após o processo de conversão, disponibiliza um ficheiro de saída contendo o esquema lógico UML. O processo é automático, permitindo iniciar a conversão a partir do esquema conceptual ORM de entrada. O processo de conversão é baseado em regras genéricas de mapeamento que têm pré-condições e prioridades de aplicação, procurando obter o esquema lógico UML.

PALAVRAS-CHAVE: ORM, UML, Modelos, Modelação, Transformação, Conversão.

Abstract

The increasing advancement of technology on several areas such as software engineering that allow applications development based on models regardless of languages and platforms.

These models represent the basis of new architectures in software development as the Model Driven Architecture (MDA) of the Object Management Group (OMG), where the main fundaments are the formal representation of these models and the transformation mechanism from one model to another.

However, modeling and modeling tools are not a new subject. Thus, complex systems can be more easily understood and managed through models that allow an abstraction of reality. The Object-Role Modeling (ORM) and the Unified Modeling Language (UML) are pragmatic examples of modeling approaches, whose concepts from a systematic point of view are composed of parts, each of which having its own goals and relationships.

Among the emerging challenges, related to the variety of approaches, is the need for a transformation mechanism between ORM models and UML models. This investigation work proposes and implements the transforming mechanism from ORM models to UML models.

The transformation takes as input a file that represents an ORM conceptual schema and makes the conversion. After the conversion process, it provides an output file containing the UML logical schema. The conversion process is automatic, allowing to start the conversion from the ORM conceptual schema entry. The conversion process is based on generic mapping rules that have pre-conditions and implementation priorities, trying to obtain the UML logical schema.

KEYWORDS: ORM, UML, Models, Modeling, Transformation, Conversion.

Índice Geral

Índice de figuras

Índice de tabelas

Lista de abreviaturas, siglas e símbolos

ATL	*ATLAS Transformation Language*
BD	*Data Base*
BOTL	*Basic Object-oriented Transformation Language*
CIM	*Computation Independent Model*
CORBA	*Common Object Request Broker Architecture*
CWM	*Common Warehouse Metamodel*
EA	*Enterprise Architect*
EMF	*Eclipse Modeling Framework*
ER	*Entity Relationship*
FCO-IM	*Fully Communication Oriented Information Modeling*
GUI	*Graphic User Interface*
MDA	*Model Drive Architecture*
MOF	*Meta Object Facility*
NIAM	*Natural Information Analysis Method*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
OMG RTF	OMG *Revision Task Force*
OML	OPEN-*Modeling Language*
OOSE	*Object-Oriented Software Engineering*
ORM	*Object Role Modeling*
OTM	*Object Modeling Technique*
PIM	*Platform-Independent Model*
POF	Paradigma Orientado por Fatos
POO	Paradigma Orientado por Objetos
PSM	*Platform-Specific Model*
QVT	*Query View Transformation*
SGBD	*Database Management System*
UML	*Unified Modeling Language*
YATL	*Yet Another Transformation Language*

XMI	*XML Metadata Interchange*

Capítulo 1 – Introdução

Nos últimos anos, os avanços tecnológicos foram significativos em várias áreas. Este avanço foi evidente no domínio da informática, onde predominavam aplicações isoladas e dependentes de certas tecnologias e ambientes. Atualmente, num contexto totalmente diversificado, constituído por várias aplicações que interagem para a resolução de problemas cada vez mais complexos. Assim, a área da engenharia de software não foi exceção. Temas como o paradigma de desenvolvimento de software, baseado em modelos, constituem novas áreas de atuação, permitindo o desenvolvimento de software a partir de modelos, independentemente das linguagens de programação e infraestruturas usadas.

Neste domínio emergiu o *Model Drive Architecture* (MDA) (MDA, 2013) do *Object Management Group* (OMG) (Object Management Group, 2010) (Mellor, Scott, Uhl, & Weise, 2004), onde os principais fundamentos são a representação formal destes modelos e os mecanismos de transformação de modelos.

A MDA constitui uma das principais abordagens para representação formal dos modelos usando especificações padrão de modelação como *MetaObject Facility* (MOF) (MOF, 2013), *Unified Modeling Language* (UML) (Rumbaugh, Jacobson, & Booch, 1999) e *Common Warehouse Metamodel* (CWM) (Rumbaugh, Jacobson, & Booch, 1999).

As linguagens de modelação não constituem novidade, sendo usadas desde os anos 90 na análise e desenho de software. No desenvolvimento de sistemas complexos facilitam a sua compreensão e gestão através de modelos que tornam a realidade abstracta mais tangível. A *Object Role Modeling* (ORM) (Halpin T. , 2008) e a UML são exemplos de linguagens de modelação, apresentando cada uma as suas próprias características e objetivos.

A ORM é uma linguagem de modelação conceptual. Na prática, os modelos ORM são mais fáceis de validar e melhorar do que os modelos de outras linguagens, tais como a UML. As suas principais características são a proximidade da linguagem natural e facilidade de comunicação com utilizadores pouco experientes.

Contudo, a UML continua a ser a linguagem universalmente reconhecida no desenho de sistemas de software e geração automática de código. Neste domínio emerge a necessidade de uma ferramenta de transformação de modelos ORM para modelos da UML.

1.1 Motivação

A compreensão dos requisitos funcionais de um sistema é vital para o sucesso dos projetos de desenvolvimento de software e para a aceitação dos clientes. Com a popularização do paradigma orientado por objeto, a UML tornou-se a linguagem padrão a aplicar nas fases de análise e desenho do processo de desenvolvimento de software. Porém, a UML ainda não apresenta características que se revelam essenciais na validação e comunicação dos requisitos aos utilizadores finais. Foi necessária a introdução de outras linguagens para conferir maior expressividade, clareza e mecanismos de validação aos modelos conceptuais construídos. A ORM ultrapassa esta lacuna, sendo atualmente uma linguagem aplicada na modelação de requisitos de negócios. Neste contexto, a ORM tem evoluído e ganho aceitação nas comunidades de modelação.

Iremos recorrer à linguagem ORM para a modelação conceptual, uma vez que está mais próxima da linguagem natural. Adicionalmente, esta linguagem tem uma representação com alto nível de abstração, modela de forma natural os fatos do mundo real, suas propriedades e seus relacionamentos. Outra razão para o recurso à ORM, reside no fato de não existirem semelhanças entre os diagramas de casos de utilização e os diagramas de classes que possibilitem a transformação automática entre estes tipos de diagramas.

Porém, a ORM não segue uma representação formal que permita a geração automática de código. Neste contexto, a UML será a linguagem mais adequada para a representação intermédia, e segundo o paradigma MDA a escolha natural para a geração de código.

A conversão para modelos UML permitirá uma representação menos complexa da estrutura e regras de negócio (modelo lógico) do sistema de software e mais próxima da implementação.

O trabalho que se apresenta nesta dissertação permite introduzir um mecanismo de transformação entre duas linguagens, a ORM e UML (Figura 1.1). A ORM sendo aplicada essencialmente no levantamento de requisitos possibilitará a geração automática de modelos de desenho UML. Este processo irá facilitar o trabalho dos analistas e modeladores de sistemas de software.

A Figura 1.1 mostra um esquema ilustrativo da problemática no contexto da transformação das linguagens ORM e UML. Com a utilização desses dois métodos, é necessário um meio de mapeamento dos elementos ORM em elementos da UML. Esse mapeamento é conseguido através de um conjunto de diretrizes que ditam as regras pelas quais os elementos são mapeados.

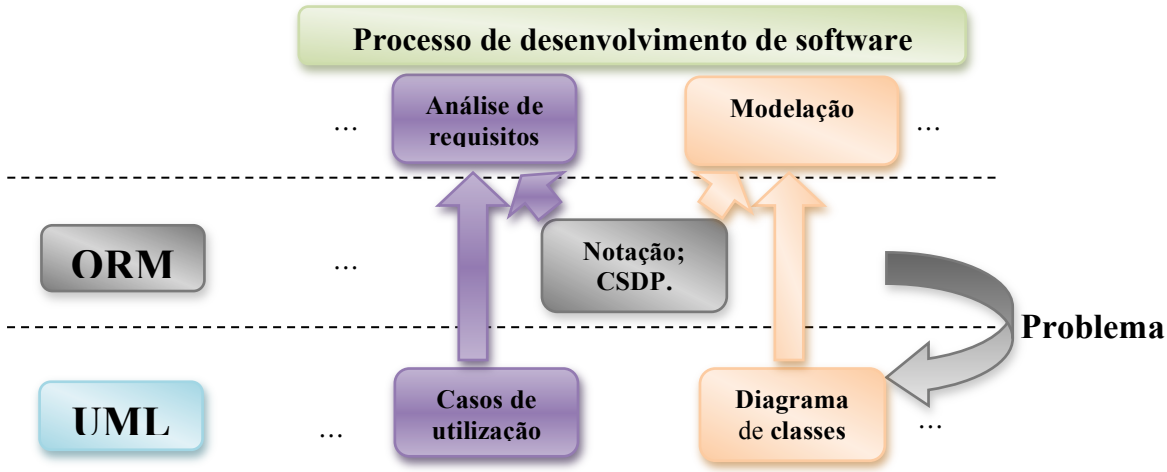


Figura 1. 1 - Esquema ilustrativo do problema

Atualmente ainda não existem e nem foram trabalhados exaustivamente estes tipos de transformações entre modelos. Existe um conjunto de regras de conversão, mas a nível de implementação o trabalho realizado não é exaustivo. Com este trabalho pretende-se estudar em profundidade os conceitos e ferramentas, de forma a implementar a transformação de modelos ORM para diagramas de classe da UML.

Vários trabalhos têm apresentado propostas para a modelação de dados em diferentes níveis de abstração. Na revisão da literatura constatou-se que existem algumas ferramentas que propõem a conversão de dados, geralmente a partir de sistemas de gestão de base de dados (SGBDs), em documentos XML (Bourret, 2007). Existem ainda outras ferramentas proprietárias que incluem módulos para transformar modelos UML para o equivalente em XML *Schema* (SPARX, 2008). Um trabalho relacionado, semelhante ao proposto nesta dissertação, descreve um algoritmo proposto por *Bollen*

(Bollen, 2002) para converter um esquema ORM num diagrama de classes UML, porém o algoritmo em questão nunca foi implementado devido a sua complexidade [**Anexo B** – Algoritmo de Transformação ORM-para-UML].

1.2 Objetivos

O ponto de partida para este trabalho foi a identificação de duas comunidades cujo tema de trabalho são as linguagens de modelação: a comunidade que trabalha com ORM e a comunidade que trabalha com UML. Aparentemente ambas as comunidades refletem sobre as mesmas questões. No entanto, o foco de trabalho é diferente. A comunidade ORM considera relevante as atividades de levantamento das necessidades implementando uma linguagem próxima da linguagem natural e que facilite a comunicação. A comunidade UML desenvolveu uma linguagem de modelação visual que permite a transição para a geração de código.

Assim sendo, o objetivo principal desta dissertação é a definição de um mecanismo que permita a transformação automática entre modelos ORM e UML baseada nas boas práticas da engenharia de software e seguindo os princípios orientadores da MDA.

Desta forma, os objetivos gerais desta tese cobrem os seguintes aspetos:

- Definir e aplicar um modelo de referência que permita comparar e identificar as características comuns ou especificas de ambas as linguagens de modelação;
- Definir o mecanismo a aplicar na transformação de modelos ORM para modelos UML.
- Desenvolver um ambiente que aplica um processo de transformação para gerar esquemas lógicos UML a partir de esquemas conceptuais ORM.
- Definir ou adotar uma metodologia que identifique as boas práticas a aplicar na transformação de modelos, especialmente no que diz respeito às linguagens de modelação abordadas neste trabalho.

1.3 Organização da dissertação

Esta dissertação está organizado da seguinte forma.

O capítulo 1 contextualiza os principais aspetos desta dissertação e apresenta a uma descrição geral do trabalho, motivação e objetivos gerais.

O capítulo 2 apresenta uma revisão da bibliografia sobre o ORM, que dá suporte ao entendimento do paradigma orientado por fatos.

O capítulo 3 apresenta uma revisão da bibliografia sobre a UML e o suporte ao entendimento do paradigma orientado por objetos.

No capítulo 4 apresenta-se uma descrição das abordagens de transformação propostas pela MDA. Inclui uma análise sucinta sobre cada uma dessas abordagens, uma descrição dos vários métodos de transformação de modelos, bem como as linguagens adequadas.

O capítulo 5 mostra a metodologia de transformação, ou seja, os métodos e técnicas que permitem a transformação de modelos ORM em modelos UML. Descreve o ambiente e todos os aspetos e detalhes da implementação.

O capítulo 6 descreve o caso de estudo que permite validar o ambiente, e apresenta os testes e resultados.

O capítulo 7 apresenta as conclusões do trabalho e um conjunto de sugestões para trabalhos futuros.

Capítulo 2 – Paradigma Orientado por Fatos (POF)

A principal área de investigação, que fundamenta este trabalho, centra-se na modelação de sistemas de software. Neste panorama, é essencial fazer uma revisão da literatura relacionada com um dos tópicos principais deste trabalho, que é o Paradigma Orientado por Fatos (POF). A *Object-Role Modeling (ORM)* (Halpin T. , 2008) é a abordagem de modelação orientada por fatos que será aplicada neste trabalho.

O presente capítulo está estruturado da seguinte forma. A secção 2.1 descreve detalhadamente o POF no contexto da ORM. A secção 2.2 incide sobre a ORM e suas características. A secção 2.3 descreve os tipos de objetos. A secção 2.4 expõe os tipos de relacionamentos. A secção 2.5 apresenta as principais restrições em ORM. A secção 2.6 descreve o tipo de fatos derivados. Finalmente, a secção 2.7 apresenta a modelação de processos em ORM.

2.1 Introdução

O Paradigma Orientado por Fatos (POF) (Halpin T. , 2005, p. 3) é uma abordagem conceptual aplicada na captura, modelação e validação de regras de negócios. Esta abordagem é feita em termos de **fatos** de interesse subjacente, verbalizados numa linguagem facilmente compreensível por utilizadores sem conhecimentos técnicos desse mesmo domínio. Um **fato** (Halpin T. , 2008, p. 33) é uma preposição verdadeira. Os fatos similares são representados através de um **tipo de fatos**. Por exemplo, os fatos *Ambrósio nasceu em Moçambique* e *António nasceu em Angola* são representados pelo tipo de fatos *Professor nasceu em País*, considerando que *Ambrósio* e *António* são professores e que *Moçambique* e *Angola* são países. Isto é, aqueles dois fatos são instâncias do tipo de fatos.

O POF compreende uma família de "dialetos" estreitamente relacionados onde se destacam: o *Natural language Information Analysis Method* (NIAM) (Halpin T. , 2007) (Halpin T. , 2008), a *Object-Role Modeling* (ORM), a *Fully Communication Oriented Information Modeling* (FCO-IM) (Halpin T. , 2007) (Halpin T. , 2008), o *Predicator Set Model* (PSM) (Halpin T. , 2007). Todas estas abordagens são muito semelhantes, apresentando apenas alguns detalhes que as diferenciam. Atualmente, a abordagem mais

conhecida, e de longe a mais aplicada, é a ORM. Assim, esta secção recorre à ORM para apresentar os conceitos associados ao POF.

Uma das características essenciais do POF (Halpin T. , 2007, pp. 19,20) é não recorrer aos atributos, por oposição aos paradigmas *Entity-Relationship* (ER) e a *Unified Modeling Language* (UML).

As vantagens do uso de relacionamentos em vez de atributos são:

- O modelo conceptual e as respectivas consultas são mais estáveis;
- O modelo conceptual facilita a verbalização natural;
- O modelo conceptual pode convenientemente usar múltiplas instâncias, sendo que, com atributos seria demasiado complexo;
- O modelo conceptual é mais uniforme.

De fato, um modelo livre de atributos é mais estável. Os modelos que utilizam atributos são mais instáveis porque muitas vezes os atributos têm de ser substituídos por relacionamentos em versões seguintes do modelo. Por exemplo, em vez de usar os atributos *eFumador* e *nasceuPaís* para o objeto *Professor*, os modelos orientados por fatos recorrem aos tipos de fatos *Professor fuma* e *Professor nasceu em país*.

Para explicar a facilidade de verbalização natural consideremos um exemplo. Em vez de recorrer aos atributos *Professor.eFumador* e *Professor.NasceuPais*, as seguintes descrições são utilizadas na ORM: *Professor fuma* e *Professor nasceu em país*. Como se pode constatar, esta descrição é mais próxima da linguagem natural, tornando mais produtiva a comunicação entre todas as partes envolvidas.

Por outro lado, a comunicação nos modelos ORM é feita através de frases simples, em que cada tipo de fato pode ser facilmente divulgada através de múltiplas instâncias. O tipo de fatos *Professor nasceu em país* tem as seguintes instâncias (fatos): *Ambrósio nasceu em Moçambique* e *António nasceu em Angola*.

Num modelo ORM livre de atributos descrevem-se todos os fatos da mesma forma, recorrendo a papéis, o que resulta numa notação uniforme e simples de aplicar. Assim sendo, não é necessária uma notação diferente para atributos ou relações, quando se aplica a mesma restrição.

No entanto, as notações baseadas em atributos permitem construir diagramas mais compactos e mais próximos da implementação. Por tais razões, o POF é frequentemente

usado em conjunto com a notação baseada em atributos, especialmente para a análise conceptual. Especificamente, a modelação orientada por fatos inclui procedimentos de mapeamento para estruturas baseadas em atributos, tais como as de ER ou UML.

2.2 Object-Role Modeling (ORM)

A ORM (Halpin T. , 2008) (Halpin T. , 2007, p. 21) foi introduzida em meados da década de 1970, com o objetivo de proporcionar um maior nível de semântica para a modelação dos sistemas de informação.

A ORM é essencialmente uma abordagem conceptual para modelar, transformar e consultar um sistema de software. A ORM difere da modelação *Entity-Relationship* (ER) (Chen, 1976) e do diagrama de classes da *Unified Modeling Language* (UML) (Rumbaugh, Jacobson, & Booch, 1999) porque a ORM trata todos os fatos como relacionamentos (unários, binários, ternários, etc.).

A notação gráfica da ORM é mais expressiva do que as notações gráficas da modelação ER e da UML. De igual forma, a linguagem textual da ORM é baseada em subconjuntos de línguas nativas, sendo por isso mais fácil de compreender pelo utilizador comum, do que linguagens técnicas como a *Object Constraint Language* (OCL) utilizada pela UML (Rumbaugh, Jacobson, & Booch, 1999).

Contudo, é de salientar que a ORM tem vindo a evoluir com o tempo. Desde a sua introdução, há mais de 30 anos, esta abordagem tem vindo a ganhar uma enorme popularidade (Halpin T. , 1998), razão pela qual tem sido, sistematicamente, sujeita a melhorias na sua notação gráfica. Neste trabalho focar-se-á exclusivamente a notação gráfica ORM de segunda geração, a qual é designada ORM2 (Halpin T. , 2008, p. 10).

A notação ORM2 é uma nova personificação da ORM, assim como, uma nova ferramenta de modelação de apoiar à notação ORM2 [**Anexo A** - Os Principais símbolos de Modelação em ORM2]. A promoção de informações de modelação da abordagem orientada por fatos foi encabeçado pela Universidade *Neumont* a cargo do Dr. Terry Halpin e, posteriormente através da Fundação ORM (Halpin T. , 2009).

A ORM2 é ao mesmo tempo um metamodelo que define um modelo ORM2 e uma ferramenta (NORMA) para a criação de modelos ORM2. A ferramenta NORMA *(Natural ORM Architect)* (Halpin T. , 2009) é um *plugin* gratuito e de código aberto *(open source)* para a *Microsoft Visual Studio* 2005/2008/2010/2012 e 2013. A

ferramenta mapeia modelos ORM/ORM2 para uma variedade de alvos de implementação, incluindo as principais bases de dados, código orientado por objetos e esquemas XML.

Os principais conceitos da ORM foi desenhado na ferramenta *Microsoft Visual Studio* 2010 com *plugin* NORMA.

Nas secções seguintes iremos ver, em detalhe, os principais conceitos da ORM, os quais estão esquematizados na figura 2.1. Esses conceitos são: fatos, tipo de fatos base e tipo de fatos derivados, tipo de objetos, relacionamentos, restrições e regras de derivação.

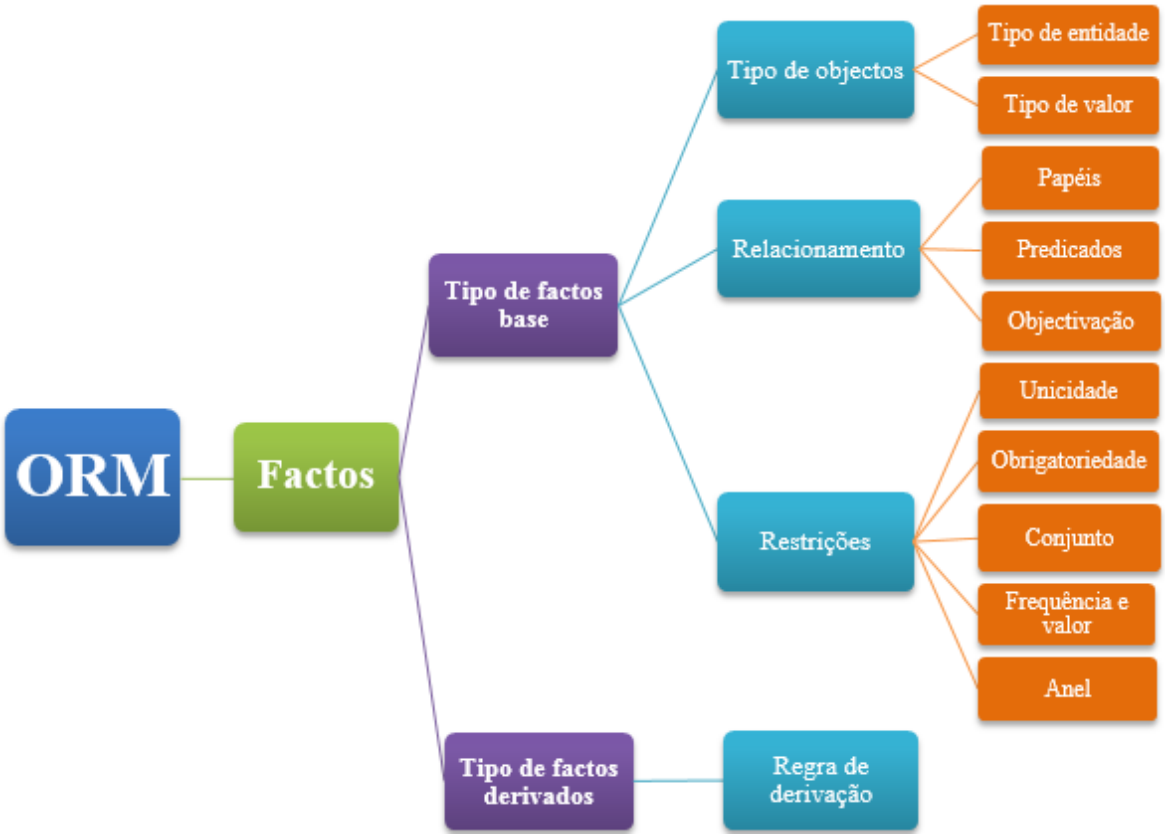


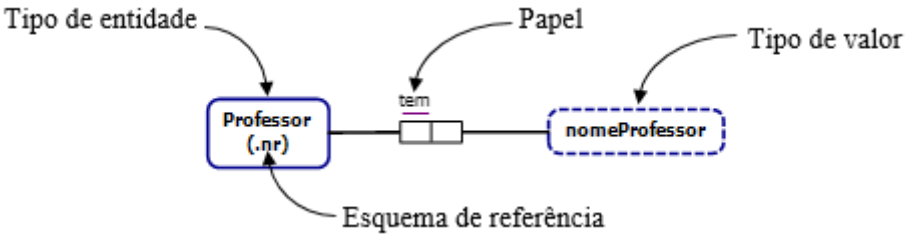
Figura 2. 1 - Os principais conceitos da ORM

2.3 Tipos de Objetos

Um **tipo de objetos** pode ser um tipo de entidades ou um tipo de valores. Uma **entidade** é uma descrição de objetos do mundo real. Por exemplo, a entidade designada por *Ambrósio* representa um objeto real, o professor que tem nome *Ambrósio*.

2.3.1 Tipo de entidades

Um **tipo de entidades** representa conceptualmente um conjunto de entidades com propriedades semelhantes. Por exemplo, o tipo de entidades *Professor* representa o conjunto formado por todos os professores, como mostra a figura 2.2.



Professor tem nomeProfessor.

Figura 2. 2 - O tipo de entidades Professor

Cada entidade é identificada, dentro dum tipo de entidade, através de um **esquema de referência**, o qual é especificado entre parênteses. Por exemplo, o tipo de entidade *Professor* tem o esquema de referência *nr* que representa o número de professor (ver figura 2.2).

2.3.2 Tipo de valores

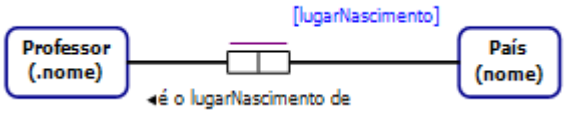
Um tipo de entidades é um conceito abstrato. De fato, não é possível, por exemplo, escrever um professor numa tabela sem identificar um professor em particular através de valores a ele associados como o seu número, o seu nome, etc. Estes valores são designados **tipos de valores**. Assim, um **tipo de valores** representa uma cadeia, um número, e assim sucessivamente, os quais serão efetivamente armazenados num repositório. Por exemplo, na figura 2.2 são ilustrados dois tipos de valores: o tipo de valores *nr*, correspondente ao número de um professor; e o tipo de valores *nomeProfessor*, o qual representa uma cadeia de caracteres correspondente ao nome de um professor.

2.4 Relacionamentos

2.4.1 Papéis

Num relacionamento entre tipos de objetos, cada tipo de objetos desempenha um determinado **papel** no relacionamento. A ORM usa o conceito de papel para representar os diferentes comportamentos de um objeto do mundo real. Assim, um papel descreve a

função de um tipo de objetos num relacionamento. Podemos associar a cada papel um nome. Por exemplo, *lugarNascimento* como é ilustrado na figura 2.3.



País é o lugarNascimento de Professor.

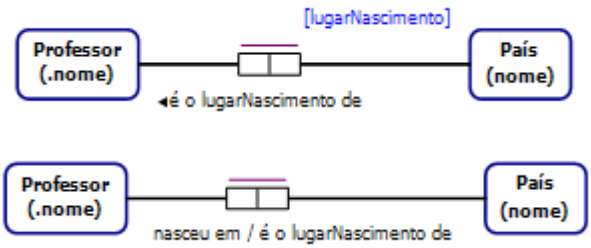
Figura 2. 3 - Nomes de papel lugarNascimento

Assim, no relacionamento ilustrado na figura 2.3, entre os tipos de objetos *Professor* e *País*, o papel de *País* tem o nome "*lugarNascimento"* e o papel de *Professor* não tem nome.

2.4.2 Predicados

ORM permite relacionamentos de qualquer aridade (número de papéis), cada um com pelo menos um predicado. Um **predicado** (Halpin T. , 2008, p. 67) é basicamente um provérbio declarativo com espaços para os objetos, e são indicados por reticências "...". Assim, completamos o provérbio preenchendo os espaços com as designações dos objetos (instâncias dos tipos de objetos).

Cada predicado (Halpin T. , 2008, pp. 83-87) é representado através de um nome, o qual reflete a ordem pela qual os diferentes tipos de objetos participam no predicado. Num relacionamento binário (dois papéis), existem dois predicados: um é chamado de predicado direto e o outro de predicado inverso. Por exemplo, no relacionamento binário na figura 2.4 entre os tipos de objetos *Professor* e *País*, existem dois predicados: o predicado nasceu em e o predicado *é o lugarNascimento de*.



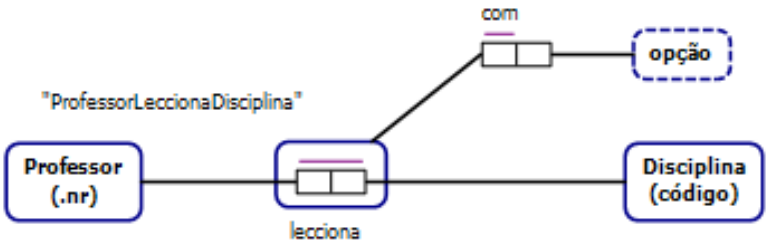
Professor nasceu em País.
País é o lugarNascimento de Professor

Figura 2. 4 - Leituras dos predicados direto e inversos

Notar que o predicado direto é o predicado que se lê da esquerda para a direita (... nasceu em ...) e que por isso, aparece à esquerda da barra "/". Por outro lado, o predicado inverso (... é o lugarNascimento de ...) lê-se da direita para a esquerda e, por isso, aparece à direita da barra "/" (ou, em alternativa, apresenta o símbolo "◄").

2.4.3 Objetivação

Os tipos de objetos podem desempenhar papéis de mais do que um tipo de fatos, por exemplo, a entidade *Professor* desempenha dois papéis, como mostra a ver figura 2.5.



Professor lecciona Disciplina.
ProfessorLeccionaDisciplina **objetivação** "Professor lecciona Disciplina".
ProfessorLeccionaDisciplina com opção.

Figura 2. 5 - Objetivação do tipo de fatos

Assim, o tipo de fatos tem de ser promovido a um tipo de entidade, chamado de **objetivação**. Por exemplo, o tipo de fatos mencionado acima, é objetivado *"Professor Lecciona Disciplina"*, e desempenha um papel o outro tipo de fatos.

Objetivar um tipo de relacionamento (binário, ternário, etc.) é percebido como um tipo de entidade que tem um esquema de referência composto cuja projeção da referência ostenta uma restrição de igualdade para o tipo de fato a ser objetivado.

Usa-se objetivação do tipo de fatos, quando se pretende especificar mais informações sobre o relacionamento entre as entidades. Por exemplo, *"Professor lecciona Disciplina com opção"*.

2.5 Restrições

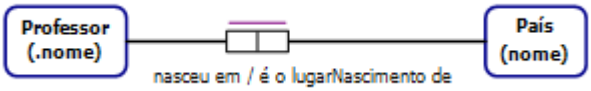
Uma **restrição** condiciona a quantidade de combinações possíveis de valores que podem ocorrer num relacionamento. Assim, por exemplo, uma restrição pode definir se todas as instâncias de um tipo de objetos participam ou não (obrigatoriedade) num determinado relacionamento. Uma restrição também pode restringir o número de

objetos (multiplicidade) de um determinado tipo que podem estar relacionados com um objeto doutro tipo num relacionamento.

As restrições mais usadas em modelos ORM são **restrições de unicidade** e **restrições de obrigatoriedade**. Existem dois tipos de restrições de unicidade: **restrições de unicidade internas** e **restrições de unicidade externas.**

2.5.1 Restrições de unicidade

Para compreender o conceito de restrição de unicidade é importante explicar o conceito de tabela de fatos. Consideremos, por exemplo, o tipo de fato *Professor nasceu em País*, o qual está ilustrado na figura 2.6. Este tipo de fato tem dois tipos de objetos, *Professor* e *País*, os quais são referenciados, respetivamente, pelos respetivos *nome* e *código*. Assim, próximo da representação gráfica do tipo de fato podemos construir a respetiva tabela de fatos. Nesta tabela, cada coluna corresponde a um tipo de objetos e cada linha corresponde a uma instância do tipo de fatos, isto é, a um fato. Assim, na figura 2.6 estão representados três fatos na tabela de fatos: *Ambrósio nasceu em Moçambique*, *Daniel nasceu em Portugal* e *Nuno nasceu em Portugal*.



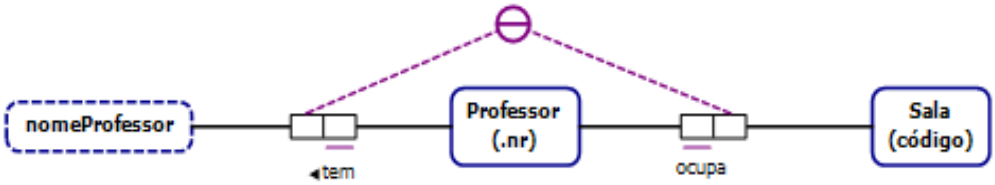
Nome do Professor	Nome do País
Ambrósio	Moçambique
Daniel	Portugal
Nuno	Portugal

Professor nasceu em País.
País é o lugarNascimento de Professor

Figura 2. 6 - A Restrição de unicidade (Halpin T. , 2008)

Uma **restrição de unicidade interna** indica que a instância de um tipo de objetos (ou combinações de tipos de objetos) pode aparecer, no máximo, uma vez na tabela de fatos. Notar que a restrição não se aplica aos tipos de objetos mas sim a tipos de objetos no contexto de um tipo de fato, isto é, a papéis. Uma restrição de unicidade interna é representada graficamente através de uma linha colocada ao longo do papel ou papéis sobre os quais a restrição se aplica. Por exemplo, na tabela de fatos da figura 2.6 cada uma das instâncias do tipo de objetos *Professor* (coluna *Nome do Professor*) não aparece mais do que uma vez. Isto é, cada professor nasceu no máximo num país. Esta

restrição está representada graficamente pela linha colocada sobre o papel do tipo de objetos *Professor*.



Nome	Número
Ambrósio	001
Nuno	002
Nuno	003

Número	Código
001	CP 2.7
002	S0.60
003	CP 2.7

Professor tem nomeProfessor.
Cada Professor tem **no máximo um** nomeProfessor.
Professor ocupa Sala.
Cada Professor ocupa **no máximo uma** Sala.

Restrição de unicidade externa
Para cada Sala **e** nomeProfessor,
no máximo um Professor ocupa **essa** Sala **e**
tem **esse** nomeProfessor.

Figura 2. 7 - A restrição de unicidade externa (Halpin T. , 2008)

Notar que cada uma das instâncias do tipo de objetos *País* (coluna *Código do País)* pode ocorrer duplicadas pois não existe uma restrição de unicidade associada ao papel respetivo.

Uma **restrição de unicidade externa** é uma restrição de unicidade que se aplica a dois ou mais papéis pertencentes a tipos de fatos diferentes. A restrição de unicidade externa faz a ligação de dois ou mais papéis sobre os quais a restrição se aplica. Por exemplo, a figura 2.7 mostra uma restrição de unicidade externa, aplicada a dois tipos de fatos: *Professor tem Nome* e *Professor está na Sala*.

As duas tabelas de fatos na figura 2.7 podem ser combinadas na tabela de fatos que se apresenta na tabela 2.1.

Número do Professor	Número	Código
001	Ambrósio	CP 2.7
002	Nuno	S0.60
003	Nuno	CP 2.7

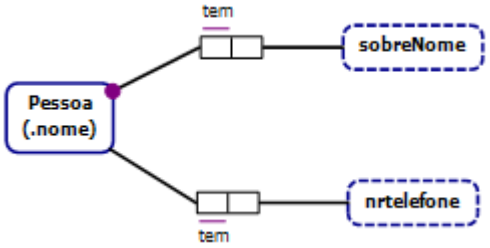
Tabela 2. 1 - Tabela de fatos resultante (Halpin T. , 2008)

A restrição de unicidade externa na figura 2.7 estipula que os valores relativos à combinação das colunas *Nome* e *Código* são únicos. Assim, por exemplo a combinação de valores (**Ambrósio S., CP2.7**) não poderá parecer mais do que uma vez. Isto é, não pode existir mais do que um professor na mesma sala com o mesmo nome.

2.5.2 Restrições de obrigatoriedade

Uma **restrição de obrigatoriedade (**ou **total)** indica que todas as instâncias de um tipo de objetos devem desempenhar um determinado papel. Pode-se destacar dois tipos de restrição de obrigatoriedade: restrição de obrigatoriedade simples e restrição de obrigatoriedade ou-inclusiva (disjuntiva).

A figura 2.8 ilustra uma restrição de obrigatoriedade simples do tipo de objetos *Professor*.

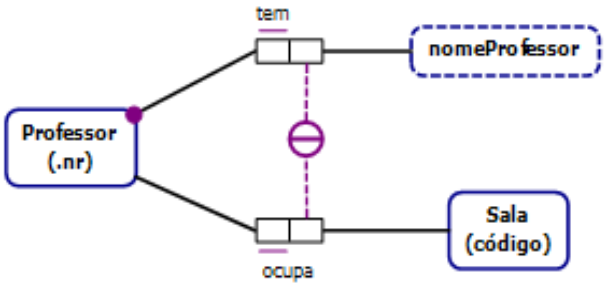


Restrição de Obrigatoriedade
Pessoa tem sobreNome.
Cada Pessoa tem **exatamente um** sobreNome.
Restrição de Unicidade
Pessoa tem nrtelefone.
Cada Pessoa tem **no máximo um** nrtelefone.

Figura 2. 8 - A restrição de obrigatoriedade (Halpin, 2008)

O ponto de obrigatoriedade ilustrada na figura 2.8 podia ser colocado numa extremidade do tipo de objeto ou numa extremidade do respetivo papel dependendo do tipo de notação a seguir. Por exemplo, todos os professores têm um *sobreNome*. Assim, a ligação do tipo de objetos *Professor* com o papel *tem sobreNome* tem o símbolo '●' numa das suas extremidades (neste caso, a extremidade relativa ao tipo de objetos).
Podemos distinguir dois tipos de esquemas de referência: esquema de referência simples e esquema de referência composto.
Um **esquema de referência simples** mapeia cada entidade para um valor único. Cada entidade é identificada por estar associada a um valor único. Por exemplo, um Professor pode ser referenciado pelo seu número (ver figura 2.2).

O conceito de **esquema de referência composto** está diretamente relacionado com os conceitos de restrição de unicidade externa e restrição de obrigatoriedade. De fato, consideremos o exemplo da figura 2.9. Um Professor que está na sala pode ser identificado não apenas pelo seu nome mais também pelo seu número. Por exemplo, a combinação do nome do Professor e a sala em que esta deve ser única. Assim, a figura 2.9 mostra uma restrição de unicidade externa aplicada aos tipos de objetos (mais precisamente, aos seus papéis) *nomeProfessor* e *Sala*.



Professor tem nomeProfessor.
Cada Professor tem **exatamente um** nomeProfessor.
Professor ocupa Sala.
Cada Professor ocupa **no máximo uma** Sala.

Restrição de unicidade externa composta
Para cada Sala **e** nomeProfessor,
no máximo um Professor ocupa **essa** Sala **e**
tem **esse** nomeProfessor.

Figura 2. 9 - Esquema de referência composto

A figura 2.9 ilustra o exemplo da tabela 2.1 com uma restrição de unicidade externa usando o número do professor como identificador primário.

Para além deste caso, o aspeto do identificador da referência composta ilustrado por uma restrição de unicidade externa com um identificador de preferência seria substituída o símbolo da figura por ⊜, sem o número como identificador primário.

2.5.3 Restrições de conjunto

Depois de explicados os conceitos relativos às restrições mais usadas (restrição de unicidade e restrição de obrigatoriedade), iremos agora explicar outras restrições, por sua vez, também importantes: restrição de valor, restrição de comparação de conjuntos e restrição de subtipo.

Uma **restrição de valor** (Halpin T. , 2008, pp. 216,217) (também chamada de **restrição de domínio**) indica o conjunto de valores possíveis que são permitidos para um tipo de

valores. Uma restrição de valor é representada graficamente especificando o conjunto de valores próximo da elipse que representa o tipo de objeto sobre o qual a restrição se aplica. Existem três formas distintas para especificar aquele conjunto de valores: enumeração, intervalo de valores e multiplicidade.

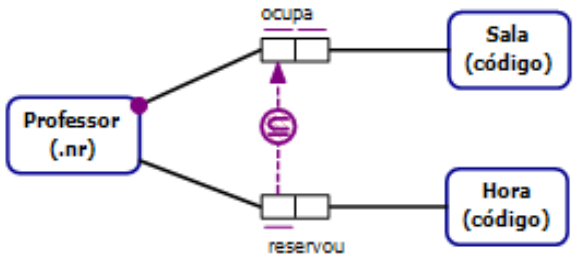
Uma **enumeração** identifica explicitamente o conjunto de valores possíveis para o tipo de valor. Ou se seja, o conjunto de valores admissíveis para o tipo de valores *código* para o tipo de entidades *sexo* pode ser especificado através de {'M','F'}.

Um **intervalo** identifica uma lista ordenada de valores possíveis onde se representa apenas os seus extremos. Por exemplo, o conjunto entre 1 e 10 cujo o extremo inferior é 1 e superior 10 pode ser representada por {1..10}. Pode se ainda fazer a inclusão ou exclusão do valor na lista. Por exemplo, o conjunto de números positivos de 0 a 100, excluindo o 0 é especificado através de {(0..100]}.

Uma **multiplicidade** permite combinar enumerações e intervalos numa restrição única. Por exemplo, o conjunto formado por letras, e por números é representado por {'a'..'z', 'A'..'Z', '0'..'10'}.

Uma **restrição de comparação de conjuntos** estabelece uma relação entre dois conjuntos de objetos de um determinado tipo. Ou seja, sobre um tipo de objetos consideramos um conjunto de objetos que desempenham um papel (r1) e sobre esse mesmo tipo de objeto consideramos outro conjunto que desempenha outro papel (r2), a restrição de comparação de conjunto irá estipular a relação entre eles. As restrições de comparação de conjuntos incluem as restrições de subconjunto, as restrições de igualdade e as restrições de exclusão.

Na figura 2.10 é ilustrada um exemplo de uma restrição de comparação de conjunto, nomeadamente a restrição de subconjunto. A figura 2.10 representa a restrição de subconjunto em que o conjunto de professores que reserva uma *Sala*, durante um período de tempo, tem de ser um subconjunto dos professores que têm atividades nessa *Sala*. Assim, o símbolo da restrição de subconjunto é colocado no sentido do papel do subconjunto (reservar) para o papel correspondente ao superconjunto (atividades).



Professor ocupa Sala.
Cada Professor ocupa **exatamente uma** Sala.
Para cada Sala, **ano máximo um** Professor ocupa **essa** Sala.
Professor reservou Hora.
Cada Professor reservou **no máximo uma** Hora.

Restrição de subconjunto
Se um Professor reservou **no máximo uma** Hora **então esse** Professor ocupa **exatamente uma** Sala.

Figura 2. 10 - Restrição de subconjunto.

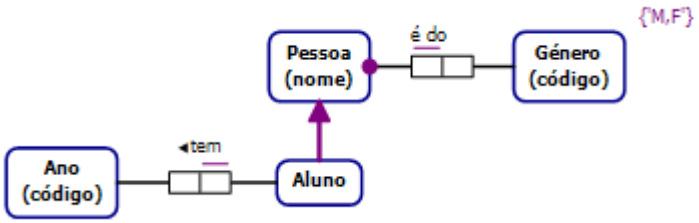
A restrição de igualdade é equivalente a restrição de subconjunto. A restrição de igualdade é feita em ambas as direções sem a indicação explícita do sentido.

Considera-se uma restrição de exclusão de dois ou mais papéis somente se esses papéis são opcionais e são desempenhados pelo mesmo tipo de objeto. Isto significa que os seus papéis devem ser separados (exclusão mútua).

Antes de passarmos para a restrição de subtipo, iremos primeiramente explicar os conceitos de **subtipo** e **supertipo.** Estes conceitos serão utilizados para a explicação do conceito de **restrições de subtipo**.

Quando num tipo de objetos observamos entidades com características próprias definimos um **subtipo** de objeto. Neste contexto, o **supertipo** identifica o tipo de objeto agregador. Por exemplo todos os objetos do subtipo *Aluno* também são do supertipo *Pessoa*, e que é de um determinado *Género* (ver figura 2.11).

Há tipos de fatos que não se aplicam a todos os objetos de um tipo de objeto, como no caso da figura 2.11, onde não se aplica o tipo de fato *"pessoa tem ano"* a todos os objetos do tipo pessoa. O conceito de subtipo aplica-se a tipo de objetos com estas propriedades. No exemplo tivemos de criar o subtipo *Aluno*. Uma **restrição de subtipo** implícita a este conceito é que todos os objetos do subtipo pertencem obrigatoriamente ao supertipo.



Restrição de subtipo

Cada Aluno é instância de Pessoa.
Pessoa é do Género.
Cada Pessoa é do **exatamente um** Género. **O valor possível do** Código do género **é** 'M,F'.

Figura 2. 11 - Restrição de subtipo

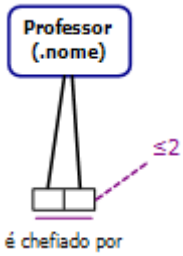
De notar que numa restrição de subtipo, o subtipo tem de ter tipos de fatos a ele associados. De fato, não há qualquer utilidade em adicionar um subtipo a um diagrama se depois não lhe for associado um ou mais tipos de fatos. O subtipo iria apenas aumentar a complexidade do diagrama. Conforme é demonstrado na figura 2.11 o subtipo *Aluno* têm somente um supertipo *Pessoa*, chama-se de **herança simples**. Caso contrário se o subtipo tiver dois ou mais supertipos, chama-se de **herança múltipla**.

2.5.4 Restrições de frequência

Uma **restrição de frequência** (também chamada de **restrição de frequência de ocorrência**) indica o número exato de vezes em que cada instância de um tipo de objetos pode desempenhar um determinado papel. Pode distinguir-se dois tipos de restrições de frequência: **restrições de frequência interna** e **restrições de frequência externa.**

Uma **restrição de frequência interna** é equivalente a restrição de unicidade interna. São restrições locais sobre o papel, e não restrições globais sobre o tipo de objeto.

Numa restrição de frequência interna, se o número de ocorrência for igual a um (=1), neste caso, a restrição seria representada pela barra de unicidade e não pela respetiva frequência. Caso contrário, a restrição é representada pelo valor da frequência sobre o papel. Por exemplo, a figura 2.12 ilustra este tipo de restrição, a qual estipula que cada professor é chefiado por, no máximo, dois professores.



Nome do Professor	Nome do Professor
Ambrósio	Paula
Ambrósio	António
José	Daniel
José	Paula

Professor é chefiado por Professor.

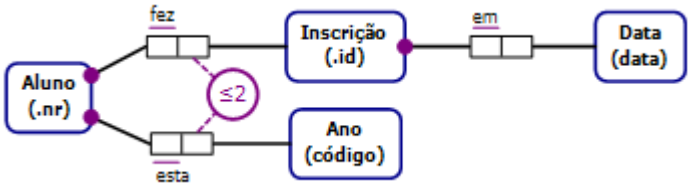
Restrição de frequência

Cada Professor$_2$ **em** "Professor$_1$ é chefiado por Professor$_2$" **ocorre no máximo** 2 **vezes**.

Figura 2. 12 - Restrição de frequência interna.

Finalmente, à semelhança do conceito de unicidade, também podemos generalizar o conceito de **restrição de frequência externa** de forma a permitir papéis de predicados diferentes.

Por exemplo, em alguns casos é permitido que o *Aluno* pode se inscrever no mesmo *Ano*, no máximo, duas vezes, caso eles não tenham transitado na primeira tentativa. São permitidos apenas mais uma única tentativa, como mostra a figura 2.13.



Aluno fez Inscrição.

Cada Aluno fez **exatamente uma** Inscrição.

Aluno esta Ano.

Cada Aluno esta **exatamente num** Ano.

Restrição de frequência

Para cada Ano **e** Inscrição,

essa combinação ocorre, no máximo 2 **vezes neste contexto**.

Figura 2. 13 - Restrição de frequência interna

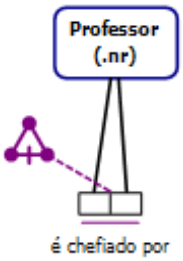
2.5.5 Restrições de anel

Uma **restrição de anel** indica quais os fatos que são válidos relativamente a um tipo de fatos quando um tipo de objetos tem um relacionamento com esse mesmo tipo de objetos.

O caminho de ida e volta passando pelo papel do tipo de objeto forma um "anel".

Este tipo de restrição tem um conjunto de propriedades que especificam a restrição. Pode-se especificar os seguintes tipos de propriedades: transitividade, reflexividade, simetria.

A figura 2.14 apresenta a restrição de anel intransitiva sobre o tipo de fato *Professor* **é chefiado por** *Professor*.



nrProfessor	nrProfessor
P1	P2
P2	P3
~~P1~~	~~P3~~

Professor é chefiado por Professor.

Restrição de Anel (intransitiva)

Se $Professor_1$ é chefiado por $Professor_2$ **e** $Professor_2$ é chefiado por $Professor_3$
então é impossível que $Professor_1$ é chefiado por $Professor_3$.

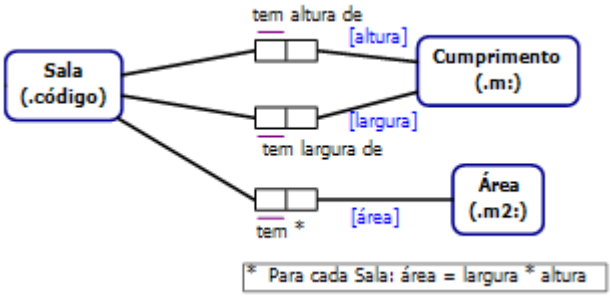
Figura 2. 14 - Restrição de anel intransitivo

A restrição de anel é intransitiva porque se um professor p_1 for chefiado pelo professor p_2 e o professor p_2 for chefiado pelo professor p_3, o professor p_1 não é chefiado pelo professor p_3.

2.6 Tipos de fatos derivados

Um **fato derivado** é uma preposição verdadeira que não é necessário ser armazenada na base de dados porque pode ser deduzida a partir de outras preposições já armazenadas. Por exemplo, a preposição *Sala1 tem Área de* $50m^2$ é um fato derivado porque a área da sala não é necessário ser armazenada na base de dados porque pode ser deduzida a partir da largura e altura da sala já armazenadas. Os fatos derivados similares são representados através de um **tipo de fato derivado**. Por exemplo, o fato derivado *Sala1 tem Área de* $50m^2$ é representado pelo tipo de fato derivado *Sala tem Área.*

A figura 2.15 apresenta o fato derivado e o tipo de fatos derivado para o cálculo da área de uma sala.



Sala tem altura de Cumprimento.
Cada Sala tem altura de **no máximo um** Cumprimento.

Sala tem largura de Cumprimento.
Cada Sala tem largura de **no máximo um** Cumprimento.

Sala tem Área.
Cada Sala tem **no máximo uma** Área.

Figura 2. 15 - Fato derivado e o tipo de fatos derivado (Halpin T. , 2008)

Cada tipo de fato derivado tem associado uma regra de derivação (Halpin T. , 2008, p. 33). Uma **regra de derivação** descreve textualmente como se define o fato derivado. A regra de derivação relaciona o tipo de fato derivado com outros tipos considerados base. A regra de derivação não tem representação gráfica em ORM. Assim, a ORM representa as regras de derivação utilizando anotações textuais.

A regra de derivação pode ser representada sob dois estilos (Halpin T. , 2008, pp. 98-100): O estilo de *atributo* e estilo *relacional*. Neste trabalho iremos utilizar o estilo de atributos, como no exemplo da figura 2.15, onde a *expressão 2.1* tem o seguinte significado: A equação *área = largura * altura* pressupõe uma restrição entre os tipos de papéis *área*, *largura* e *altura* de uma sala que se referem aos atributos do tipo de objeto *Sala*. Numa associação binária, o nome dos papéis *altura* e *largura* podem ser designados por atributos nominais do tipo de objeto *Sala*.

```
* Para cada Sala,
        área = largura * altura
```

2.7 Modelação de processos em ORM

Todo sistema de informação tem um ciclo de vida que é chamado de **Ciclos de vida de um Sistema** (Halpin T. , 2008). O desenvolvimento de um sistema envolve cinco diversas etapas (ver figura 2.16): especificação, desenho, implementação, teste e manutenção.

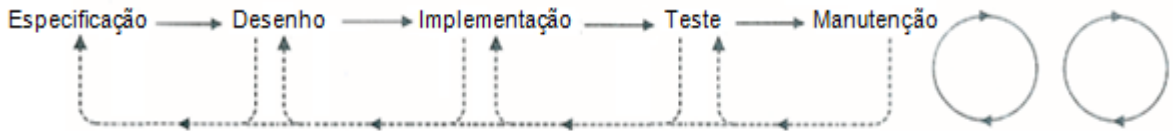


Figura 2. 16 - Ciclo de vida de um sistema

A um encadeamento específico dessas etapas para a construção do sistema dá-se o nome de modelo de ciclo de vida. Há diversos modelos de ciclo de vida. Nesta secção iremos utilizar o **modelo iterativo**. O ciclo das etapas é feito progressivamente, na medida em que, são feitas repetições sucessivas do ciclo de vida, com refinamentos progressivos no sistema.

Iremos somente focar a etapa de **desenho** que descreve as fases específicas da ORM utilizadas na modelação de processos em ORM. A especificação dos requisitos define o problema e os documentos produzidos durante o desenho especificam uma solução particular para o problema. A etapa de desenho representa um processo criativo de transformar o problema numa solução. A etapa de desenho descreve um procedimento iterativo feito em duas partes: **Desenho conceptual** (desenho do sistema) e **desenho lógico** (desenho técnico).

O **desenho conceptual,** também conhecido por "análise", é um esquema que indica o que o sistema irá fazer, ou seja, a origem e tratamento dos dados, duração dos eventos, o aspeto do sistema, etc.

O desenho conceptual compreende três principais secções: tipos de fatos, restrições e regras de derivação explicadas acima.

Se o desenho conceptual for corretamente planeado, ele fornece um modelo da estrutura do sistema designado por **desenho lógico**. Uma vez concluída a análise, pode-se realizar um mapeamento do desenho conceptual para um desenho lógico. Por exemplo, podemos mapear um modelo conceptual para um modelo relacional.

O método de modelação em ORM inclui uma notação e procedimentos para essa mesma notação. Esse procedimento chama-se **Processo de Desenho do Esquema Conceptual (CSDP)**. O esquema conceptual particulariza a estrutura da informação da aplicação através de **tipo de fatos**, **restrições** e **regras de derivação**. É aplicada a cada módulo o CSDP, e os consequentes sub-esquemas são integrados no esquema conceptual global.

O desenho do esquema conceptual é realizado em sete passos. A tabela 2.2 mostra os passos do **CSDP**.

Passo	Descrição
1.	Transforma a informação em fatos elementares, e verifica a sua qualidade.
2.	Desenhe o tipo de fatos.
3.	Verifica a existência de tipos de entidade que devem ser combinados, e assinala qualquer derivação aritmética.
4.	Adicionar restrições de unicidade, e verificar aridade do tipo fatos.
5.	Adicionar restrições de obrigatoriedade de papel, e verificar se há derivações lógicas.
6.	Adiciona valores, restrição de comparação de conjunto e restrição de subtipo.
7.	Adiciona outras restrições e executa as verificações finais.

Tabela 2. 2 - O processo de desenho do esquema conceptual (CSDP)(Halpin T. , 2008)

O procedimento que envolve os sete passos começa com a análise de informações que pretendemos obter no final (saída), ou a informação que pretendemos introduzir (entrada), no sistema de informação. Basicamente, os três primeiros passos estão direcionados em identificar os tipos de **fatos elementares** e **derivados**.

O **Passo 1** é o mais crítico. A informação necessária para o sistema é verbalizada em linguagem natural e transformada para fatos elementares. Por exemplo, *Ambrósio nasceu em Moçambique.*

O **Passo 2** consiste em desenhar os **tipos de fatos** no diagrama do esquema conceptual. Assim, são especificados os **tipos de objetos**, os **predicados** e os **esquemas de referência.** Por exemplo, o tipo de objetos *Professor* é referenciado pelo número de professor e tem um nome de professor (ver figura 2.2).

O **Passo 3** identifica os **tipos de fatos derivados**. Por exemplo, o fato derivado *Sala01 tem Área de 50m*^{2} é identificado pelo tipo de fato derivado *Sala tem Área* (ver figura 2.15).

Nos passos seguintes são adicionadas as restrições para os tipos de fatos.

O **Passo 4** especifica as **restrições de unicidade** e verifica a aridade (unária, binária, …) dos tipos de fatos. Por exemplo, para o fato '*Ambrósio nasceu em Moçambique'* a restrição de unicidade pode ser especificada como '*Cada Professor nasceu no máximo num País'*.

O **Passo 5** especifica as **restrições de obrigatoriedade**. Por exemplo, todos os professores têm um *Sobrenome*. A restrição de obrigatoriedade pode ser especificada como *'Cada Professor tem um Sobrenome'* (ver figura 2.8).

O **Passo 6** adiciona as **restrições de valor** (*enumeração, intervalo de valores e multiplicidade*), por exemplo, o código para o tipo de entidades *Género* pode ser representado através dos valores *'M'* e *'F'* (Ver figura 2.11), as **restrições de comparação de conjuntos** (*subconjunto, igualdade e exclusão*) (ver figura 2.10) e as **restrições de subtipo** (ver figura 2.11).

O **Passo 7** adiciona as **restrições de anel**. Por exemplo, o fato *Professor1* **é chefiado por** *Professor2* pode ser representado através de uma restrição de anel do tipo de fato *Professor* **é chefiado por** *Professor* (ver figura 2.14), e faz verificações finais dos passos anteriores.

A aplicação do CSDP ao caso de estudo deste trabalho está descrita detalhadamente no **Anexo C**.

Capítulo 3 – Paradigma Orientado por Objetos (POO)

Este capítulo descreve o outro tópico de investigação deste trabalho, no seguimento do capítulo anterior. O tópico em questão é o Paradigma Orientado por Objetos (POO). Assim, o desenvolvimento deste tema permite uma introdução à Linguagem de Modelação Unificada (UML) que será aplicada no presente trabalho e uma análise comparativa sobre as duas abordagens.

Este capítulo está estruturado da seguinte forma. A secção 3.1 descreve detalhadamente o POO. A secção 3.2 incide sobre a UML e suas características. A secção 3.3 descreve o conceito de elemento. A secção 3.4 expõe os vários tipos de relacionamentos. A secção 3.5 descreve a metodologia aplicada na modelação de processos com recurso à UML. A secção 3.6 descreve a modelação de processos em UML e a secção 3.7 apresenta as conclusões sobre os paradigmas ORM e UML.

3.1 Introdução

O Paradigma Orientado por Objetos (POO) (Silva & Vieira, 2001) (Rumbaugh, Jacobson, & Booch, 1999) é uma abordagem que traduz a forma de analisar a realidade que nos rodeia. Este paradigma traz para o domínio do desenvolvimento de sistemas de software uma abordagem à semelhança do mundo real em que os conceitos retratam os acontecimentos diários. Para representar esta realidade, o paradigma recorre ao conceito de objeto que incorpora dados (atributos) e um conjunto de operações (métodos) que manipulam estes dados.

Esta abordagem na sua perspetiva orientada a objetos apresenta algumas vantagens comparativamente à abordagem estruturada (Silva & Vieira, 2001, pp. 67-108), das quais se destacam:

- Uma melhor organização do código e contribuição para o reaproveitamento de código.
- Reduzir a interdependência de código e facilita a sua manutenção.
- Uma das características próprias do POO é a **coesão**. Este mecanismo permite melhorar a inter-relação entre o conjunto de recursos de uma classe.

- O conceito de **ocultação de informação**, mas conhecido por *Information hiding*, consiste em evitar o acesso aos dados de um objeto que não são necessários para a compressão do seu funcionamento, permitindo apresentar apenas ao exterior o essencial.
- O conceito de **herança** próprio do POO envolve a possibilidade de uma nova classe, chamada de subclasse ou classe derivada, ser criada com base numa classe existente, chamada de superclasse ou classe base. A classe derivada herda todos as características da classe base, podendo a classe derivada "aumentar" e/ou refinar a classe base, acrescentando detalhes específicos próprios da classe.
- O conceito de **polimorfismo** presente no POO constitui uma característica que permite as classes derivadas invocarem métodos da superclasse.

O POO (Silva & Vieira, 2001) surgiu associado à programação no final dos anos sessenta, com a linguagem *Simula*. Nos anos setenta, a linguagem *Smalltalk,* desenvolvida pela *Xerox*, posicionou o POO como um novo paradigma. Até meados da década de oitenta este paradigma apenas foi utilizado a nível de programação. Contudo, na década de noventa, assistiu-se à proliferação de métodos e notações para modelação associados ao POO, dos quais se destacam os métodos de *Booch*, OMT e OOSE. Após uma época de inúmeras propostas, designada por fragmentação, surge a necessidade de normalização com contributos de vários parceiros (Silva & Vieira, 2001). Este esforço no âmbito da OMG (Object Management Group, 2010) levou ao aparecimento do UML como padrão para a modelação de software em meados de 1997. Esta abordagem pode ser compreendida com uma nova forma de encarar e modelar o mundo exterior. Atualmente, existe um vasto conjunto de linguagens para a modelação de software segundo o POO, tais como: a *OPEN modeling language* (OML) (Firesmith, Graham, & Henderson-Sellers, 1998), a *Unified Modeling Language* (UML) (Rumbaugh, Jacobson, & Booch, 1999), *Object Modeling Technique* (OTM) (Rumbaugh, Jacobson, & Booch, 1999) e *Object-Oriented Software Engineering* (OOSE) (Rumbaugh, Jacobson, & Booch, 1999). Porém, a UML é a abordagem mais popular e a mais aplicada em modelação de software. Neste contexto, esta secção irá centrar-se na UML para apresentar os conceitos associados ao POO, e aos processos de desenvolvimento de software.

3.2 *Unified Modeling Language* (UML)

A *Unified Modeling Language* (UML) (Rumbaugh, Jacobson, & Booch, 1999) (Silva & Vieira, 2001) é uma linguagem padrão para especificação, construção, visualização e documentação de produtos de um sistema.

A UML é particularmente uma linguagem adequada para modelar sistemas que variam desde os sistemas de informação empresarial, aplicações distribuídas baseadas na Web e até mesmo sistemas embebidos em tempo real.

A UML (Rumbaugh, Jacobson, & Booch, 1999) foi introduzida na década de 1990. Após a primeira versão da UML, as revisões posteriores passaram a ser da responsabilidade da OMG RTF *(Revision Task Force)*. Este grupo assume um papel essencial na resolução de inconsistências, ambiguidades e omissões, bem como na introdução de novas características na UML. A UML tem vindo a evoluir, razão pela qual tem vindo a sofrer mudanças nas suas versões, como mostra a figura 3.1.

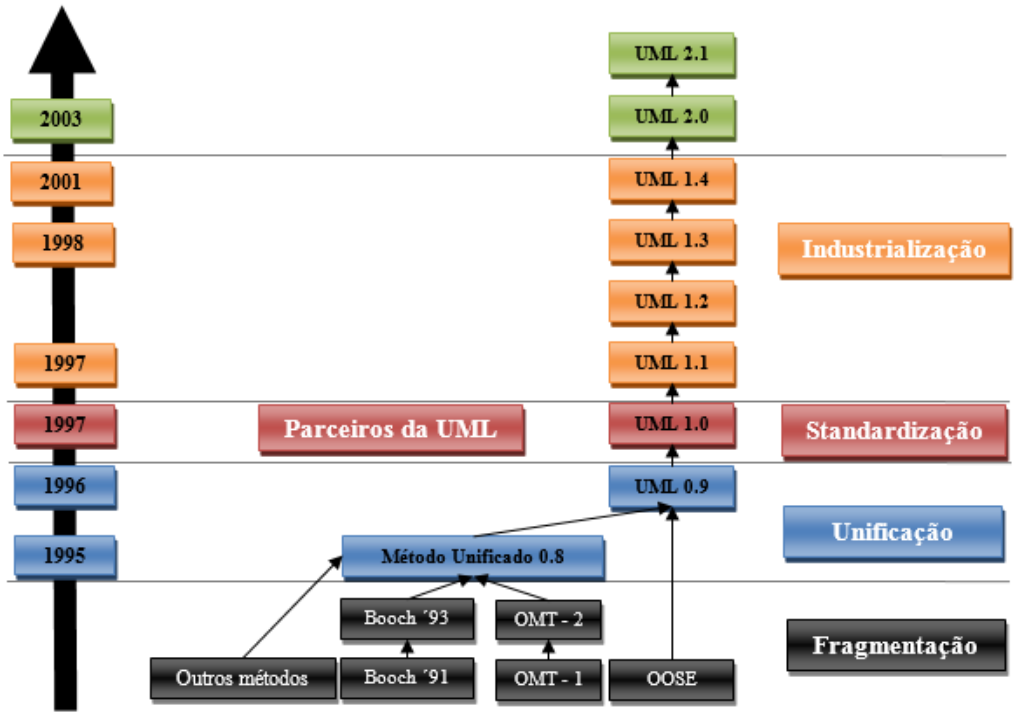


Figura 3. 1 - Visão histórica da UML (Silva & Vieira,

A figura 3.1 apresenta uma visão histórica da evolução das linguagens de modelação orientadas por objetos até à sua unificação, em particular destacamos a UML. Por outro lado, uma vez que a versão mais recente da UML é a versão 2.1, este trabalho focar-se-á exclusivamente nos conceitos associados à UML 2.1.

O POO recorre a um conjunto de modelos para representar determinadas características dos elementos e seus relacionamentos, sendo o modelo uma abstração de um sistema associado ao problema que se pretende solucionar. A descrição de um modelo pode ser feita através de diagramas ou linguagem natural. No contexto da UML o conjunto de modelos permite representar as várias perspetivas do domínio do sistema, tendo cada um deles, um objetivo específico.

No entanto, apesar de a UML suportar uma variedade de modelos desde a sua fase de análise até a fase de desenho, nomeadamente:

- Os casos de utilização para a especificação de requisitos;
- Os diagramas de classes e diagramas de objetos para a modelação da estrutura. Sendo o diagrama de classes usado para a modelação da estrutura que pretende representar e o diagrama de objetos para representar o relacionamento do elementos;
- O diagrama de sequência para a modelação do comportamento onde se descreve a interação entre objetos ao longo do tempo;
- O diagrama de instalação para a modelação da arquitetura onde se descreve aspetos de implementação dos componentes de softwares na infraestrutura de hardware.

O foco deste trabalho não é a análise detalhada dessa variedade de modelos, uma vez que se pretende fazer a transformação de modelos da ORM para diagramas de classe da UML. Como a ORM trabalha só a nível dos dados, este capítulo apenas focará os diagramas de classes e os diagramas de objetos, bem como os conceitos associados a esses mesmos diagramas.

O diagrama de classes é um modelo que apresenta a estrutura estática (domínio) de um sistema representando os vários elementos e seus relacionamentos.

O diagrama de objetos é um modelo que descreve como os elementos de um determinado grupo (classe) estão relacionados entre si. Ambos os diagramas oferecem uma notação gráfica formal para a modelação de elementos e seus relacionamentos.

Nas secções 3.3, 3.4 e 3.5 apresenta-se detalhadamente os principais conceitos da UML, aplicados neste trabalho os quais estão ilustrados no esquema da figura 3.2 (elementos básicos e relacionamentos).

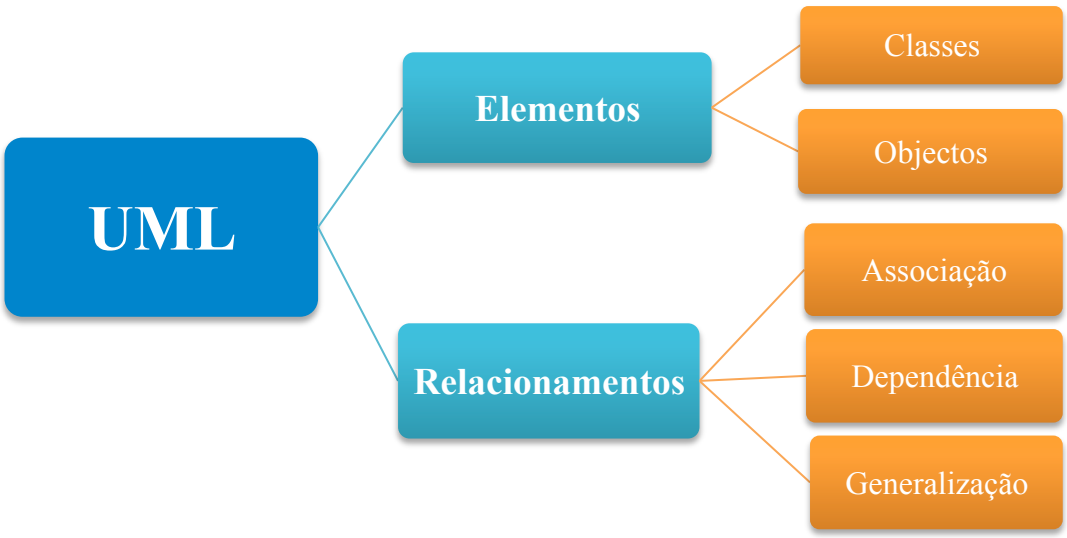


Figura 3. 2 - Os principais conceitos da UML

3.3 Elementos básicos

Um **elemento** é a descrição de um conceito do mundo real com uma existência independente. Podemos distinguir vários tipos de elementos: pacote, classe, interface, objetos, enumeração e tabelas.

O objetivo deste trabalho passa por descrever somente o que se pode relacionar com elementos de ORM que são: classe, interface, objeto e enumeração.

3.3.1 Classe

Uma **classe** representa um elemento do mundo real caracterizado por atributos, métodos e com uma linguagem semântica comum. Uma classe contém três secções distintas. Na primeira secção apresenta-se o nome da classe, na segunda a lista dos atributos e na terceira a lista dos métodos, como mostra a figura 3.3.

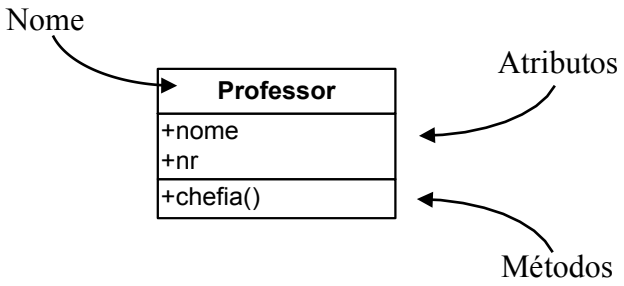


Figura 3. 3 - Classe Professor

O nome da classe é uma designação que identifica com exatidão o que a classe representa. Por exemplo, a classe *Professor* representa o agrupado de elementos formados somente por professores.

Os atributos da classe representam um conjunto de propriedades partilhadas por todos os elementos que constituem a classe. Por exemplo, os elementos da classe *Professor* têm um *nome*. O atributo é, portanto, uma abstração do tipo de dados ou de um estado do elemento da classe que a abrange. Os métodos da classe representam ações realizadas pelos elementos de uma classe podem realizar. Por exemplo, o elemento da classe *Professor* realiza uma ação que é *chefia*.

3.3.2 Objetos

Um **objeto** (Rumbaugh, Jacobson, & Booch, 1999) é uma instância de uma classe, que contém todos os *atributos* e *métodos* definidos na própria classe. Por exemplo, o elemento *ambrósio* é uma instância da classe *Professor* (ver figura 3.4).

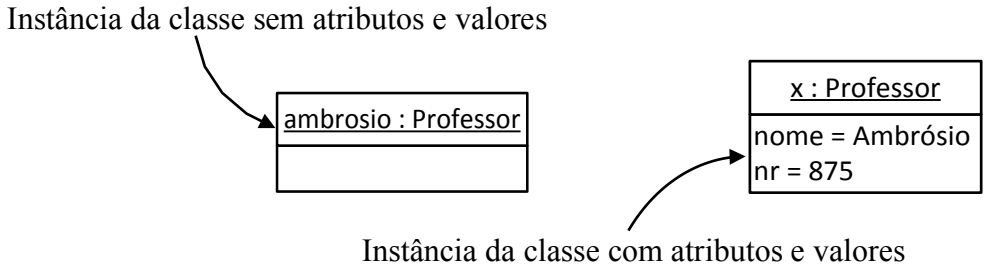


Figura 3. 4 - Representação gráfica de objeto

Cada objeto possui identidade e, portanto, é diferenciado dos demais. A alteração na condição de um objeto pode identificar alterações no seu comportamento vulgarmente conhecido por *estado*. Assim, o estado de um objeto corresponde simplesmente à sua condição atual, a qual pode ditar mudanças no seu comportamento.

Portanto, um estado pertence a exatamente uma classe e é representado pelos valores dos atributos que um objeto da classe pode adquirir. Varia constantemente durante a

interação do objeto com outros objetos. Porém, um estado em UML descreve o estado interno de um objeto para uma classe em particular. É de salientar que nem toda a modificação num dos valores de um atributo pode ser considerada de estado, mas somente, alterações significativas no comportamento do objeto.

Outro conceito importante a citar são os **tipos de dados** (também chamados **tipos primitivos**) representam valores que não possuem identidade e não estão sujeitos a efeitos paralelos. Geralmente representam conceitos de um domínio matemático e seus valores são inalteráveis.

É importante salientar que o valor armazenado por um atributo pode ser atualizado, mas o valor em si não. Em UML, os tipos primitivos predefinidos são: numéricos, *strings* e booleanos.

Existe outro tipo de dados que são definidos pelo utilizador, chamados de *enumeração*. O utilizador pode especificar cada um dos valores (distintos) deste tipo. Quaisquer métodos definidos para enumerações devem ser rotulados *(tagged values)* especificado através de *«enum»*, para garantir que os valores não podem ser alterados (ver figura 3.5).

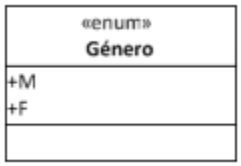


Figura 3. 5 - Representação gráfica de objeto

O exemplo da figura 3.5 ilustra o tipo de dados enumerados para o Género, a qual indica que possui apenas dois valores, masculino e feminino especificado através de 'M','F'.

3.4 Relacionamento

A definição de classes e suas instâncias é apenas o princípio do processo de modelação. Para a implementação de um produto de software é essencial que o modelo de um sistema represente também a interação entre os objetos do domínio do problema designada por **relacionamento**.

Um relacionamento tem designações diferentes no contexto de diagramas de classes e diagramas de objetos. O relacionamento entre duas classes designa-se por associação e entre dois objetos designa-se por ligação.

A UML oferece diversos mecanismos para expressar estas relações, sendo as principais as **associações**, **dependências** e as **generalizações**. As associações e dependências caracterizam as relações estruturais entre as classes enquanto a generalização cria uma taxonomia entre as entidades.

3.4.1 Associação

A **associação** indica um relacionamento estrutural de ligação de os objetos de uma classe com objetos da outra classe. Por exemplo, uma associação entre a classe *Professor* e a classe *País* (*o professor 'Ambrósio' nasceu em País 'Moçambique'*). Isso significa que uma instância de *Professor* terá uma ligação com uma instância de *País*.

A figura 3.6 ilustra uma associação em que, um *Professor* está associado a um *País*.

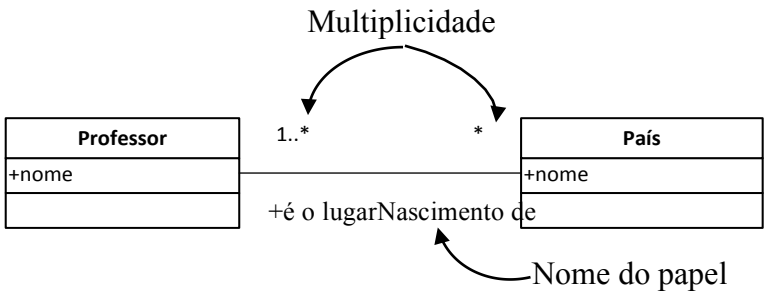


Figura 3. 6 - Associação entre a classe Professor e classe País

Uma associação pode ter um nome, e usa-se esse nome para descrever a natureza do relacionamento. Quando uma classe participa em uma associação, ela tem um papel específico na associação. Pode-se explicitamente atribuir o nome ao papel. Por exemplo, *'é o lugarNascimento de'* como é ilustrado na figura 3.6. Assim, na associação entre os elementos da classe *Professor* e *País*, o papel de *País* tem o nome *'é o lugarNascimento de'* e o papel de *Professor* não tem nome. Para que não haja ambiguidade sobre o seu significado, pode-se dar um sentido ao nome, apontando a direção de leitura que se pretende.

A associação pode ter várias aridades (unária, binária, etc.).

A mesma classe pode desempenhar o mesmo ou diferentes papéis em outras associações. Em muitas situações de modelação, é importante que se indique quantos

objetos podem ser ligados através de uma instância de uma associação, que se designa por **multiplicidade** (ver 3.6) de um papel de associação, e é escrito como uma expressão que avalia um intervalo de valores ou de um valor explícito para cada objeto da classe.

Podemos mostrar uma multiplicidade de exatamente *um (1), zero ou mais (0...1), muitos (0...*),* ou *um ou mais (1...*).* Podemos até indicar um número exato (por exemplo, 3.7).

Existe um tipo especial de uma associação chamada de **agregação**. Num relacionamento de associação com agregação podemos distinguir dois tipos de agregação: agregação simples e agregação composta ou composição.

A agregação simples é um tipo de relacionamento com características *"é parte de"*, como mostra a figura 3.7.

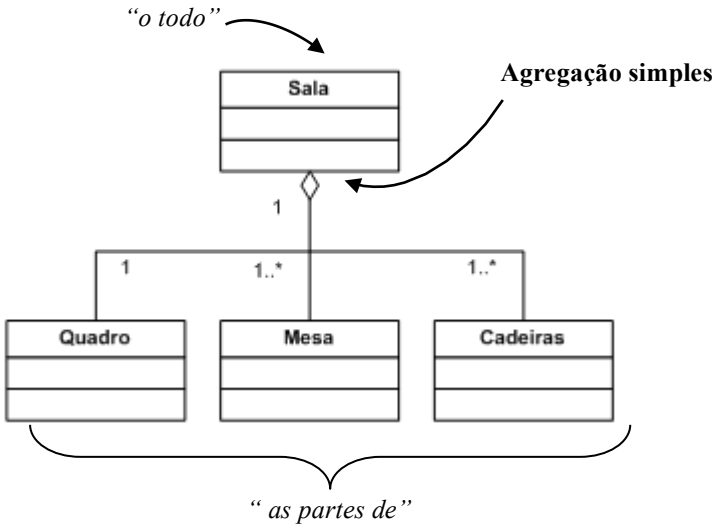


Figura 3. 7 - Associação por agregação simples

Neste tipo de associação, existe um grau de coesão entre as partes, podendo haver um certo grau de independência entre eles. Por exemplo, uma sala pode existir sem quadro, mesas e cadeiras, e vice e versa. Mas, todos os componentes juntos podem formar a "sala de aulas",

O outro tipo de agregação, composta ou composição é um tipo de relacionamento com características *"todo/parte"*, onde existe um elevado grau de coesão entre o todo e as partes, com total grau de dependência entre eles. Desta forma, se o todo não existir, as partes também não existirão. Por exemplo, considerando o exemplo tirado do nosso caso de estudo, reflete o fato que *"um Departamento não existe fora do contexto da Escola"*, como mostra a figura 3.8.

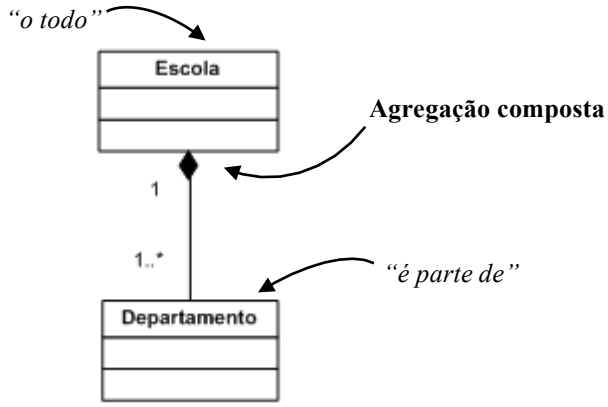


Figura 3. 8 - Associação com agregação composta

A figura 3.8 ilustra uma agregação composta entre a classe *Escola* e a classe *Departamento*. Uma agregação composta indica que uma parte pode pertencer a apenas um todo e que a vida útil do conjunto determina a vida útil do conjunto.

3.4.2 Dependência

Uma **dependência** é um relacionamento entre dois tipos de elementos. Por exemplo, na dependência da classe *Professor* para a classe *Departamento*, o *Professor número '875'* tem uma dependência para o *Departamento* de *'Informática'* com *código '550'* (figura 3.9).

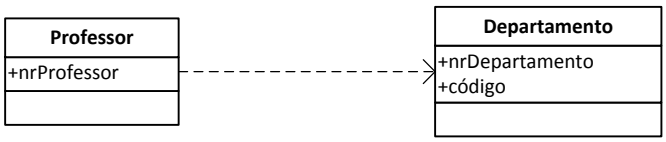


Figura 3. 9 - A dependência entre a classe Professor e a classe Departamento

Este conceito indica que a alteração na especificação de um elemento de uma classe pode afetar o elemento da classe dependente.

3.4.3 Generalização

Uma **generalização**, também conhecida como herança, é um relacionamento entre elementos generalistas (chamada de superclasse ou pai) e elementos específicos e especializados (chamado de subclasse ou filho). A subclasse herda as propriedades e métodos da superclasse. Por exemplo, todos os elementos da subclasse *Aluno* e da subclasse *Professor* herdam as características da superclasse *Pessoa*, como mostra a figura 3.10.

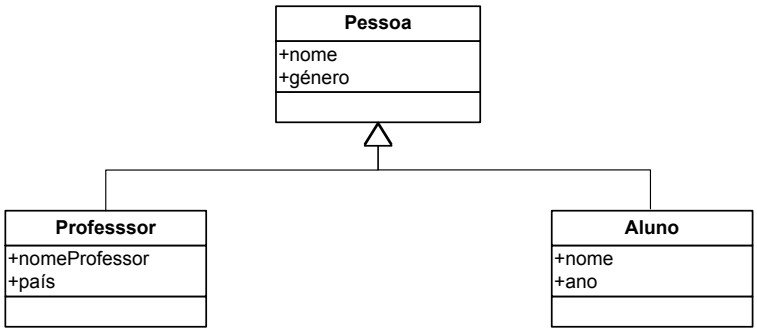


Figura 3. 10 - Generalização da superclasse Pessoa e as subclasses Professor e Aluno

Por outro lado, a generalização significa que o filho é substituível pelo pai (*polimorfismo*). O filho herda as propriedades dos seus pais, em especial os seus atributos e métodos. Mas, nas subclasses os atributos e os métodos herdados podem ser modificados. Para complementar, novos atributos e métodos podem ser adicionados às subclasses.

3.5 Restrições

Uma **restrição** é uma condição semântica ou declarações expressas textualmente. A restrição pode ser declarada sob forma de uma linguagem natural, ou em diferentes tipos de linguagens com uma semântica bem definida.

Em UML, uma restrição consiste na especificação de uma condição que permitem complementar ou alterar o significativo de qualquer elemento num relacionamento. Esta condição pode ser especificada em UML através de um dos mecanismos: **linguagem informal**, **linguagem formal** ou **linguagem de programação.**

No entanto, existem restrições predefinidas em UML, por exemplo, a restrição de subconjunto.

As declarações textuais explícitas que denotam alguma propriedade particular a ser satisfeita pelas entidades são consideradas de **restrições**, sendo expressas nos modelos UML através de chavetas *{...}*.

3.5.1 Restrições informais

Uma restrição informal é uma condição baseada numa linguagem natural, por exemplo, texto escrito em português (ver figura 3.11). Note-se que uma restrição não é só uma afirmação, um mecanismo executável, mas sim, indica uma condição que deve ser

imposta pelo sistema, por exemplo, quando determinamos que "*um professor, para chefiar um departamento tem também de ser, necessariamente, membro desse departamento"*, estamos a introduzir uma restrição no modelo, conforme mostra a figura abaixo.

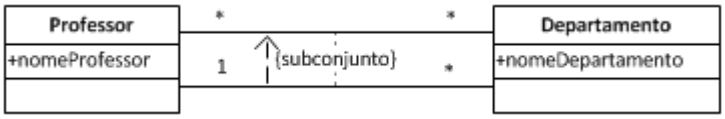


Figura 3. 11 - Restrição informal em UML

A figura 3.11 ilustra um exemplo de aplicação de uma restrição informal por recurso à linguagem natural, descrita através da restrição de subconjunto {subconjunto} predefinida em UML.

3.5.2 Restrições formais

Uma restrição formal é uma restrição baseada no cálculo de predicados, por exemplo, a linguagem formal *Object Constraint Language* (OCL) (OMG, Object Management Group, 2012)**,** (Rumbaugh, Jacobson, & Booch, 1999) (ver figura 3.12).

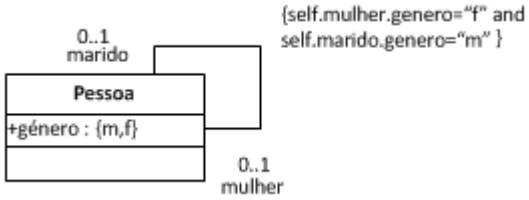


Figura 3. 12 - Restrição formal (OCL) em UML (OMG, 2012)

A figura 3.12 ilustra um exemplo de utilização de uma restrição formal OCL, em que se especifica que *"uma pessoa pode estar casada apenas com outra pessoa do sexo oposto"*.

A UML foi projetada para ser uma linguagem extensível. Para além das **restrições**, descritas anteriormente, oferece outros mecanismos: os **estereótipos** *(stereotypes)* e os **valores etiquetados** *(tagged values)*.

Os **estereótipos** permitem que uma semântica diferente seja associada a uma construção já existente. Ao rotular o classificador *Género* com *enumeração* (ver figura 3.5), fica estabelecido que este agora possui um significado diferente do de uma classe como *Pessoa* (figura 3.12).

Por fim, os **valores etiquetados** representam valores que podem ser associados a um elemento do modelo, introduzindo, implicitamente, uma restrição, por exemplo, através

de uma etiqueta e um valor pode ser definida um *tagged value* para expressar a versão de uma determinada classe.

3.6 Modelação de processos em UML

A crescente complexidade dos sistemas de software aumenta as necessidades em relação às ferramentas e metodologias de suporte aos processos de desenvolvimento de software. Os modelos de processos suportam o desenvolvimento de software, na medida em que facilitam a abstração dos procedimentos que gerem o projeto de software. Neste domínio, a UML é uma linguagem visual de análise e desenho de software importante para o desenvolvimento de software de qualidade. Segue-se uma breve descrição das características base de um processo de desenvolvimento de software.

O desenvolvimento de software divide-se essencialmente em cinco fases (ver figura 3.13): **análise de requisitos, análise e desenho do sistema, implementação (programação) do sistema** e **testes**.

Figura 3. 13 - Ciclo de vida de um sistema orientado por objetos, UML

No que diz respeito a este trabalho, iremos centrar a atenção as fases de **análise** e de **desenho**, uma vez que se pretende analisar o diagrama de classes da UML na modelação de sistemas de software. Contudo, e não sendo o objetivo deste trabalho, existem diversos modelos de processos de desenvolvimento de software (Rumbaugh, Jacobson, & Booch, 1999).

A figura 3.13 descreve o modelo iterativo aplicado neste trabalho. Neste tipo de modelo, as iterações sucessivas apresentação ações repetidas, de forma a que o resultado final seja um produto de qualidade. Assim, problemas detetados num determinado momento permitem modificações e/ou melhorias em relação à iteração anterior. Neste caso, não existirá risco elevado de reformulação de todo o trabalho, desde o seu inicio.

Depois da identificação dos utilizadores e levantamento de requisitos (funcionais e não funcionais) através dos "casos de utilização" (Rumbaugh, Jacobson, & Booch, 1999) (Silva & Vieira, 2001), a fase de análise permite identificar as primeiras entidades

(classes) e mecanismos presentes no domínio do problema. O modelo de domínio representado no diagrama de classes, inclui as classes candidatas bem como as relações entre elas.

Na fase de **desenho**, os modelos de análise são refinados e expandidos, surgem classes de suporte e novos relacionamentos. O objetivo é construir uma infraestrutura técnica que forneça detalhes para a fase seguinte, de implementação.

Para concluir o estudo do trabalho relacionado, a secção final apresenta uma comparação entre os aspetos descritos nas capítulos 2 e 3 sobre as abordagens analisadas.

3.7 Comparação dos paradigmas ORM e UML

Esta secção sintetiza os principais tipos de abordagens estudadas neste trabalho e analisa as suas semelhanças e diferenças. Proporciona uma comparação básica de ambas as abordagens e descreve o processo de conversão das notações. Os conceitos e a notação associados a estas duas abordagens fazem parte integral dos capítulos anteriores, 2 e 3 respetivamente. As subsecções seguintes incidem principalmente nas **estruturas de dados**, nos **relacionamentos**, e nas **restrições** e **regras de derivação**.

3.7.1 Estrutura de dados

Uma vez feita a descrição detalhada das abordagens estudas, um dos objetivos do trabalho é comparar e identificar as características comuns ou especificas destas linguagens de modelação. Segue-se a comparação dos elementos estruturais.

Os *objetos* e os *valores de dados* da UML são representados em ORM, respetivamente, por *entidades*, e *valores* (figura 2.1). Contudo, o que a ORM classifica como *tipo de entidade*, em UML é representado por uma *classe*. Os *tipos de dados* em UML correspondem basicamente a *tipos de valores* na ORM.

Na ORM, cada *entidade* é identificada pelo seu *esquema de referência* para permitir a comunicação com esta entidade, por exemplo, a entidade *professor* é identificada pelo número do professor *"(.nr)"*.

Em ORM cada *tipo de valor* e cada *esquema de referência* são convertidos para *atributos* da *classe* UML correspondente. Por exemplo, o tipo de valor *nomeProfessor* corresponde ao atributo *"nome{P}"*, e o esquema de referência *"(.nr)"* corresponde ao

atributo *"nr."*. A figura 3.14 exemplifica os conceitos acima mencionados em ORM e UML. Cada *classe* UML tem *atributos*, os quais são obrigatórios e têm um *único* valor.

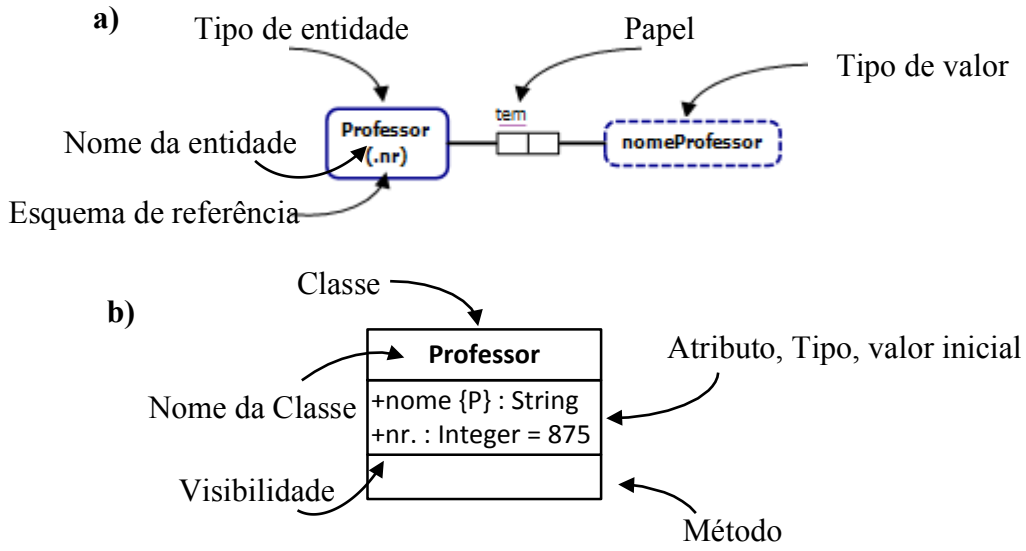


Figura 3. 14 - Resumo de toda a estrutura de dados em ORM a) e UML b)

O mecanismo de *visibilidade* ilustrado na figura 3.14 b), representado pelos símbolos *"+", "-", "#", "~"* indicam se o atributo da classe são *públicos*, *privados*, *protegidos* ou *pacote*. (Rumbaugh, Jacobson, & Booch, 1999). Estes conceitos estão relacionados com a implementação de software e não se apresentam como relevantes ao nível conceptual, nesse sentido não são representados no modelo conceptual ORM.

3.7.2 Relacionamento

Outra característica importante, na comparação entre as duas linguagens é o **relacionamento**. A ORM modela o mundo em termos de *fatos* e *papéis* e, portanto, tem apenas uma estrutura de dados, o *tipo de relacionamento*, que pode ser: *papéis, predicados e objetivação* (ver figura 2.1). Em UML um *relacionamento* pode ser do tipo: *associação, dependência e generalização* (ver figura 3.2).

Os *tipos de objetos* em ORM *(tipo de entidades e tipo de valores)* têm um *nome* e desempenham *papéis*. As *classes* em UML têm um *nome* (designação), tem *atributos*, e também pode ter *operações* (implementadas como *métodos*) e desempenham *papéis*. Não focamos os detalhes das *operações* próprias das classes UML, uma vez que estamos apenas a estudar a perspetiva dos dados. Assim, iremos centrar a atenção nos *atributos*.

Em particular, os *atributos* não são usados em ORM. Esta é a diferença fundamental entre ORM e UML. Para cada *atributo* em UML (ver figura 3.15 b)), a ORM utiliza um *tipo de relacionamento* (ver figura 3.15 a)). Esta particularidade torna os diagramas ORM mais "espaçosos" que os diagramas UML correspondentes, conforma mostra a figura 3.15. Contudo, apesar deste fato, existem grandes vantagens evidentes, as quais foram discutidas anteriormente no capítulo 2.

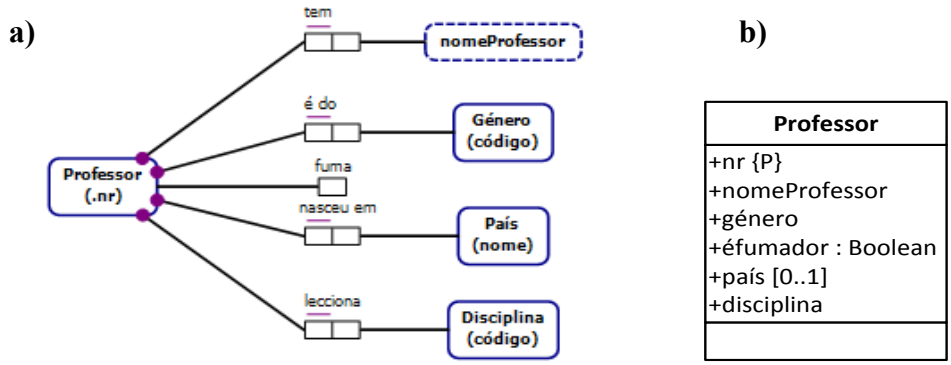


Figura 3. 15 - Tipos de relacionamentos em ORM a) representados com atributos em UML b)

A figura 3.15 a) apresenta o modelo ORM, que representa o tipo de entidade *Professor*, e seus tipos de relacionamento, cada professores é identificado pelo número de professor "*(.nr)"*; cada professor deve (ponto de obrigatoriedade do papel) ter um nome, género, país de nascimento e a disciplina que leciona. O modelo ORM ilustra um *predicado unário "fuma"*. A ausência de ponto de obrigatoriedade no primeiro papel do tipo de fato *"fuma"* indica que este papel é opcional (nem todos os professores fumam). O modelo UML correspondente (figura 3.15 b)) mostra a correspondência dos tipos de entidades *Género*, *Pais* e *Disciplinas* em *atributos* da classe *Professor*. Em UML, os *tipos de entidades* são representados como *atributos*. A UML não sustenta *relacionamento unário*, por isso, modela o *predicado unário* como *atributos booleanos*, por exemplo, *eFumador: boolean*. No que diz respeito às *associações*, apresentamos de seguida a comparação entre as duas notações. A *instância de relacionamento* em ORM é chamado de *link* em UML. Um *tipo de relacionamento* em ORM é chamado de uma *associação* em UML (por exemplo, *Professor* leciona uma *Disciplina*). Tanto em ORM como em UML, um *papel* representa os diferentes comportamentos de um *relacionamento*, o número de papéis de um *relacionamento* é a sua aridade. A ORM permite relacionamentos de qualquer *aridade* (número de papéis), cada um com pelo

menos uma leitura ou nome de predicado. Um relacionamento *n-ário* pode ter até *n* leituras (iniciada em cada papel), facilitando a verbalização das restrições e navegação em qualquer direção.

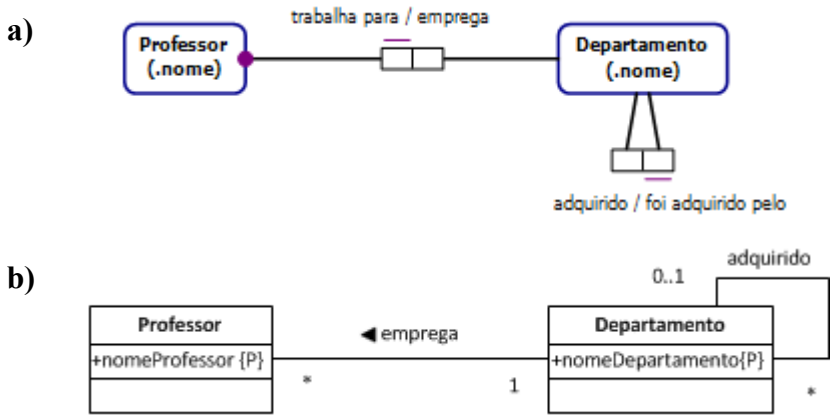


Figura 3.16 - Correspondência de restrições de multiplicidade em ORM a) e UML b) numa associações binário

A UML usa *atributos booleanos* em vez de *tipo de relacionamentos unários*, característicos da ORM, mas permite relacionamentos com várias aridades. Para representar a *multiplicidade* nos relacionamentos, a ORM utiliza as *restrições de unicidade* e de *obrigatoriedade*, como mostra a figura 3.16 a). No modelo ORM, O ponto de obrigatoriedade do papel indica que o papel "*trabalha para"* é indispensável. Isto é, para ser professor tem necessariamente de trabalhar para uma escola.

A UML representa as *restrições de multiplicidade* nas associações, longe do papel o que estão relacionadas, na direção em que a associação é lida.

Para *associações binárias*, existem quatro possibilidades para as **restrições de unicidade** *(n:1,1:n; 1:1, m:n)* e quatro possibilidades para os **papéis obrigatórios** *(só o papel da esquerda é obrigatório; apenas o papel direito é obrigatório; ambos os papéis são obrigatórios e ambos os papéis são opcional)*. Portanto, existem 16 combinações de *multiplicidade* possíveis de *associações binárias*. Os quatro primeiros casos são apresentados na figura 3.17, abrangendo as situações em que ambos os papéis são opcionais. As restrições são lidas da esquerda para a direita.

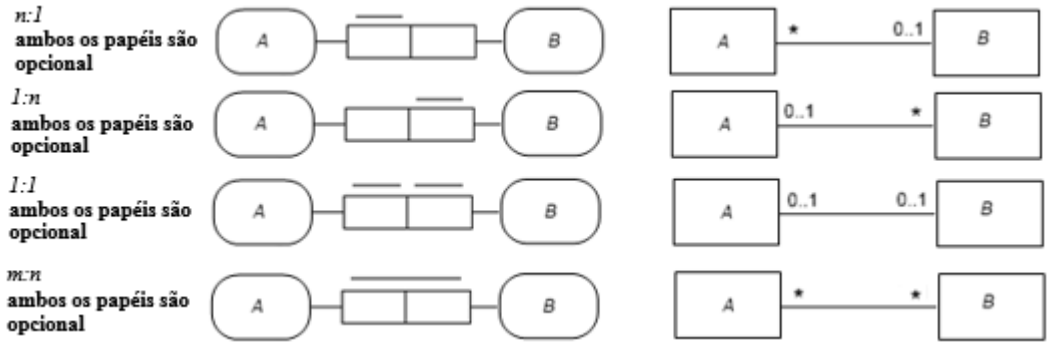


Figura 3. 17 - Equivalência de padrões de restrição em ORM e UML, os dois papéis são opcionais (Halpin T. , 1998)

Os próximos quatro casos, apresentados na figura 3.18, demonstram as situações em que o primeiro papel é obrigatório e o segundo papel é opcional.

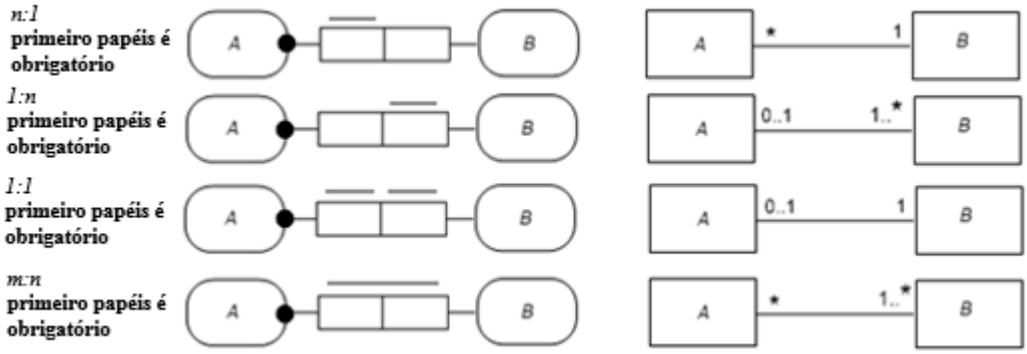


Figura 3. 18 - Equivalência de padrões de restrição em UML e ORM, primeiro papel é obrigatório (Halpin T. , 1998)

Na sequência, os próximos quatro casos, ilustrados na figura 3.19, abrangem as situações em que o segundo papel é obrigatório e o primeiro papel é opcional.

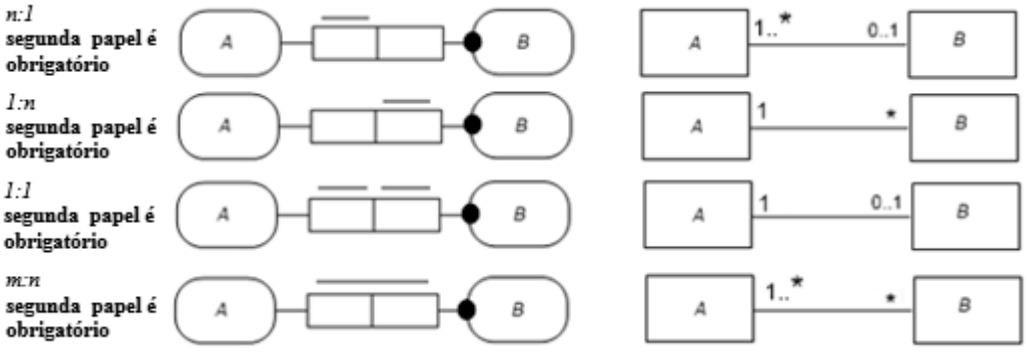


Figura 3. 19 - Equivalência de padrões de restrição em UML e ORM, segundo papel é obrigatória (Halpin T. , 1998)

Finalmente, a figura 3.20 abrange os últimos quatro casos em que ambos os papéis são obrigatórios.

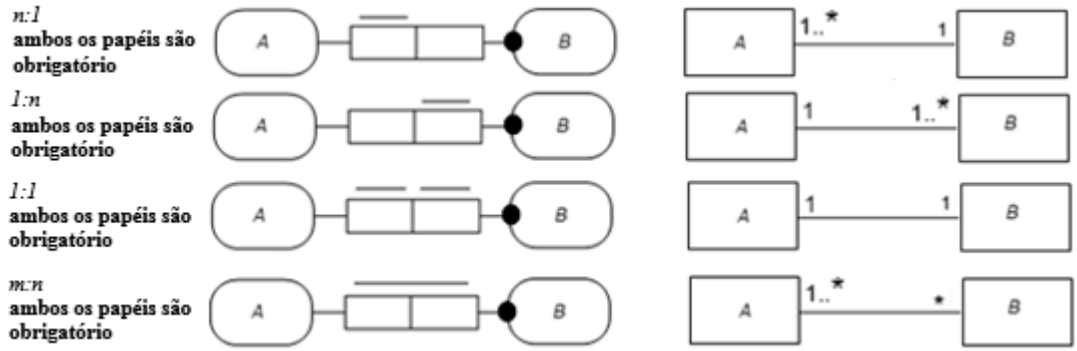


Figura 3. 20 - Equivalência de padrões de restrição em UML e ORM, dois papéis são obrigatórios (Halpin T. , 1998)

As restrições de *multiplicidade de atributos* indicam quantos objetos podem ser ligados através de uma instância de uma associação. Os vários tipos de *multiplicidade* servem para mostrar uma *multiplicidade de exatamente um (1), zero ou mais (0...1), muitos (0...*), ou um ou mais (1...*)*, ou ainda até para indicar um número exato (por exemplo, 3). Para estes casos, a UML e a ORM usam *restrições de multiplicidade* associadas aos papéis. Contudo, a ORM é mais expressiva, uma vez que pode aplicar tais restrições a coleções de papéis.

Uma *restrição interna* aplica-se a papéis numa única *associação*. Numa associação *n-ária* simples, cada *restrição de unicidade interna* deve cobrir pelo menos *n-1 papéis*. A ORM e a UML integram todas as possíveis *restrições de unicidade interna*.

ORM e UML, ambas, permitem a objetivação de associações como objetos de primeira classe, designados por *tipos de objetos aninhados* em ORM e *classes associação* em UML. Em vez de distanciar a objetivação da associação subjacente, a ORM representa graficamente uma *objetivação* envolvendo a *associação* com um *tipo de objeto*. Esta é exibida entre aspas *" "* indicando que um *tipo de fatos* é descrito como um *tipo de entidade*. Por exemplo, *"ProfessorLeccionaDisciplina"* como é ilustrado na figura 3.21 a).

A UML representa graficamente as *associações de classe* separadamente, fazendo a ligação por uma linha tracejada (ver figura 3.21 b)).

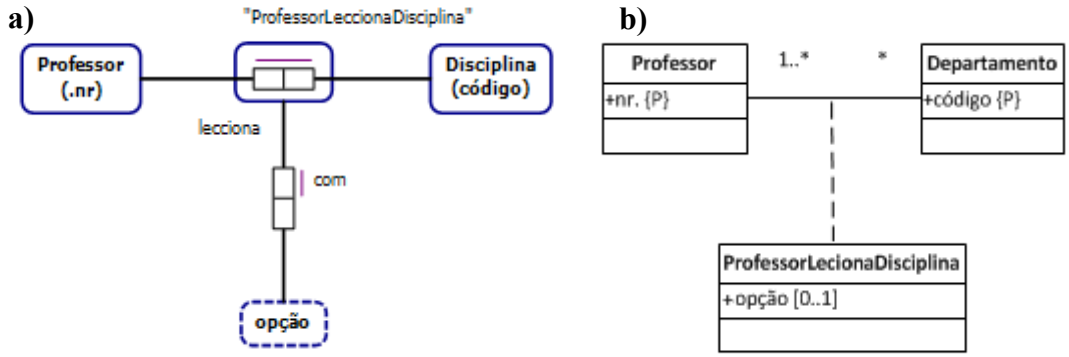


Figura 3. 21 - "ProfessorLeccionaDisciplina" é representado como uma objetivação de associação em ORM a) e UML b) (Halpin T. ,

A **tabela 3.1** resume as correspondências principais entre a modelação conceptual de dados em ORM e UML. O símbolo "—" indica que não existe correspondência predefinida entre os conceitos.

Instâncias / Estruturas	
ORM	**UML**
Entidade	Objeto
Valor	Valor de dados
Objeto	Objeto ou Valor de Dados
Tipo de Entidade	Classe
Esquema de referência	Atributo
Tipo de Valor	Atributo
Tipo de Objeto	Classe ou Tipo de Dados
— *{utilização de tipo de relacionamento}*	Atributos
Tipo de Relacionamento unário	— *{utilização de atributos Booleanos}*
2+-ary tipo de relacionamento	Associação
Associação de tipo de objeto	Classes de Associação

— = *Sem correspondência dos conceitos*

Tabela 3. 1 - Correspondência básica entre os conceitos conceptuais da ORM e UML para estrutura de dados (Halpin & Bloesch,

3.7.3 Restrições

O tópico restrições foi detalhado anteriormente nos capítulos 2 e 3. Referiu-se que as restrições mais usadas em modelos ORM são as *restrições de unicidade (interna e externa)* e as de *obrigatoriedade.*

A UML sendo uma linguagem extensível, possui construtores para as *restrições*, mas também para os *estereótipos* e *valores etiquetados*. As *restrições* são especificadas *informalmente* e *formalmente* de acordo com as condições da restrição.

Para corresponder as *restrições de unicidade interna* (ver figura 3.22) às *restrições de obrigatoriedade* (ver figura 3.23) em ORM, a UML utiliza as *restrições de multiplicidade*.

a) **b)**

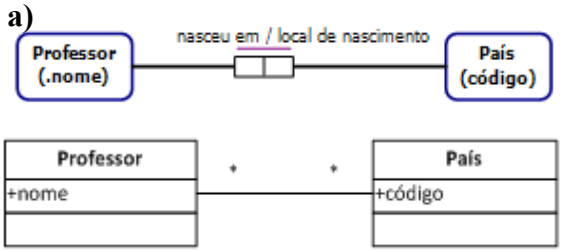


Figura 3. 22 - Restrição de unicidade em ORM a) e restrições de multiplicidade em UML b)

a) **b)**

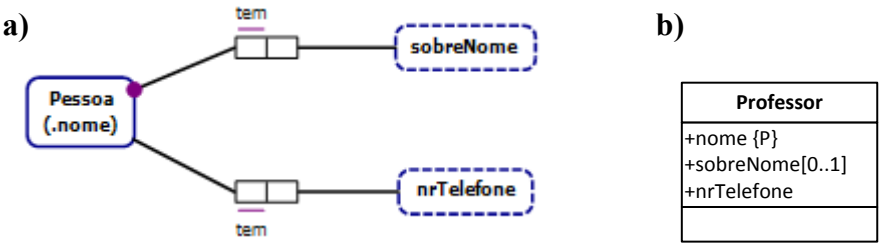


Figura 3. 23 - Restrição de obrigatoriedade em ORM a) e restrições de multiplicidade em UML b)

Para as *restrições de unicidade externa* em ORM, a UML não tem uma correspondência direta entre os conceitos, para isso, usa *associação quantificadas* (Halpin T., 2008) para representar estas restrições. Além das restrições ORM anteriormente mencionadas, a ORM permite outras restrições igualmente importantes na modelação de dados: as **restrições de conjunto,** as **restrições de frequência** e as **restrições de anel**. As *restrições de conjunto* incluem as *restrições de valor*, *restrição de comparação de conjuntos* e *restrição de subtipo*.

Nesta secção apresenta-se um estudo comparativo destas restrições em ORM e UML. Como correspondente às *restrições de valor* ORM, a UML usa o tipo de dados *enumeração* e a representação textual da restrição *(restrição textual)*. Em ORM, a *restrição de valor* (ver figura 3.24 a) limita os valores permitidos ao *tipo de valor*, a um conjunto finito de valores (*enumeração*), a intervalo de valores (*intervalo*), ou a uma combinação dos anteriores (*mistura*). Esta restrição é representada graficamente, colocando os valores possíveis dentro de chavetas *" {...}"* ao lado do *tipo de entidade*, ou *tipo de valor.* Para mais detalhes consulte o capítulo 2.

O tipo de dados *enumeração* em UML pode ser modelado como uma *classe*, estereotipado com *"«enumeration»"*, sendo os seus valores apresentados como *atributos*. O *intervalo* e *mistura* podem ser descritos com a declaração de uma *restrição textual* entre chavetas, através de linguagem formal ou informal, ver figura 3.24 (b).

As *restrições de valor*, como as *restrições de enumeração*, *intervalo* e *mistura*, podem ser declaradas em ORM tal como são em UML, ou seja, através da *restrição textual*. Para mais detalhes, consulte (Halpin T., 2008).

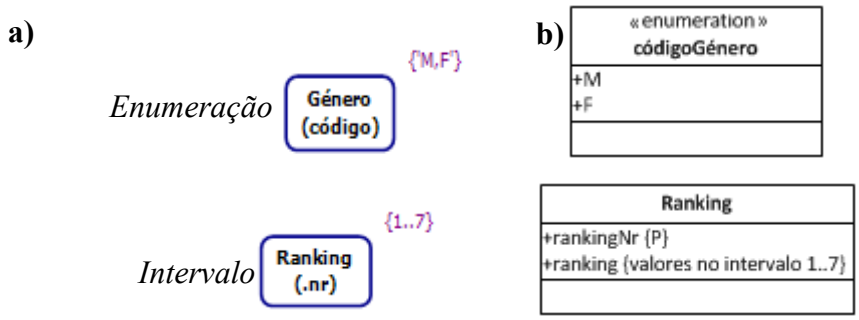


Figura 3. 24 - Restrição de valor em ORM a) e enumeração ou restrições textuais em UML b)

Uma *restrição de comparação de conjuntos* estabelece uma relação entre dois conjuntos de objetos de um determinado tipo. Os vários tipos de restrições de comparação de conjuntos são: *restrições de subconjunto*, *restrições de igualdade* e *restrições de exclusão*. Iremos apresentar o exemplo de uma *restrição de comparação de conjunto*, nomeadamente a *restrição de subconjunto*. A descrição detalhada destas restrições podem ser consultadas em (Halpin T. , 2008, pp. 369-371).

Uma *restrição de subconjunto* em ORM é definida entre qualquer par de sequências de *papéis*, ligados entre si, através de um símbolo de restrição de subconjunto. Este símbolo é colocado no sentido do papel do *subconjunto* fonte para o papel do *superconjunto* alvo. Por exemplo, a restrição de subconjunto na figura 3.25 a) indica que *"qualquer professor que chefia um departamento deve ser um membro desse departamento"*.

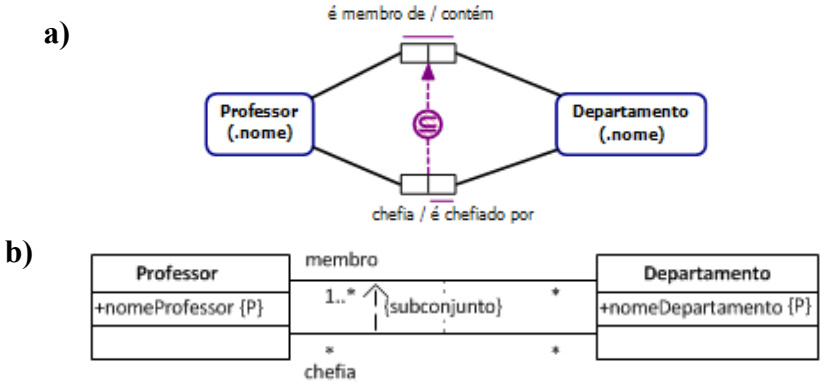


Figura 3. 25 - Restrição de subconjunto em ORM a) e UML b)

No entanto, a UML não tem uma notação gráfica para as *restrições subconjunto* entre os *papéis* ou partes de *associações*. Portanto, a UML tem um mecanismo de extensibilidade que permite expressar uma restrição de subconjunto. Este mecanismo consiste em anexar um comentário que inclui uma *restrição textual "{subconjunto}"* ao lado de uma seta tracejada entre as associação, como é ilustrado na figura 3.25 b).

Ambas as linguagens suportam *subtipos ("is-a")*, onde cada instância de um *subtipo* é também uma instância do seu *supertipo*. Em ORM, o *subtipo* herda todos *papéis* do seu *supertipo*. Em UML, o *subtipo* herda todos os *atributos*, *associação* e *operações/métodos* do seu *supertipo*. A introdução destes conceitos foi apresentada nos capítulos 2 e 3.A ORM introduz o conceito de *subtipos*, e a UML o conceito de *generalização*. Tanto ORM como UML permitem *herança simples*, assim como *herança múltipla* (em que, um *subtipo* tem mais de um *supertipo* direto).

As *restrições de subtipo* em ORM e UML, são representadas graficamente através de uma seta a ligar aos seus *supertipos*. A figura 3.26 mostra o exemplo de *restrição de subtipo*, na qual indica que todos os objetos do *subtipo Aluno* também são do *supertipo Pessoa*, e que é de um determinado *género* (masculino *"M"* ou feminino *"F"*) (exemplo retirado do capítulo 2).

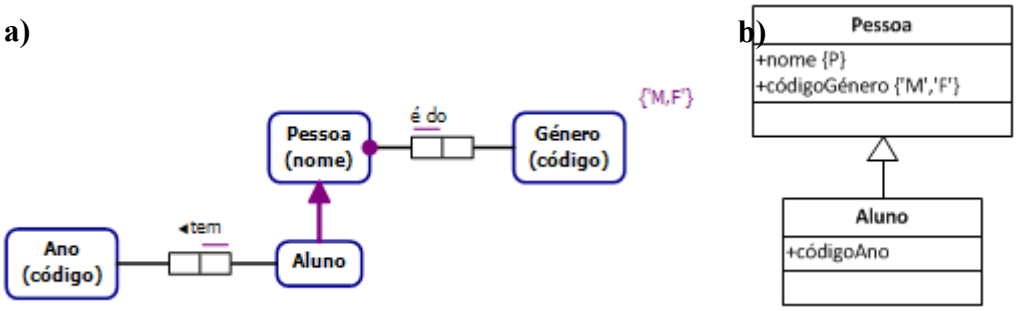


Figura 3. 26 - Restrição de subtipo em ORM a) e UML b)

Por defeito, os *subtipos* em ORM podem sobrepor-se, quando existem dois ou mais *subtipos* para um *supertipo*. Para resolver esta situação, a ORM permite adição de restrições gráficas para indicar que os *subtipos* são *mutuamente exclusivos* (Halpin T. , 2008, pp. 372-376).

A UML fornece um fraco suporte (*generalização*) para definir *subtipos* semelhantes aos definidos em ORM, uma vez que a correspondência dos conceitos é *"incompleta"*, ou seja, nem todos os *subtipos* foram especificados para UML (ver tabela 3.2) (Halpin T. , 2008, pp. 372-376).

Foi referido acima que a UML usa *restrições de multiplicidade* ao invés de *restrições de frequência (interna e externa)* características da ORM (capítulo 2). A figura 3.27 ilustra uma *restrição de frequência*, a qual determina que cada professor é chefiado, no máximo, por dois professores.

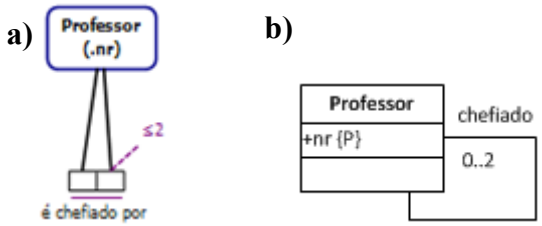


Figura 3. 27 - Restrição de frequência interna em ORM a) e UML b)

A *restrição de frequência externa* é em tudo semelhante a *restrição de frequência interna*, desta forma aplicado a papéis de predicados diferentes.

Outro tipo de restrição anteriormente mencionada foi a de *anel*. Em ORM, esta restrição indica que um *tipo de fato* tem pelo menos dois papéis desempenhados pelo mesmo *tipo de objeto* (seja diretamente, ou indiretamente através de um *supertipo*). A ORM permite várias *restrições de anel* a serem aplicadas nesses pares de papéis. A UML não suporta *restrições de anel*, para isso, usa uma *restrição textual* para representá-las, utilizando linguagem natural. A figura 3.28 ilustra uma *restrição de anel (intransitiva)*, a qual determina se um professor p_1 for chefiado pelo professor p_2 e o professor p_2 for chefiado pelo professor p_3, então o professor p_1 não é chefiado pelo professor p_3. (capítulo 2)

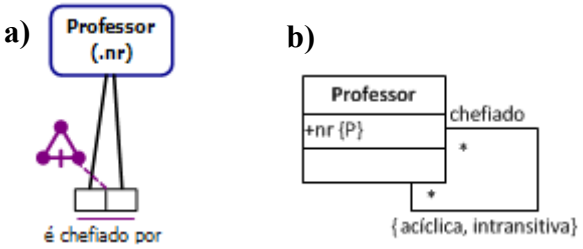


Figura 3. 28 - Restrição de anel (intransitivo) em ORM a) e restrições textuais em UML b)

Em UML, o termo *"agregação"* é usado para descrever a relação *todo/ parte* (descrita anteriormente). A ORM não tem uma notação especial para *agregação*. Essa situação, em ORM é modelada com recurso as *restrições de unicidade* e/ou de *restrições de obrigatoriedade*.

Esta secção apresentou os tipos de restrições mais usadas, contudo existem outras restrições em ORM que não foram estudadas em detalhe, nomeadamente: *restrição de união* (Halpin T., 2008).

3.7.4 Regras de derivação

A definição dos conceitos associados a regras de derivação foi introduzida no capítulo 2. Contudo, foi ilustrado que os *tipos de fato derivados* em ORM são marcados com um asterisco, e têm associado uma *regra de derivação* específica, expressa na linguagem textual da ORM (ver figura 3.29 a).

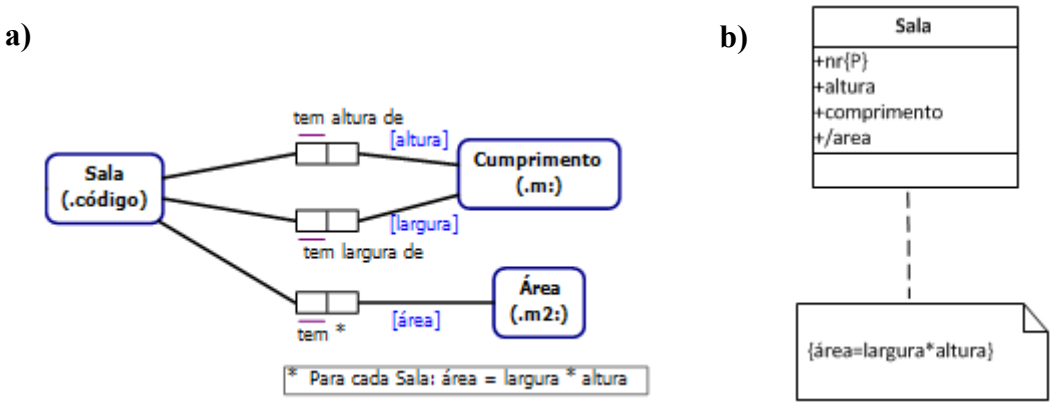


Figura 3. 29 - Derivação de área em ORM a) e UML b) (Halpin T. , 2008)

A *derivação de uma associação* em UML é representada graficamente por uma barra "/" antes do nome da mesma (Halpin T. , 2008). Entretanto, uma *regra de derivação* pode ser expressa em UML como uma *restrição*, conectada a uma associação, através

de ligação de dependência ou simplesmente descrita textualmente entre chavetas "{…}" como mostra a figura 3.29 b).

Todavia, a regra de derivação também pode ser modelada como uma restrição de união de subconjunto ou de igualdade (Halpin T. , 2008).

A **tabela 3.2** resume as correspondências principais entre a modelação conceptual de restrições em ORM e UML. Cada tipo de restrição foi descrito e exemplificado nos capítulos anteriores. O símbolo "—" indica que não existe correspondência predefinida entre os conceitos, e o símbolo "**§"** indica correspondência incompleta.

RESTRIÇÕES	
ORM	**UML**
Unicidade Interna	Multiplicidade de...*1 §*
Unicidade Externa	— *{usa associações qualificadas. §}*
Regra de Obrigatoriedade Simples	Multiplicidade de *1*...
Frequência: interna e externa	Multiplicidade *§;* —
Valor	Enumeração, e textual
Subconjunto e igualdade	Subconjunto *§*
Exclusão	Ou-restrições *§*
Subtipo e definição	Subclasse discriminada etc. *§*
Restrição de Anel	—
Cardinalidade de Objeto	Multiplicidade de Classe
— *{usa unicidade e obrigatoriedade. §}*	Agregação / composição
Restrições textuais	Restrições textuais

§ = Correspondência incompleta dos conceitos correspondentes

— = Sem correspondência dos conceitos

Tabela 3. 2 - Correspondência básica entre os conceitos conceptuais da ORM e UML para as restrições (Halpin & Bloesch,

Capítulo 4 – Abordagens de transformação de modelos

Este capítulo apresenta uma comparação sobre as abordagens de transformação propostas na Arquitetura dirigida por Modelos (MDA – *Model Driven Architecture*). Apresenta-se uma análise sucinta sobre cada uma das abordagens consideradas, uma descrição dos vários métodos de transformação de modelos, bem como as linguagens de modelação.

Este capítulo está estruturado da seguinte forma: a secção 4.1 apresenta uma introdução sobre as abordagens de transformação de modelos. A secção 4.2 descreve os principais objetivos do *Object Management Group* (OMG). A secção 4.3 apresenta os principais conceitos sobre o desenvolvimento de software baseado em modelos no domínio da MDA. As secções 4.4, 4.5 e 4.6 incidem nas transformações e mapeamentos de modelos. A secção 4.7 descreve a abordagem proposta a aplicar neste trabalho. Finalmente as secções 4.8 e 4.9, apresenta os metamodelos e o mapeamento ORM e UML.

4.1 Introdução

No início da computação, quando uma aplicação era executada numa única máquina, era comum encontrar sistemas monolíticos que apresentavam todas as funcionalidades da aplicação numa única e imensa camada. Neste contexto, a manutenção e atualização do *software* era extremamente penosa e complexa. Porém, a complexidade do desenvolvimento de aplicações levou à necessidade de novos tipos de arquiteturas, culminado com uma nova arquitetura de várias camadas. Com o objetivo de se manter diversas aplicações e um única ou várias base de dados, a arquitetura monolítica evoluiu para uma arquitetura de duas camadas e posteriormente para três camadas.

Como a maior parte dos sistemas atuais segue este modelo de arquitetura de três camadas. Neste capítulo vamos basearmo-nos nesse modelo para indicar como as abordagens ORM e a UML, aqui estudadas, se relacionam com o desenvolvimento de aplicações segundo esta arquitetura (ver figura 4.1).

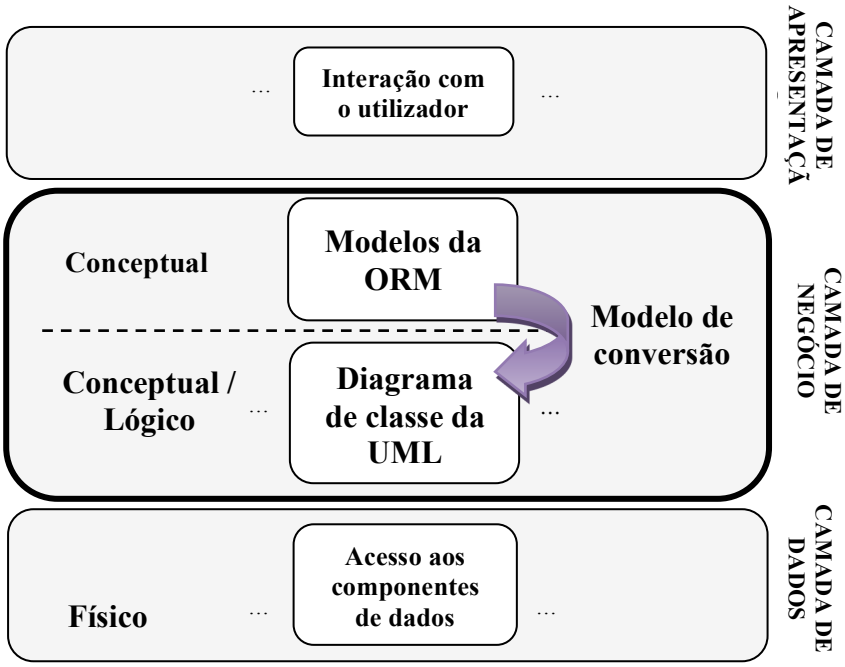


Figura 4. 1 - Arquitetura de três camadas.

A figura 4.1 ilustra uma arquitetura de três camadas, a qual mostra a separação das funcionalidades usando camadas, com o objetivo de separar a camada lógica da camada de apresentação, a camada lógica do negócio da conexão com a base de dados (lógica de acesso a dados). A separação em três camadas torna o sistema mais flexível, de modo que as partes podem ser alteradas independentemente. Nesta arquitetura, para cada camada existem modelos para diferentes níveis de abstração: modelo conceptual; modelo lógico; modelo físico e modelo externo.

Uma vez que os sistemas atuais requerem funcionalidades específicas para as diversas plataformas de execução, o desenvolvimento destas funcionalidades pode criar muitas vezes dependência entre estas funcionalidades e a plataforma para a qual o sistema é desenvolvido. Todavia, essa dependência deve ser minimizada para evitar que os sistemas fiquem obsoletos, permitindo assim, uma migração para uma versão mais recente, uma vez que as tecnologias existentes estão em constante evolução. Para que esta evolução tecnológica não resulte num elevado esforço de migração para as plataformas mais recentes, é necessária uma abordagem eficiente de apoio a estas transformações. Uma solução viável é a transformação de modelos, em que podemos migrar de um modelo para outro modelo. Nesse sentido, é necessária a introdução de uma abordagem baseada em modelos e definir as linguagens de modelação para especificação formal dos elementos desses modelos.

4.2 O *"Object Management Group"* (OMG)

O ***Object Management Group* (OMG)** (OMG, 2011) é uma organização internacional fundada em 1989, sem fins lucrativos, que padroniza o uso de especificações da indústria do software para aplicações com interoperabilidade empresarial. No contexto deste trabalho, o conjunto de padrões criado designa-se por ***Model Driven architecture* (MDA)** (OMG, Miller, & Mukerji, 2003). Em conformidade com estas especificações, a OMG possibilita o desenvolvimento de aplicações heterogéneas através da maioria das plataformas de *hardware* e sistemas operativos. Apresentamos algumas especificações definidas pela OMG (OMG, 2011):

- CORBA *(Common Object Request Broker Architecture)*;
- UML *(Unified Modeling Language);*
- MOF *(Meta Object Facility);*
- XMI *(XML Metadata Interchange)*;
- CWM *(Common Warehouse Metamodel);*
- MDA *(Model Driven Architecture).*

Na especificação do OMG para manipulação dos modelos, iremos focar-nos na UML (descrita no capítulo 3) e na MDA. Neste contexto, a próxima secção irá centrar-se na arquitetura dirigida por modelos, concretamente sobre os termos e conceitos, modelos e níveis de abstração associados.

4.3 Model Driven Architecture (MDA)

A ***Model Driven architecture* (MDA)** (OMG, Miller, & Mukerji, 2003) é uma abordagem de desenvolvimento de sistemas baseada em modelos criada pelo OMG, que tem como principal foco a separação entre a especificação das funcionalidades e a especificação da implementação para uma plataforma específica.

A transformação de modelos é uma das propostas fundamentais da MDA, e consiste na transformação de um modelo independente da plataforma (*Plataform-Independent Model*-PIM) para um modelo específico da plataforma (*Plataform-Specific Model*-PSM) e na geração automática de código. No entanto, para compreender esta abordagem, é necessário definir e compreender alguns termos e conceitos no contexto da MDA, os quais são descritos de seguida.

4.3.1 Terminologia e conceitos da MDA

Existem vários termos e conceitos associados ao MDA. A figura 4.2 ilustra um resumo dos conceitos e a relação entre eles (Silva & Vieira, 2001).

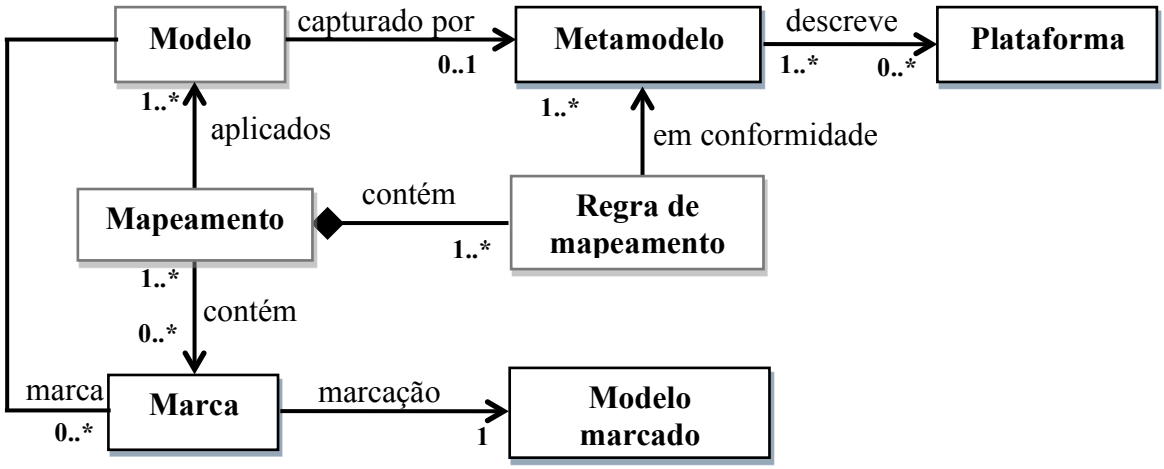


Figura 4. 2 - Termos e conceitos da MDA (Silva & Vieira, 2001)

Conforme apresentado na figura 4.2, partimos de uma realidade e construímos o **modelo**, o qual consiste num conjunto de elementos que descrevem tal realidade, seja ela física ou abstrata. Geralmente, um modelo é uma combinação de desenhos e textos explicativos. O texto pode ser representado por uma linguagem formal ou em linguagem natural. O modelo pode ser capturado por recurso a uma linguagem de modelação específica para o domínio da aplicação, ou seja, o **metamodelo.** O metamodelo define a estrutura, semântica, e restrições de modelos que partilham a mesma sintaxe. Por exemplo, um modelo que utiliza a UML é construído com base no metamodelo UML, o qual descreve como os modelos UML podem ser estruturados, os elementos que podem conter, e as propriedades dos elementos que eles apresentam. O metamodelo pode especificar as características de uma **plataforma**, seja ela dependente ou independente. A **plataforma** pode ser descrita como sendo a especificação de um ambiente de execução de um conjunto de modelos.

Sobre um ou mais modelos, podem ser aplicados **mapeamentos**, que contém um ou mais modelos como entrada (*fonte*) e produz um modelo de saída (*alvo*). Um mapeamento contém **regras de mapeamento.** Estas regras são descritas ao nível do metamodelo, de tal forma que serão aplicadas a todos os conjuntos de modelos *fonte* que estejam em conformidade com o metamodelo dado. Em alguns casos podem

coexistir várias regras de mapeamento, nessa situação é necessário um mapeamento adicional de entrada para selecionar a regra a ser aplicada, o que se designa por **marca**. A **marca** é um sinal aplicado nos elementos do modelo e contêm informação necessária para a transformação do modelo. O processo utilizado para guiar a transformação de modelos usando marcas e designado por **marcação**. As marcas representam um conceito ao nível do PSM que deve ser aplicado para um determinado elemento ao nível do PIM. Esta marca indica como este elemento será transformado, ou seja, é uma forma possível de diferenciar arquiteturas alvo. Estas marcas podem consistir em estereótipos definidos por um "perfil UML"[1]; valores etiquetados (*tagged value*); expressões em *Object Constraint Language* (OCL) (Rumbaugh, Jacobson, & Booch, 1999) para a especificação de restrições na modelação dos diagramas UML e comentário (*anchor text*).

4.3.2 Modelos e níveis de abstração em MDA

A MDA propôs a criação de modelos em diferentes níveis de abstração. Estes níveis separam a implementação de uma arquitetura específica do modelo conceptual de uma aplicação. A arquitetura de camadas, ilustrada na figura 4.3 é baseada na arquitetura clássica de quatro camadas do OMG. Segundo esta organização, as camadas distingue-se pelos níveis de generalidade e de abstração dos seus elementos constituintes, sendo denominadas por: M0, M1, M2 e M3.

Figura 4. 3 - Arquitetura de quatro camadas (Silva & Videira, 2001)

Este tipo de arquitetura mostra a separação dos vários níveis, e não só, trata também a complexidade inerentes às várias linguagens. Existe um forte relacionamento entre as camadas, as mais abstratas oferecem uma infraestrutura para as mais concretas.

[1] Um ***perfil UML*** é considerado um mecanismo de extensão da UML. O *perfil UML* é aplicado à especificação da linguagem UML e descreve uma nova linguagem de modelação. O perfil pode adicionar novas funcionalidades ou restringir.

O nível mais baixo da arquitetura, o nível M0 pode ser visto como o padrão representativo da realidade. Os níveis M1, M2, e M3 especificam respetivamente modelos, metamodelos e meta-metamodelos.

A camada **objetos (reais) do utilizador** (**nível M0**) corresponde à camada de gestão e instanciação dos dados/objetos. Os objetos e suas interações representam os componentes básicos de qualquer sistema. Estas entidades são descritas no modelo, mas representam a informação propriamente dita, a instanciação na aplicação.

A camada **modelo** (**nível M1**) é compreendida pelos *metadados*[2] que descrevem dados na camada objetos. O modelo expressa entidades presentes no domínio semântico da aplicação (abstração do sistema). Nesta camada, define-se as regras do sistema.

A camada do **metamodelo** (**nível M2**) descreve a estrutura sintática e semântica do *metadado*. Um metamodelo é também um modelo que define a regras de criação de modelos de uma determinada linguagem. Um metamodelo é uma instância de um meta-metamodelo (nível M3).

A camada do **meta-metamodelo** (**nível M3**) compreende a descrição da estrutura e da semântica dos *meta-metadados*. O meta-metamodelo foi proposto pelo OMG para lidar com a complexidade de UML. Ele estabelece, em particular, um alicerce sobre as possíveis extensões da UML e de novos metamodelos construídos. O *Meta Object Facility (*MOF) providência a especificação básica para UML. Por exemplo, todos os construtores de entidades de UML são agrupados em uma única especificação de MOF, a *metaclass*[3].

A MDA especifica três modelos padrão de um sistema nomeadamente, ***Computation Independent Model (CIM)***; ***Plataform-Independent Model (PIM)*** e ***Plataform-Specific Model (PSM).*** O ***Computation Independent Model (CIM)*** pode ser descrito como o mecanismo de análise da concepção de um sistema, do ponto de vista do ambiente e dos requisitos do sistema. O CIM, também chamado de **modelo de domínio** ou **modelo de negócio,** não mostra detalhes da estrutura do sistema, apenas modela o sistema considerando as regras de negócios. Normalmente, o modelo é independente da implementação do sistema, sendo bastante útil para adotar e definir um vocabulário

[2] O conceito ***metadado*** é definida como uma abstração de um dado, ou seja, *"dados sobre outros dados"*. Tem a característica de gestão dos dados de um sistema para fornecer um suporte á documentação, reusabilidade e interoperabilidade.

[3] O conceito ***metaclass***, orientado a objetos, é uma classe cujas instâncias também são classes e não objetos. As metaclasses definem o comportamento de certas classes e suas instâncias.

entre modelos (OMG, Miller, & Mukerji, 2003). O trabalho proposto nesta dissertação não aborda o modelo CIM, tendo como ponto inicial o ***Plataform-Independent Model (PIM).***

O ***Plataform-Independent Model (PIM)*** descreve o sistema a implementar, sem especificar os detalhes sobre a plataforma onde a aplicação será implementada (OMG, Miller, & Mukerji, 2003). Este modelo define o sistema de acordo com as regras de negócio, ignorando os aspetos referentes à plataforma de implementação.

O ***Plataform-Specific Model (PSM)*** é o resultado da transformação de modelo PIM de um sistema, incluindo informação sobre a tecnologia usada na execução numa plataforma específica (OMG, Miller, & Mukerji, 2003). O modelo PIM pode gerar um ou mais modelos PSMs, de acordo com as plataformas disponíveis.

Após a descrição dos modelos PIM e PSM, a próxima secção descreve sumariamente as abordagens que permitem a transformação automática entre estes tipos de modelos,

4.4 Abordagens de transformações de modelos

Esta secção apresenta as abordagens que são usados para a transformação de modelos (OMG, Miller, & Mukerji, 2003).

1. **Transformação de modelos usando marcas;**
2. **Transformação de modelos usando metamodelos;**
3. **Transformação de modelos usando modelos;**
4. **Transformação de modelos usando padrões.**

4.4.1 Transformação de modelos usando marcas

A **transformação de modelos usando marcas** é descrita como a transformação de um modelo PIM marcado em modelos específicos de diferentes plataformas através da aplicação de regras de mapeamento. Este mapeamento é a especificação da transformação, incluindo regras e outra informação, de forma a transformar um modelo PIM marcado e produzir um modelo PSM para uma plataforma específica (OMG, Miller, & Mukerji, 2003). A figura 4.4 ilustra o processo de transformação de modelos usando marcas, de acordo com a MDA.

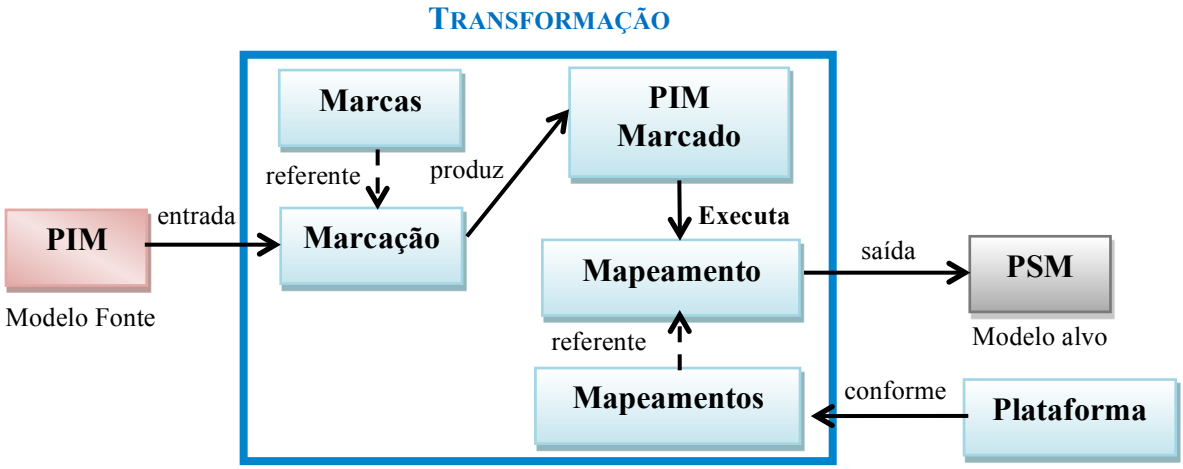


Figura 4. 4 - Transformação de modelos usando marcas (OMG, Miller, & Mukerji, 2003)

O processo de **transformação** pode ser entendido como uma função aplicada sobre um conjunto de conceitos, resultando num novo estado (modificado) relativamente ao estado original. Neste processo estão envolvidos vários conceitos: as **marcas** que produzem um **PIM marcado** através do processo de **marcação**, o **mapeamento** que contém as regras e informação adicional para transformar um modelo PIM marcado, produzindo um modelo PSM para uma plataforma específica. As marcas são uma forma possível de diferenciar arquiteturas alvo, ou seja, indica como um dado elemento será transformado para uma determinada tecnologia. Consistem na especificação do mecanismo para transformar os elementos de um modelo, em conformidade com as especificações das regras, em elementos de um outro modelo.

4.4.2 Transformação de modelos usando metamodelos

A **transformação de modelos usando metamodelos** especifica como, um modelo (PIM), que obedece a um metamodelo A, se converte num modelo PSM que obedece ao metamodelo B. O mapeamento segue as regras de transformação, as quais descrevem como elementos do metamodelo independente da plataforma se transformam em elementos do metamodelo específico para uma dada plataforma. A figura 4.5 ilustra o processo de transformação de modelos em MDA usando metamodelos.

Figura 4. 5 Transformação de modelos usando metamodelos (OMG, Miller, & Mukerji, 2003)

4.4.3 Transformação de modelos usando modelos

A **transformação de modelos usando modelos** declara que os elementos do PIM são subtipos desses tipos independentes da plataforma, e os elementos do PSM são subtipos dos tipos específicos da plataforma. (ver figura 4.6).

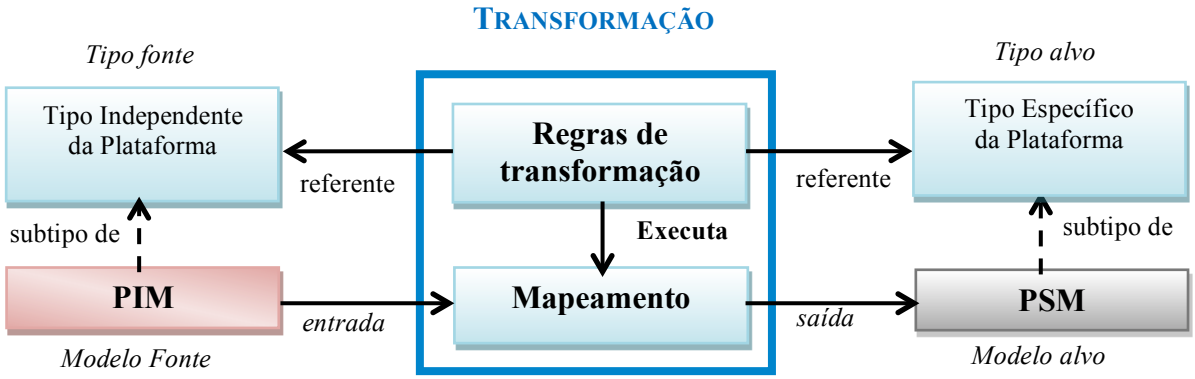


Figura 4. 6 - Transformação de modelos usando modelos (OMG, Miller, & Mukerji, 2003)**.**

Esta transformação é apresentada com base no mapeamento de tipos independentes da plataforma para os tipos específicos da plataforma. Os elementos no PIM são subtipos dos elementos especificados no modelo independente da plataforma. Os elementos no PSM são subtipos dos elementos especificados no modelo dependente plataforma.

4.4.4 Transformação de modelos usando padrões

A **transformação de modelos usando padrões** pode ser considerada como uma extensão das abordagens de mapeamento de modelos (tipos) e metamodelo, que inclui

padrões, bem como os tipos ou conceitos de linguagem de modelação. Este tipo de transformação é apresentado na figura 4.7.

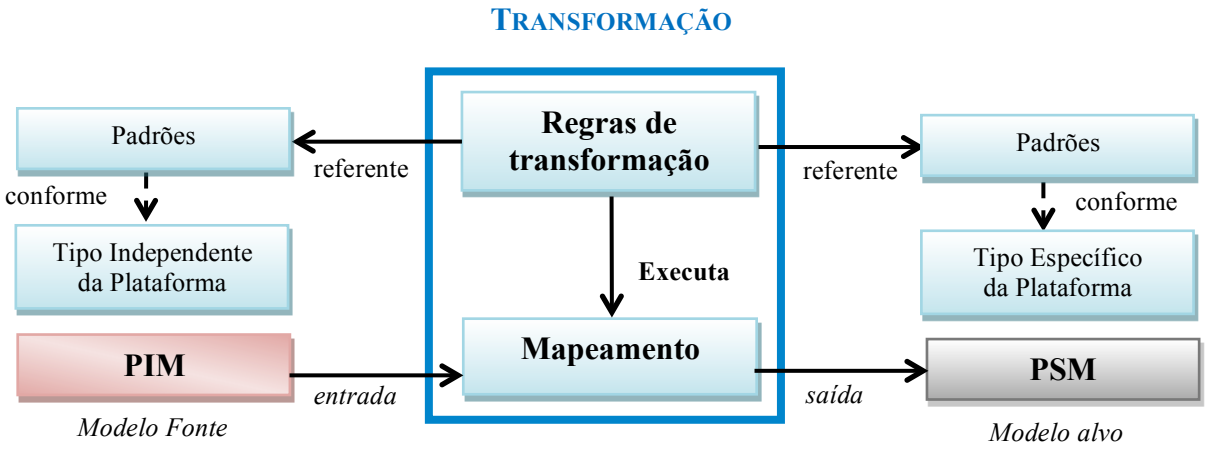


Figura 4. 7 - Transformação de modelos usando padrões (OMG, Miller, & Mukerji, 2003)

Esta abordagem de transformação especifica aproximações ou padrões em conjunto com os tipos ou conceitos do modelo. Emprega modelos genéricos de marcas e mapeamentos de um PIM para um PSM. Além das abordagens de transformação de modelos, apresentadas anteriormente, existem outros métodos e abordagem de transformação de modelos que aqui não foram citados (OMG, Miller, & Mukerji, 2003). A definição, e os métodos utilizados para fazer a transformação usando as abordagens de transformação de modelos supracitadas serão descritas a seguir.

4.5 Transformações de modelos MDA

De acordo com a definição de transformação de modelos proposta pela MDA versão 1.0.1 (OMG, Miller, & Mukerji, 2003):

> *"A transformação de modelo é o processo de conversão de um modelo para outro modelo do mesmo sistema".*

A transformação de modelos constitui um mecanismo fundamental da MDA, através do qual um modelo independente de plataforma (PIM) é transformado num modelo específico de uma determinada plataforma (PSM). A figura 4.8 ilustra a transformação de modelos na abordagem da MDA, que se inicia no modelo PIM de modo a produzir

um ou mais modelos PSM, bem como algumas informação adicionais a serem incluídas durante a transformação.

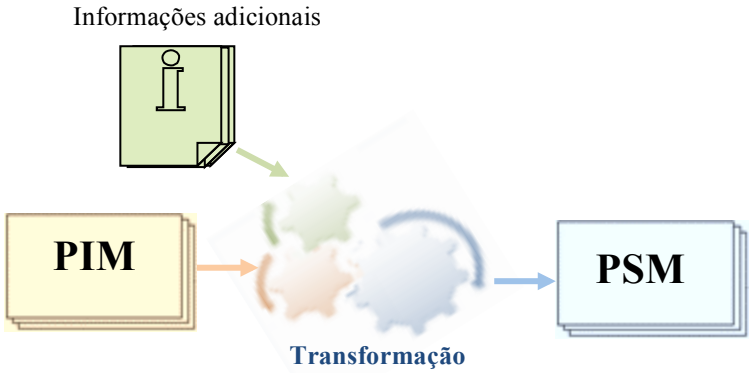


Figura 4. 8 - Transformação de modelo PIM para PSM

Existem várias ferramentas de suporte a transformação de modelo (OMG, Miller, & Mukerji, 2003) s. As transformações podem utilizar diferentes métodos, nomeadamente: **transformação manual, transformação de um PIM que segue um perfil**, **transformação usando padrões e marcações** e **transformação automática**. Assim, uma transformação de PIM para PSM pode ser realizada manualmente, através da aplicação de *perfil UML* (extensões UML), de padrões *(patterns)*, marcas *(markings)* ou automaticamente, como se descreve de seguida.

4.5.1 Transformação manual

Este método consiste em descrever manualmente as ideias, decisões de conceição de uma transformação de modelos PIM para PSM durante o processo de desenvolvimento de um *software* que está em conformidade com os requisitos previamente definidos. A contribuição da MDA, neste caso, vem da distinção explícita entre um modelo independente da plataforma (PIM) e o modelo específico da plataforma (PSM) (OMG, Miller, & Mukerji, 2003).

4.5.2 Transformação de PIM que segue um perfil

Este método consiste em manipular um modelo PIM com marcações utilizando um perfil UML independente de plataforma. O PIM inicial *é* transformado para um PSM expresso por um segundo perfil UML, sendo este específico para determinada plataforma (OMG, Miller, & Mukerji, 2003).

4.5.3 Transformação usando padrões e marcações

Este método consiste na utilização de padrões para definir mapeamentos, as marcações correspondem a elementos desses padrões (OMG, Miller, & Mukerji, 2003). Este método pode ser aplicado de duas formas distintas:

- Os elementos de um PIM são marcados e transformados de acordo com o padrão estabelecido no mapeamento, produzindo um PSM.
- Recorrendo a regras que especificam que todos os elementos no PIM sigam um determinado padrão, os quais, serão transformados em instâncias de um outro padrão, no PSM.

4.5.4 Transformação automática

Este método consiste na transformação direta de um PIM para PSM sem a necessidade de adicionar informação ao PIM de origem. O PIM de origem contém toda a informação necessária para a transformação no modelo alvo. O PIM é completo em relação à sua classificação, estrutura, invariantes, bem como, pré e pós-condições. Assim, é possível especificar o comportamento diretamente no modelo de origem utilizando uma linguagem de transformação que seguem um algoritmo.

Para a MDA, uma característica fundamental do método de transformação de modelos é a noção de mapeamento. Segundo a arquitetura MDA, um mapeamento é um conjunto de regras utilizadas para modificar, refinar ou transformar um modelo para outro (MDA , 2001).

A figura 4.9 apresenta uma descrição do Metamodelo MDA.

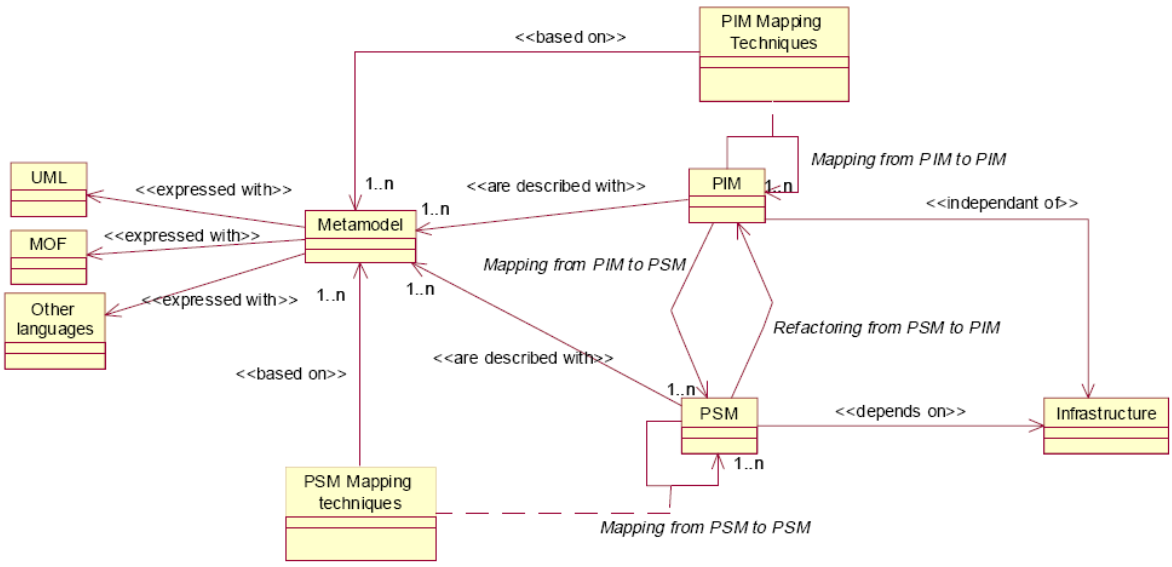


Figura 4. 9 - Metamodelo da MDA (MDA , 2001, p. 12)

Na figura pode-se constatar que os modelos PIM, modelos PSM e técnicas de mapeamento são descritos com base no metamodelo, expresso preferencialmente com as especificações formais da OMG como UML, MOF ou CWM (MDA , 2001). As técnicas de mapeamento aplicadas nestas transformações são:

1. **PIM – PIM** – São transformações realizadas sobre os modelos independentes de plataforma, não aplicando os conceitos relacionados com a plataforma. Por exemplo, transformação de um modelo conceptual para um modelo lógico, ao nível de duas linguagens de modelação diferentes associadas a abordagens de análise e desenho de sistemas (ORM, UML). Classifica-se nesta categoria o tema deste trabalho.
2. **PIM – PSM** – São transformações nas quais são aplicados os conceitos relacionados com a plataforma, sobre um modelo independente, dando origem a um ou mais modelos específicos da plataforma ou tecnologia. Por exemplo, a transformação de um modelo de classes UML para um programa *Java* (OMG, Miller, & Mukerji, 2003).
3. **PSM – PSM** – São transformações realizadas sobre os modelos específicos da plataforma no sentido de refinar ou melhorar o modelo PSM. Por exemplo, aplicação de padrões de projeto específicos para uma plataforma *J2EE* (OMG, Miller, & Mukerji, 2003).

As transformações de modelos podem agregar ou não características e elementos constituintes de uma plataforma específica. Esta transformação pode ser realizada no sentido direto **(PIM-PSM)** no sentido inverso **(PSM-PIM)** ou refinamento **(PSM-PSM)**, conforme mostra a figura 4.10.

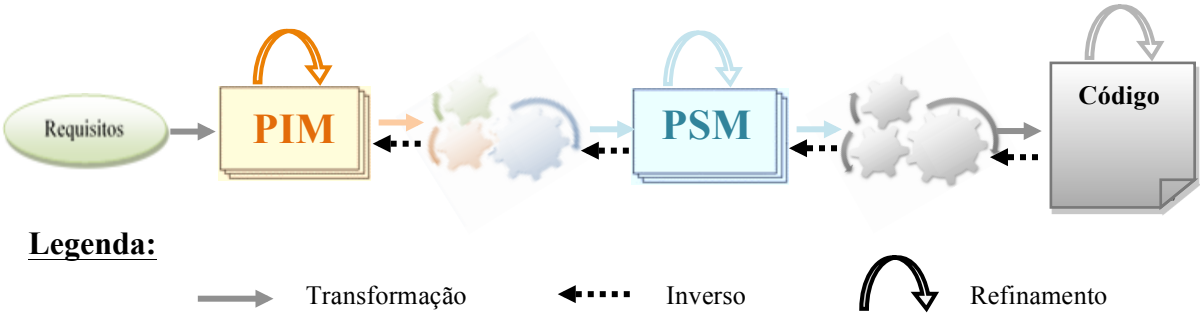


Figura 4. 10 - Transformação de Modelos da MDA

4.6 Mapeamentos MDA

Um mapeamento MDA (OMG, Miller, & Mukerji, 2003) fornece especificações para a transformação de um PIM a um PSM para uma plataforma específica. O modelo de plataforma vai determinar a natureza do mapeamento.

A MDA define os seguintes tipos de mapeamentos (OMG, Miller, & Mukerji, 2003):

1. **Mapeamento de tipo de modelo**
2. **Mapeamento de instâncias do modelo**
3. **Outros**

4.6.1 Mapeamento de tipo de modelo

O **mapeamento de tipo de modelo** descreve um mapeamento a partir de qualquer modelo construído usando tipos especificados na linguagem PIM para modelos expressos através de tipos de uma linguagem PSM (OMG, Miller, & Mukerji, 2003).

O PIM é descrito usando uma linguagem de modelação independente da plataforma, por exemplo, UML, ORM, etc. O **mapeamento de metamodelo** é um exemplo específico de um mapeamento do tipo de modelo, onde os tipos de elementos no modelo PIM e no modelo PSM são especificados como metamodelos MOF. Enquadra-se aqui o tipo de mapeamento proposto neste trabalho.

4.6.2 Mapeamento de instâncias do modelo

O segundo tipo, conhecido por **mapeamento de instâncias do modelo**, é descrito através de marcações que indicam como a transformação deve ser feita. Este tipo de mapeamento difere do anterior, na transformação de instâncias de um mesmo tipo, ou seja, as instâncias podem ter mapeamentos distintos, de acordo com as marcas (OMG, Miller, & Mukerji, 2003).

Outra abordagem ao mapeamento de modelos é a identificação de elementos do modelo no PIM, que deve ser transformados de modo particular, dada a escolha de uma plataforma específica para o PSM. Pode-se destacar as **"marcas"** neste tipo de mapeamento.

O mapeamento de instância do modelo vai definir marcas. Uma marca representa um conceito no PSM, e é aplicado a um elemento do PIM, para indicar como é que o elemento é transformado (OMG, Miller, & Mukerji, 2003).

4.6.3 Outros

Além das técnicas de mapeamento mencionados anteriormente, existem outros tipos de mapeamentos que não foram analisados neste trabalho, nomeadamente, combinação de mapeamento de tipo e de instância de modelo; moldes *(templates)*; modelos marcados; linguagem de mapeamento (OMG, Miller, & Mukerji, 2003).

4.7 A Abordagem Proposta

Entre as abordagens de transformação de modelos definidas na MDA para suportar o desenvolvimento de sistemas, considerou-se que as **abordagens de transformação usando marcas, modelo e padrões**, estão mais focadas em modelos específicos da arquitetura e geração de código. As marcas representam um conceito no PSM, que são aplicadas a elementos de um PIM para indicar como estes elementos são transformados numa plataforma específica.

As transformações de modelos com recurso a padrões empregam modelos genéricos de marcas e mapeamentos de um PIM para um PSM.

As marcas, sendo específicas para plataformas, não fazem parte do modelo independente de plataforma. Assim, a abordagem de transformação com recurso a marca não foi considerada.

Neste contexto, optou-se por aplicar a abordagem de **transformação de modelos usando metamodelos**, uma vez que está de acordo com os objetivos deste trabalho, o qual consiste na transformação de modelos ao nível de linguagens de modelação distintas, ORM e UML.

Para a especificação da transformação, o **mapeamento de metamodelo** é o tipo de mapeamento mais adequado, considerando que pretendemos trabalhar com os metamodelos das linguagens ORM e da UML. O mapeamento de metamodelos entre os modelos ORM para UML serão especificados em **ECORE**, no padrão *EMOF* (Jouault F. , 2009). Assim, o mapeamento descreve as regras de mapeamento e/ou algoritmos de acordo com os tipos de instâncias no metamodelo *EMOF* da linguagem de modelação do PIM de origem (ORM), na qual resulta na geração de instâncias de tipos no metamodelo *EMOF* na linguagem de modelação do PIM de destino (UML).

4.8 Metamodelos ORM e UML

Os conceitos da ORM e da UML foram anteriormente definidos neste trabalho. Apresentou-se uma visão geral dos conceitos da ORM no capítulo 2, e descrição dos conceitos associados a UML, no capítulo 3.

Com o objetivo de destacar os conceitos comuns para a conversão entre os dois modelos, esta secção inclui a descrição dos dois metamodelos. Cada metamodelo visa especificar a gramática de um modelo sintaticamente válido para ORM bem como para a UML. Estes metamodelos serão detalhados nas subsecções seguintes.

4.8.1 Metamodelo ORM

Devido a riqueza de conceitos da ORM, o metamodelo ORM proposto neste trabalho será organizado em três partes: a primeira descreve os *tipos principais,* a segunda apresenta os *relacionamentos* e, finalmente, a terceira detalha as *restrições*.

Para uniformizar a comparação dos metamodelos, o metamodelo ORM (o desenho completo pode ser consultado no Anexo D), foi representado utilizando a mesma metalinguagem, ou seja, a UML.

De acordo com a terminologia apresentada no capítulo 2, a figura 4.11 apresenta o metamodelo dos principais tipos da ORM, nomeadamente o *Tipo de Objetos*

(*ObjectType*) que combina os conceitos de *Tipo de Entidade* (*EntityType*), e *Tipo de Valor* (*ValueType*).

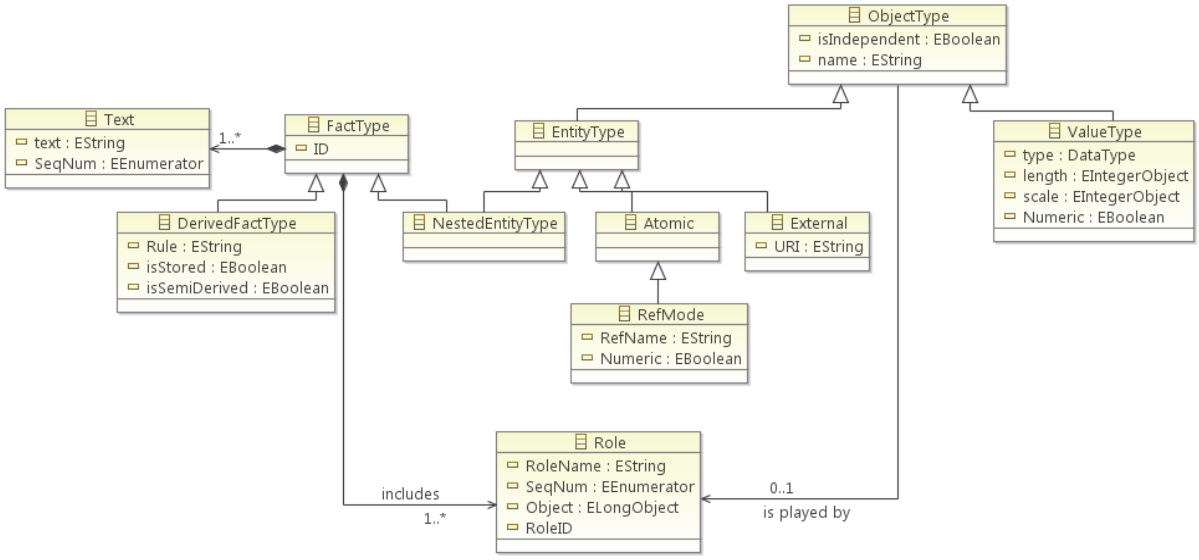


Figura 4. 11 - Metamodelo dos principais tipos da ORM (Halpin & Cuyler, 2003)

A ORM suporta com extrema flexibilidade a leitura de associação (Halpin & Cuyler, 2003). Em geral, um predicado *n-ário* tem *n!* formas de leitura dos seus papéis e para cada uma destas formas, uma ou mais leituras podem ser apresentadas no diagrama. Estas formas de leituras diretas e inversas foram descritas anteriormente no capítulo 2. Opcionalmente, a qualquer *papel* pode ser atribuído um nome. Para qualquer *tipo de objeto*, os nomes dos seus *papéis* devem ser únicos para evitar ambiguidade na leitura destes.

De acordo com os conceitos sobre relacionamentos ORM, apresentados no capítulo 2, a figura 4.12 mostra o metamodelo base de relacionamentos ORM com os seus elementos: **papel** *(Role),* **predicado** (*Predicate*) e **objetivação** (*Objectification*) (Halpin & Cuyler, 2003) (Halpin T. , 2008).

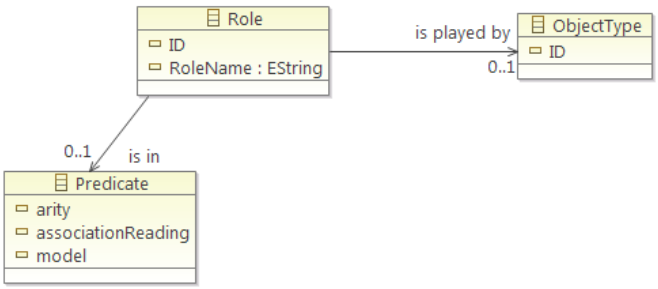


Figura 4. 12 - Metamodelo de relacionamento ORM (Halpin & Cuyler, 2003)

Assim, um *papel* é parte de uma associação em ORM. A *aridade* de uma associação é o seu número de *papéis*. Uma associação é composta de um *predicado* lógico e dos tipos de objetos que desempenham os *papéis*. Os *predicados* são exibidos em sequência do *papel*, e têm uma ou mais leituras, dependendo da ordem em que os *papéis* são interpretados. O tipo de relacionamento, *objectivação* (Halpin T. , 2008) não foi muito aprofundado neste trabalho, uma vez que só se aplica quando se pretende especificar mais informações sobre o relacionamento.

Finalmente, a figura 4.13 ilustra o metamodelo simplificado para as *restrições (Constraint)* ORM. A descrição e exemplos de cada uma das restrições apresentadas foram descritas anteriormente no capítulo 2.

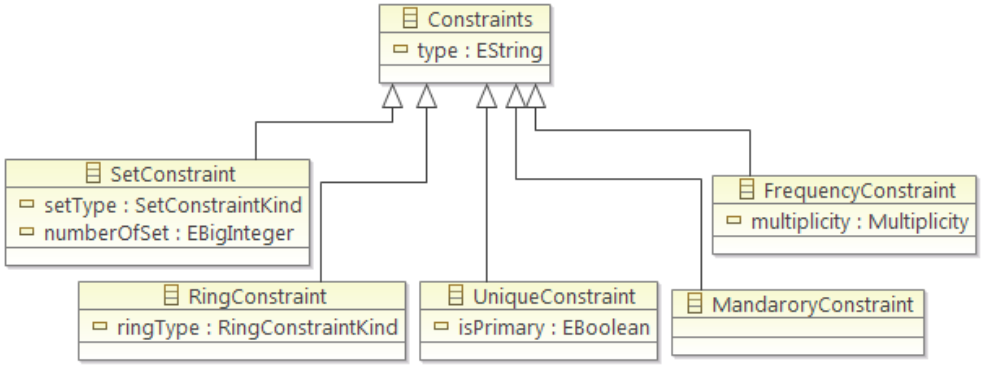


Figura 4. 13 - Metamodelo de restrições ORM (Halpin & Cuyler, 2003)

Este metamodelo das restrições visa mostrar não só as diferentes restrições em ORM, mas também o relacionamento entre elas. Para efeitos de implementação é possível combinar alguns destes tipos de restrições. Podem eventualmente serem adicionados outros tipos de restrições (Halpin T. , 2008). A ORM tem um vasto leque de restrições, algumas das quais foram omissas (Halpin T. , 2008), uma vez que não foram analisadas neste trabalho.

4.8.2 Metamodelo UML

Devido à sua extensão e complexidade, o metamodelo UML descrito nesta secção é uma versão simplificada. Contudo, a sua especificação completa pode ser encontrada em (ORM, 2004).

O metamodelo UML é definido através de um diagrama de classes da UML. De acordo com a terminologia apresentada no capítulo 3, a figura 4.14 apresenta o metamodelo simplificado (Anneke G. Kleppe, 2003) dos principais tipos da UML, nomeadamente o

elemento básico (*ModelElement*) que combina os conceitos de **classe** *(Class)* e **objetos** *(Objects)* e a interação entre os objetos, os **relacionamentos** *(relationship)*.

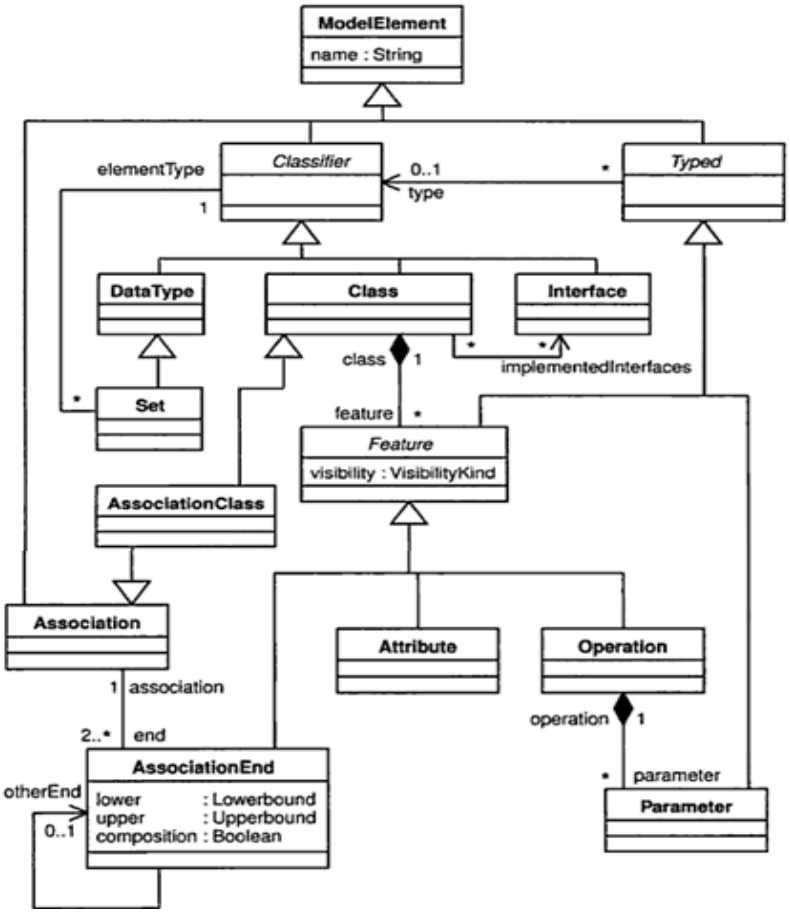


Figura 4. 14 - Metamodelo simplificado da UML (Anneke G. Kleppe, 2003)

As restrições *(constraint)* constituem um mecanismo de extensão em UML. As restrições não são um mecanismo de extensão padrão, dado que não permitem modificar um metamodelos existentes, em vez disso, possibilitam a adaptação de um metamodelo existente adicionando construções específicas de um domínio, plataforma ou método em particular. Além das restrições a UML providencia outros mecanismos que permitem estendê-lo de forma consistente: marcas e estereótipos *(stereotype)*. Estes mecanismos são aplicados aos elementos do modelo. Representam, portanto, extensões à própria linguagem que permitem alterar a estrutura e semântica dos modelos.

A figura 4.15 ilustra a sintaxe abstrata dos mecanismos de extensão do UML. Note-se a definição e relação entre as **metaclasses** *(metaclass),* **estereótipos** *(stereotype)*, **restrições** *(constraint)* e **valores etiquetados** *(taggedValue)* (Silva & Videira, 2001).

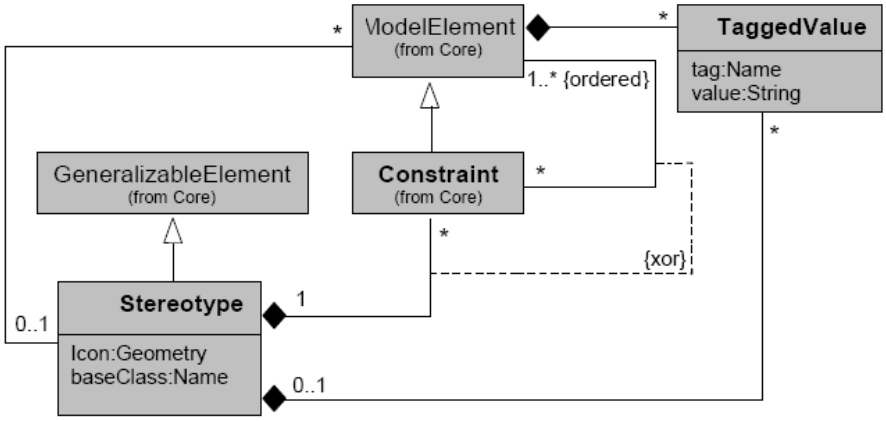


Figura 4. 15 Mecanismos de extensão da UML (Silva & Videira, 2001)

O conceito de restrição consiste na especificação da semântica associada a um mais elemento do modelo. Essa especificação é escrita numa determinada linguagem de restrições. Por exemplo: OCL, linguagem de programação, notação matemática ou linguagem natural (Silva & Vieira, 2001). É pelo fato da escolha da linguagem ser arbitrária, que as restrições são um mecanismo de extensão.

No metamodelo uma restrição *(Constraint)* associada diretamente a um *ModelElement* descreve as restrições semânticas que esse *ModelElement* tem de satisfazer. Por outro lado, restrições associadas a um *Stereotype* aplicam-se a cada *ModelElement* ligados ao *Stereotype* (Silva & Videira, 2001).

4.9 Mapeamento ORM e UML

Nesta seção, são apresentados os elementos da ORM e como ocorre manifestação desses elementos em UML.

Com base nas **tabelas 3.1** e **3.2** (ver capítulo 3) são apresentadas as correspondências básicas de mapeamento para a modelação de dados e de restrições entre esquemas conceptuais ORM e diagrama de classes UML.

As regras de transformação que constituem a definição de transformação estão resumidas na seguinte tabela 4.1.

Nome da regra	Tipo de elemento de entrada no metamodelo ORM	Tipo de elemento de saída no metamodelo UML
EntityType2Class	**Tipo de entidade** *(EntityType)*	**Classe** *(Class)*
ValueType2Attribute	**Tipo de valor** *(ValueType)*	**Atributo** *(Attribute)*
UnaryRel2Attribute	**Tipo de Relacionamento Unário** *(Unary Relashionship Type)*	**Atributo *booleano*** *(Boolean Attribute)*
RefMode2Attribute	**Esquema de Referência** *(Reference Model)*	**Atributo** *(Attribute)*
NaryRel2Association	**Tipo de Relacionamento n-anário** *(n-ary Relashionship Type)*	**Associação** ***(Association)***
Subtype2SubClass	**Subtipo** *(Subtype)*	**Subclasse** *(Subclass)*
IUniq2Mult	**Unicidade Interna** *(Internal Uniqueness)*	**Multiplicidade** *(Multiplicity of ..1)*
MandRole2Mult	**Regra de Obrigatoriedade** *(Mandatory Role)*	**Multiplicidade** *(Multiplicity of 1..)*
Freq2Mult	**Frequência** *(Frequency)*	**Multiplicidade** *(Multiplicity)*
Subset2Subset	**Subconjunto** *(Subset)*	**Subconjunto** *(Subset)*
Value2Enum	**Valor** *(Value)*	**Enumerado** *(Enumeration)*

Tabela 4. 1 - Regras de mapeamento

Capitulo 5 – Metodologias de Transformação

Este capítulo apresenta a descrição sucinta das regras de transformação, bem como do processo de transformação de modelos ORM para modelos UML. Segue-se uma breve descrição das suas secções. A secção 5.1 apresenta a visão geral do processo de transformação, as regras de transformação, a linguagem de transformação e os metamodelos. A secção 5.2 apresenta o processo de transformação proposto. Para concluir, a secção 5.3, descreve o ambiente de desenvolvimento, onde se apresenta a ferramenta de transformação, a criação do projeto, execução e resultados obtidos.

5.1 O Processo de transformação

O processo de transformação é baseado num mecanismo que envolve uma sequência de procedimentos de transformação de um modelo fonte para um modelo alvo.

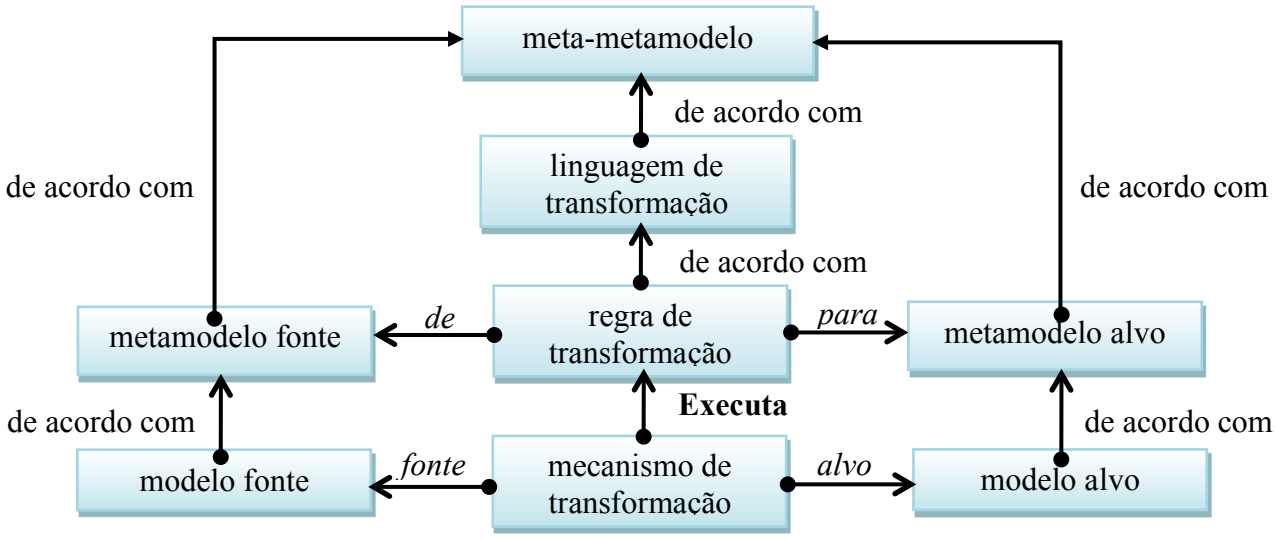


Figura 5. 1 - Processo de Transformação de modelos (ATLAS, 2006)

A figura 5.1 descreve genericamente o processo de transformação de modelos. Como se pode constatar, o *modelo fonte* em conformidade com o seu *metamodelo fonte* é transformado para o *modelo alvo* que está de acordo com o *metamodelo alvo*.

O *mecanismo de transformação* de um *modelo fonte* para um *modelo alvo* executa as *regras de transformação* de acordo com a *linguagem de transformação* estando esta linguagem em conformidade com um *meta-metamodelo*.

De seguida apresento a descrição dos conceitos relacionados com o processo de transformação, nomeadamente as regras de transformação a linguagem de transformação e os metamodelos.

5.1.1 Regras de transformação

A geração de elementos do modelo alvo é conseguida através da especificação das regras de transformação. Assim, segue-se uma breve descrição das regras de transformação entre as linguagens ORM e UML, baseadas na tabela de mapeamentos entre os conceitos da ORM e UML apresentada no capítulo 4.

Regra 1: *EntityType2Class*

Esta regra recebe um **Tipo de entidade** *(EntityType)* no modelo ORM de origem (este elemento é um subelemento de **tipo de objeto** *(ObjectType)* da ORM). A regra converte o *EntityType* para uma *Class* no modelo UML de destino.

Regra 2: *ValueType2Attribute*

Esta regra converte cada **tipo de valor** *(ValueType)* em um **atributo** *(Attribute)*. Começa-se com a classe criada na regra 1 *(EntityType2Class),* depois a **regra 2** converte os *ValueType* para *Attribute* da classe criada.

Regra 3: *UnaryRel2Attribute*

Se um **tipo de factos** *(FactType)* desempenha um único **papel** *(unary),*esta regra converte *FactType* para um atributo booleano *(Boolean Attribute).*

Regra 4: *RefMode2Attribute*

Se um **Tipo de entidade** *(EntityType)* tem um **esquema de referência** (*Reference Model),* esta regra converte para um atributo primário *(Attribute {P})* da sua respectiva classe UML **(regra 1).**

Regra 5: *NaryRel2Association*

Se um **tipo de factos** *(FactType)* desempenha mais de dois **papéis** *(n> 2)* esta regra converte o *FactType* de entrada do modelo ORM para uma associação de classe da UML *(class association)*.

Regra 6: *Subtype2Subclass*

Se um **Tipo de entidade** *(EntityType)* ORM é um **subtipo** *(Subtype)* ou **supertipo** *(Supertype)* esta regra converte para uma subclasse *(Subclass)* ou superclasse (*Superclass*) da UML.

Regra 7: *IUniq2Mult*

Numa restrição ORM de **Unicidade** *(uniqueness)* sobre o tipo de papel, esta regra converte para uma restrição UML de multiplicidade *(Multiplicity)* com cardinalidade de *{..1}*.

Regra 8: *MandRole2Mult*

Numa restrição ORM de **obrigatoriedade** *(mandatory)* sobre o tipo de papel, esta regra converte para uma restrição UML de multiplicidade *(Multiplicity)* com cardinalidade de *{1..}*.

Regra 9: *Freq2Mult*

Numa restrição ORM de **frequência** *(Frequency)* sobre o tipo de papel, esta regra converte para uma restrição UML de multiplicidade *(Multiplicity)* com cardinalidade correspondente.

Regra 10: *Subset2Subset*

Para uma restrição ORM de **subconjunto** *(Subset)*, esta regra converte para uma restrição UML de **subconjunto** *(Subset)* corresponde.

Os exemplos que ilustram cada uma das regras de mapeamento foram apresentados no capítulo 3, secção 3.7.

No processo de transformação de modelo, as regras de transformação são especificadas de acordo com uma linguagem (***Model Transformation Language* - MTL**) que define essa transformação.

5.1.2 Linguagem de Transformação de Modelos (MTL)

Uma Linguagem de Transformação de Modelos (MTL) é uma linguagem que define o mecanismo de transformação de um modelo fonte num modelo alvo. Esta linguagem manipula elementos de modelos, considerando os metamodelos utilizados para criar essas instâncias (modelos). Nesse contexto, existem várias linguagens específicas para a transformação de modelos nomeadamente: *ATLAS Transformation Language (*ATL), *Yet Another Transformation Language* (YATL), *Basic Object-oriented Transformation Language* (BOTL) (ATL, 2011).

Nesse trabalho optou-se pela linguagem de transforma ATL, por ser uma das linguagens mais utilizadas neste domínio, e por apresentar um vasto suporte documental. A *ATLAS Transformation Language* (ATL) (ATL, 2011) é uma linguagem de transformação de

modelos que disponibiliza mecanismos e um *kit* de ferramentas para a transformação de modelos, permitindo desenvolver e integrar modelos e metamodelos *Ecore, EMOF*, *KM3* (linguagem de domínio específico para metamodelos) (EMF, 2011). Esta Linguagem foi desenvolvida e é mantida pela OBEO e *AtlanMod* anteriormente chamado de *ATLAS Group* (ATLAS, 2006). Na área da engenharia orientada por modelos *(Model-Driven Engineering (MDE))*, a ATL fornece mecanismos para produzir um conjunto de modelos alvo a partir de um conjunto de modelos de origem. A ATL foi desenvolvida sobre a plataforma *eclipse*[4] sob os termos da *Eclipse Public License*, sendo um componente do *M2M (Machine-to-Machine)*, dentro do projeto de modelação *Eclipse (Eclipse Modeling Project – EMP)* (EMF, 2011).

A ATL tem a seguinte estrutura geral de transformação: (1) o programa de transformações ATL é definida através de módulos (Jouault & Kurtev, 2006) (ATLAS, 2006), em que cada módulo contém, um cabeçalho obrigatório *(header)*, um conjunto de funções livres secundárias chamadas de auxiliares *(helper)* e de um conjunto de regras *(rule)*. Segue-se uma breve descrição de cada um dos destes elementos.

O *header*, dá nome ao módulo de transformação e declara os modelos *fonte* e *alvo*. Começa com uma palavra-chave predefinida *"module"*, seguida pelo nome do módulo. Na instrução *"create"*, a variável **OUT** identifica o modelo de destino. Após a palavra-chave *"from"*, a variável **IN** indica o modelo de origem. A próxima declaração mostra a estrutura do cabeçalho.

```
module [nome módulo];
create OUT: [modelo alvo] from IN: [modelo fonte];
```

Esta informação permite a importação dos metamodelos fonte e alvo correspondentes aos modelos no programa ATL. Caso haja mais do que um modelo de origem e/ou modelos alvo, podem igualmente ser enumerados no segmento de cabeçalho.

Os *helpers* e as regras (*rules*) são os construtores utilizados para implementar as funcionalidades da transformação. O termo "*helper*" vem da especificação da Linguagem de Restrições de Objetos (OCL) (OMG, 2012) e define dois tipos de "*helpers*": operações e atributos. m "*helper*" de atributo não aceita parâmetros sendo aplicado no contexto do módulo ou de um elemento do modelo ATL, nestes casos será

[4] http://www.eclipse.org

referenciado como atributo (*attribute*) em substituição de "*helper*" de atributo. O "*helper*" de operação é usado na especificação de operações (*methods*) à semelhança dos métodos nas linguagens de programação orientadas por objetos (por exemplo Java), sendo aplicado no contexto de um elemento de modelo ou no contexto de um módulo.

Nos "*helpers*" de atributos, os valores de retorno são calculados apenas uma vez, na primeira ocasião em que são aplicados. Por sua vez, nos "*helpers*" de operação, o resultado é calculado sempre que o "*helper*" é chamado, pois pode depender dos valores dos parâmetros. Os "*helpers*" de operação têm um nome, um contexto e um tipo. A definição de parâmetros inclui o nome e o tipo do parâmetro. O "*helper*" de operação ATL é definido de acordo com o seguinte esquema:

```
helper [contexto tipo de contexto]? def:[nome_do_helper (parâmetros)] : [tipo de retorno] = exp;
```

Uma regra (*rule*) é um construtor básico em ATL utilizado para descrever uma transformação lógica. As regras ATL podem ser de dois tipos: **declarativas** ou **imperativas**. Iremos somente focar nas regras declarativas. A descrição das regras imperativas podem ser consultadas em (OMG, 2012).

Uma **regra declarativa** permite combinar alguns tipos dos elementos de um modelo de origem, e gerar a partir deles um certo tipo de elementos no modelo alvo. As regras declarativas são também chamadas de regras combinadas *(matched rules)*. Este tipo de regra permite igualmente especificar o elemento do modelo de origem que deve ser correspondido no modelo alvo, o número e o tipo dos elementos do modelo alvo gerados, e a forma como esses elementos do modelo alvo devem ser inicializados a partir dos elementos da fonte. O trecho que se segue mostra uma regra de transformação declarativa simples da ATL:

```
rule nome_regra {
        from
                var_entrada:tipo_var_entrada[nome_modelo]? [(condição
                )]?
        [whith {
                var1: tipo_var1 = init_exp1;
                ...
                varn: tipo_varn = init_expn;
        }]?
        to
                var_saída1: tipo_var_saída1 [nome_modelo]? (
```

```
                    ...
                )
                ...
                out_varn: tipo_var_saídan [nome_model]? (
                    ...
                )
        [do {
                declarações
        }]?
}
```

No processo de transformação de modelos descrito acima, os metamodelos fonte e alvo tem que obedecer a um meta-metamodelo, como por exemplo *MOF* ou *Ecore*, este último será analisado nesta dissertação.

5.1.3 Meta-metamodelo *Ecore*

O *Ecore* é a linguagem de metamodelação do ***Eclipse Modeling Framework (EMF)*** (Jouault F. , 2009) (EMF, 2011) permitindo definir metamodelos e criar modelos (por instanciação do metamodelo). O metamodelo *Ecore* está alinhado com o EMOF (*Essential Meta-Object Facility*), sendo considerado uma versão simplificada do metamodelo MOF. A figura 5.2 apresenta o metamodelo *Ecore* de acordo com a definição da EMF.

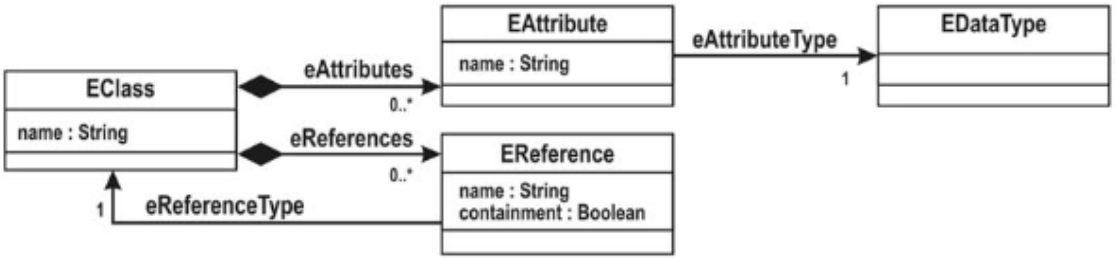


Figura 5. 2 - Metamodelo Ecore simplificado (Budinsky, 2003) (Jouault F. , 2009)

O EMF é uma ferramenta de modelação baseada na linguagem ***Ecore***, a qual está alinhada com a variante EMOF proposta pela OMG. Assim, através da linguagem *Ecore*, permite uma ótima gestão de modelos e metamodelos de formato textual ou gráfico.

O EMF é uma plataforma de modelação para o ambiente de desenvolvimento *Eclipse*. O *Eclipse* permite o desenvolvimento e integração de novas funcionalidades (*plugins*) para gestão de modelos e metamodelos. A EMF é uma ferramenta de modelação baseada no

Eclipse que permite a importação e gestão de modelos e metamodelos. Esta plataforma é bastante útil no desenvolvimento orientado por modelos pois permite simplificar as fases de análise e de implementação no processo de desenvolvimento de sistemas de software. O metamodelo da linguagem do sistema a desenvolver é especificado no formato *Ecore*, sendo deste modo definida a sintaxe da linguagem de domínio da solução. A partir de uma especificação do modelo no formato XMI, a EMF fornece suporte e ferramentas de execução. O ***XML Metadata Interchange (XMI)*** (OMG, 2011) é um padrão da OMG (OMG, 2012) para troca de informação baseado em XML. Um dos objetivos do XMI é facilitar o intercâmbio de metadados entre ferramentas de modelação baseadas na UML e de repositórios de metadados baseados no MOF. Uma vez que o metamodelo *Ecore* define a sintaxe da linguagem, para representar as instâncias desses metamodelos, o EMF recorre ao formato XMI como padrão para a representação de modelos. Assim, a ferramenta EMF possibilita a geração de modelos no formato XMI tendo por base um metamodelo *Ecore*.

5.2 Processo de Transformação Proposto

De acordo com a definição do processo de transformação descrita no capítulo 5.1 que implementa o processo de transformação e o posicionamento dos modelo, metamodelos e metametamodelos respectivamente numa arquitetura de quatro camadas (capitulo 4), encontramos no nível M1 os modelos ORM e UML que descrevem entidades presentes nos modelos ORM e UML apresentados nos ficheiros *xmi (ORM.xmi e UML.xmi)*. O nível M2 descreve os metamodelos das linguagens correspondentes *(ORM.ecore e UML.ecore)*. O nível M3 corresponde ao metametamodelo *ecore* o qual especifica as regras a que metamodelos obedecem.

A figura 5.3 exemplifica o processo de transformação baseado em metamodelos, como foi apresentado na secção 4.4.2 do capítulo anterior. O *modelo ORM (fonte)*, no formato XMI *(ORM.xmi)*, em conformidade com o metamodelo correspondente *(ORM.ecore)*, é transformado para o *modelo UML (alvo)* no formato XMI *(UML.xmi)* em conformidade com o seu metamodelo *(UML.ecore)*. A transformação é definida através de um conjunto de regras de transformação ATL. A definição da transformação *(orm2uml.atl)* está em conformidade com o metamodelo ATL, sendo escrita na linguagem ATL.

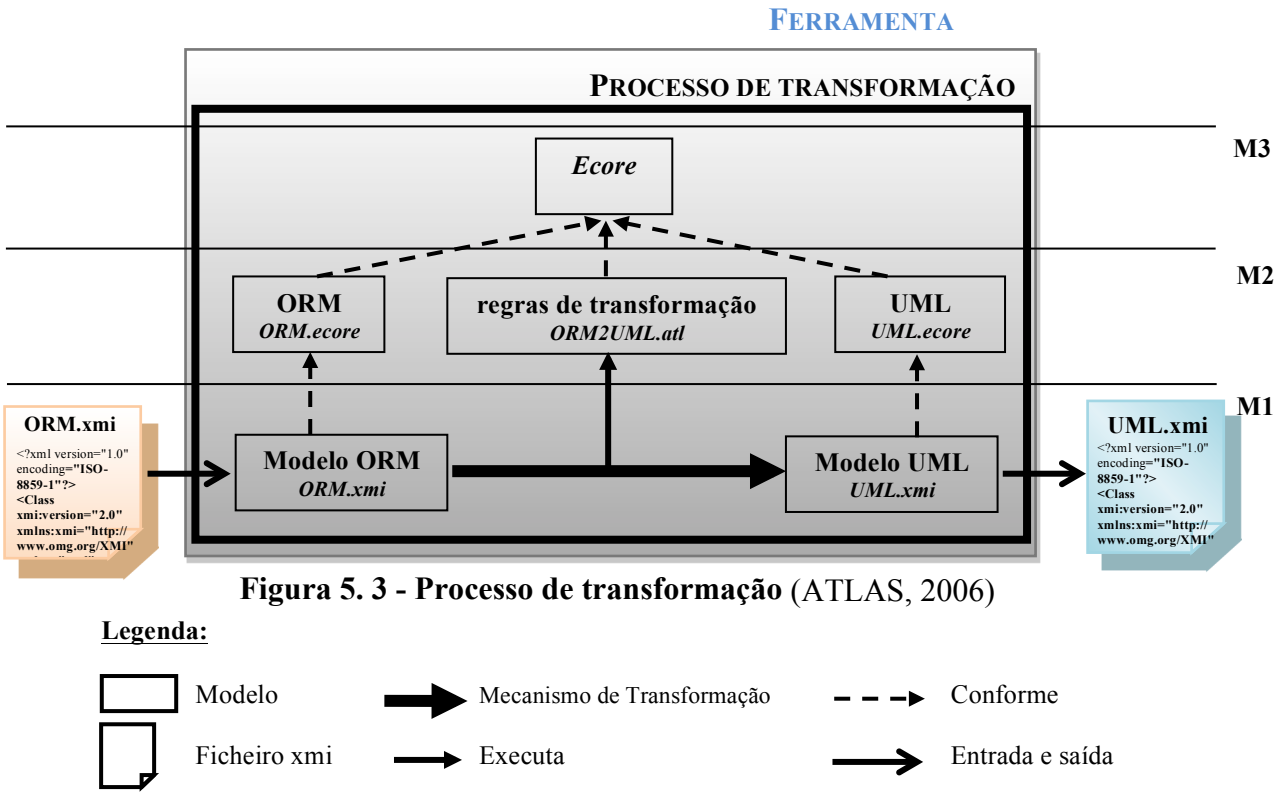


Figura 5. 3 - Processo de transformação (ATLAS, 2006)

Contudo, a transformação entre diagramas ORM e UML, criados nos vários ambientes de modelação, não é compatível com o formato *xmi* de entrada e saída da transformação ATL. Sendo essencial a adequação do processo de transformação baseado em metamodelos, de forma a incluir uma transformação entre diagramas fonte e alvo, para o formato *xmi*.

O processo proposto consiste numa extensão à transformação modelo-para-modelo e inclui as três fases que se descrevem de seguida:

- **Converter o esquema conceptual inicial em um modelo de fonte** - foca a criação de um modelo de fonte baseado num esquema conceptual. Uma vez que o processo de transformação não aceita, como entrada, um esquema conceptual de qualquer formato, a primeira fase do processo envolve uma transformação XSLT entre o formato de esquema conceptual e o formato *xmi* do modelo fonte.
- **Especificação de metamodelos e de regras de transformação** - relacionado com o método para transformar o modelo fonte no modelo alvo, o que inclui

pelo menos dois metamodelos e várias regras de transformações. Esta segunda fase é a mais complexa e inclui várias atividades (ver tabela 5.1).

- **Converter o modelo alvo no esquema conceptual de saída** - criação de um esquema conceptual com base no modelo alvo.
-

Atividade	Práticas
Especificação dos metamodelos	Criação dos metamodelos com base nos conceitos da linguagem fonte e alvo.
Definir as regras de transformação	Descrição das regras de transformação entre os dois domínios
Executar a transformação	Identificação do modelo fonte (formato xmi), executar as regras de transformação e gerar o modelo alvo (formato xmi)

Tabela 5. 1 - Atividades da segunda fase do processo de transformação

De acordo com o processo acima descrito, a figura 5.4 ilustra o trecho das principais regras de transformação de ORM para UML, definidas da primeira atividade da fase 2. A segunda atividade consiste na definição de quatro regras de transformações concretas. A primeira regra, **`rule ORM2UML`** resulta no mapeamento das *metaclass* do modelo ORM e UML. Para o nosso exemplo *"Professor leciona uma Disciplina"* são necessárias três regras de transformação para fazer a conversão de um modelo para o outro, **`rule Role2Association`** (segunda regra na figura 5.4) que permite converter os papéis desempenhados pelo tipos de facto, **`rule Element2Class`**, a terceira regra na figura 5.4 serve para transformar os tipos de objeto *(ObjectType) EntityType* bem como *ValueType*, e **`rule RefMode2Attribute`** (quarta regra na figura 5.4) que converte o esquema de referência ORM (regra 4, descrita nas subsecção 5.1.1 Regras de transformação).

```
rule ORM2Uml {
      from
            input: orm!ORM
      to
            output: uml!UML (
                  id <- input.id,
                  has <- input.has->select(x | not
x.oclIsKindOf(orm!Role)),
```

```
                  includes <- input.has->select(x |
x.oclIsKindOf(orm!Role))
            )
}
rule Role2Association {
      from
            s: orm!Role(s.oclIsKindOf(orm!Role))
      to
            t2: uml!Association(
                  name <-s.roleName,
                  end <- Set{ae1,ae2}
            ),
            ae1: uml!AssociationEnd(
                  otherEnd <- ae2,
                  lower <- 0,
                  upper <- 1
            ),
            ae2: uml!AssociationEnd(
                  otherEnd <- ae1,
                  lower <- 0,
                  upper <- 1
            )
}
rule Element2Class{
      from
            s: orm!Element( s.oclIsKindOf(orm!ObjectType))
      to
            t1: uml!Class(
                  name <- s.name,
                  features1<- s.reference
            )
}
rule ValueType2Attribute{
      from
            input: orm!ValueType(input.oclIsTypeOf(orm!ValueType))
      to
            output: uml!Attribute (
                  name <- input.name, -- name - é o nome da classe
                  type <- input.type
                  )
}
rule RefMode2Attribute {
      from
            input1 : orm!RefMode(input1.oclIsTypeOf(orm!RefMode))
      to
          output1 : uml!Attribute (
                  name <- input1.name, -- name - é o nome do atributo
da classe
                  type <- input1.type
            )
}
```

Figura 5. 4 - Regras de transformação de ORM para UML "Professor leciona uma Disciplina"

A fase 2 do processo termina com a execução de transformação. O programa ATL (*orm2uml.atl*) recebe como entrada um modelo fonte em formato XMI, *"ORM.xmi"* que contém o esquema conceptual do modelo ORM, assim como dois metamodelos ORM e UML.

Este modelo de entrada é processado e efetuam-se as conversões de elementos do *modelo ORM* para elementos do *modelo UML*, através das *regras de transformação* (descritas acima) especificadas no programa ATL. A saída do programa ATL é um modelo em formato XMI *"UML.xmi"*.

A fase 2 do processo é suportada por um ambiente de desenvolvimento que permite: (1) a criação do projeto ATL; (2) a criação dos modelos, metamodelos, programa ATL (regras de transformação); (3) execução, verificação e resultados, que serão apresentados na secção seguinte.

5.3 Ambiente de desenvolvimento

Esta secção apresenta uma explicação geral sobre a ferramenta de transformação escolhida, evidenciando as suas principais características e a implementação passo à passo do processo de transformação proposto.

Muitas das ferramentas de modelação e transformação fazem uso da plataforma *Eclipse* como plataforma padrão de desenvolvimento. Isto deve-se ao fato de que hoje, o *Eclipse* ser considerado uma das melhores ferramentas baseada em modelos através do padrão de modelos ECORE, baseado no padrão EMOF, criado pela OMG. Através do seu modelo .*Ecore*, permite uma boa gestão de elementos de modelos de forma textual ou gráfica. Esta plataforma possibilita o desenvolvimento e integração de novas funcionalidades e *plugins* a outras plataformas já existentes para gestão de modelos.

Assim, o ambiente de desenvolvimento escolhido foi a ferramenta de modelação *Eclipse* na versão *GALILEO*. Este pacote contém uma coleção de componentes de projeto de modelação *Eclipse* que inclui, o ambiente de modelação Eclipse ***(EMF - Eclipse Modeling Framework)***, gráfico ***(GMF - Graphical Modeling Framework)*** (GMF, 2013), MTD XSD/OCL/ UML2, M2T e elementos EMFT integrado. Inclui o *kit* de desenvolvimento de Software/Aplicativos *(SDK - Software Development Kit)* completo, ferramentas de desenvolvimento e código fonte. Note-se que o pacote de modelação possui um *plugin* para a linguagem de transformação escolhida, ATL. Este

plugin ATL disponibilizam uma *interface* gráfico simples, e não só, contém componentes e soluções para a transformação de modelos.

Com o ambiente EMF (EMF, 2011), a plataforma *Eclipse* permite a importação e gestão de metamodelos, através dos quais se podem criar textualmente instâncias deles (modelos). Com a tecnologia GMF (GMF, 2013), os modelos anteriormente criados pela EMF podem agora ser manipulados de forma gráfica, podendo assim, serem criados modelos no padrão EMOF baseados nos seus metamodelos. Por se tratar de um ambiente aberto, a plataforma *Eclipse* permite a transformação de modelos em ATL, QVT ou qualquer outra linguagem de transformação para as quais já existam ou se queira desenvolver um *plugin* que a implemente.

Utilizando as plataformas *Eclipse*, EMF e GMF podemos, desde que existam os metamodelos ORM e UML criar um processo de transformação onde após a criação do modelo ORM se obtenha de forma automática o modelo UML, através de uma transformação entre os elementos dos modelos envolvidos na transformação.

A próxima tarefa, a criação do projeto ATL, será descrita passo à passo nas subseções seguintes.

5.3.1 Criar um projeto ATL

A criação do projeto ATL será feito de acordo com a sequência numérica apresentada na figura 5.5. O projeto será denominada de "*ORM2UML"*. Depois da criação do projeto ATL, procedeu-se a criação da estrutura do projeto, os metamodelos. Nestes metamodelos serão representados as entidades das linguagens ORM e UML.

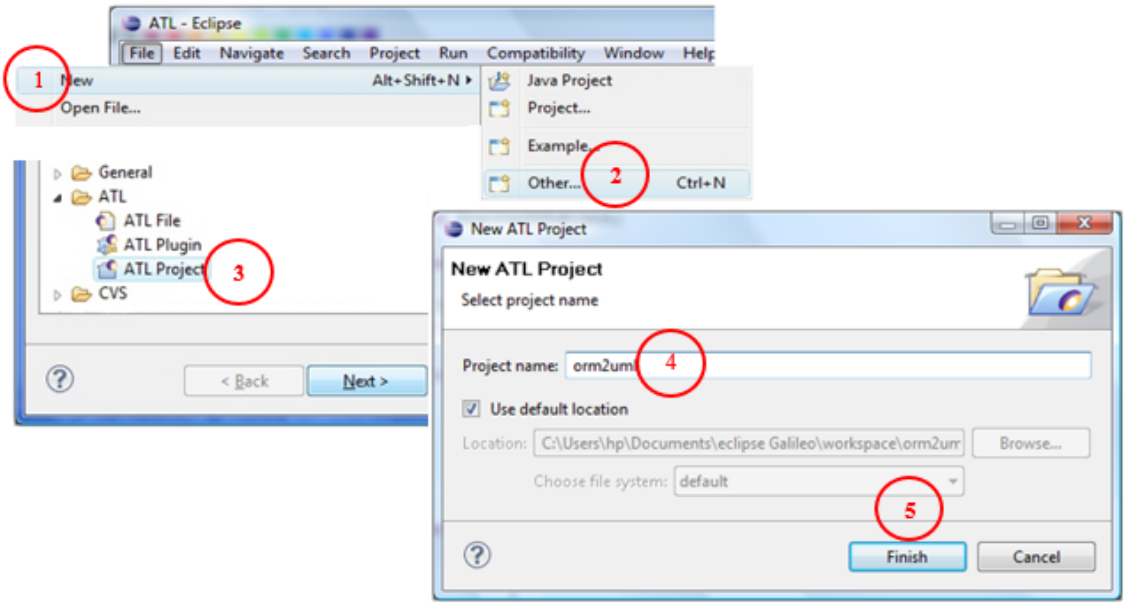


Figura 5. 5 - Criação do projeto ATL

5.3.2 Criação do Metamodelo

Para facilitar a criação do metamodelo *Ecore*, a ferramenta permite a criação de diagramas do tipo *Ecore* (*Ecore Diagram)*, aos quais fica associado um ficheiro *.ecorediag* e um ficheiro *.ecore* (ver figura 5.6).

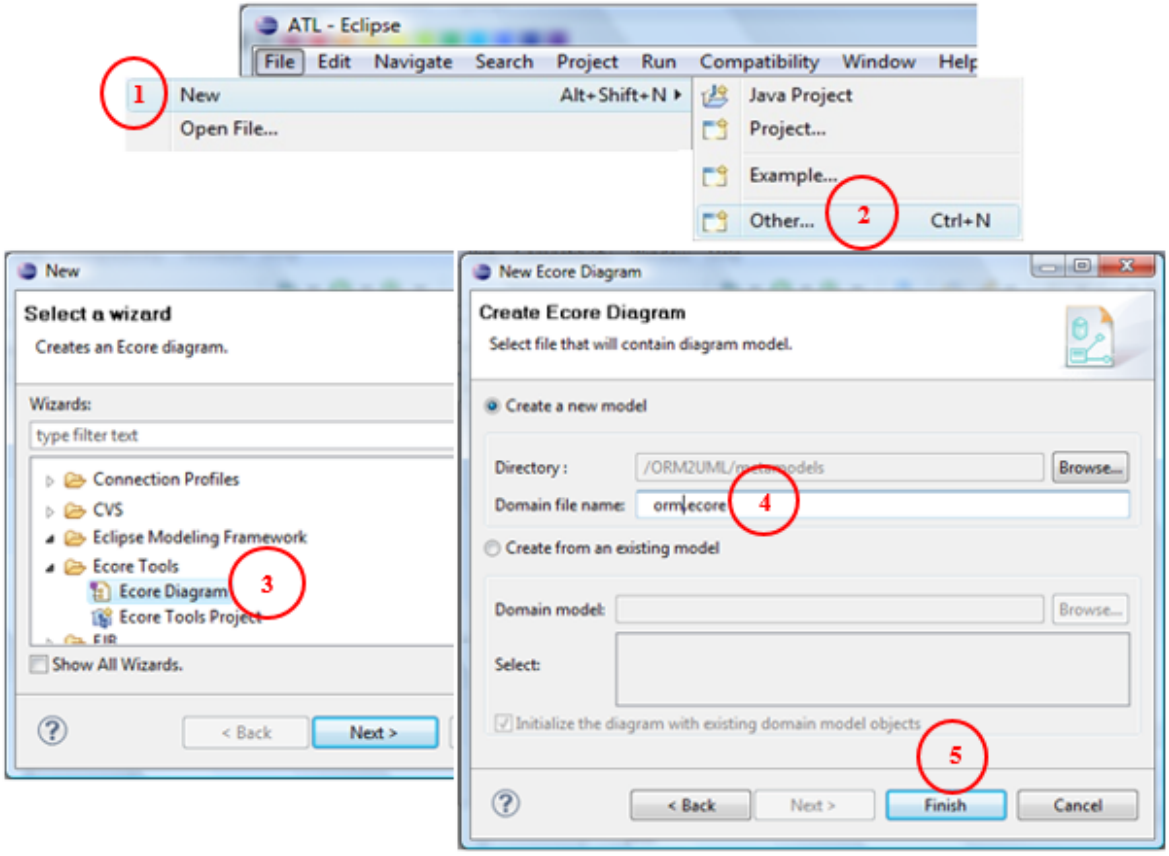


Figura 5. 6 - Criação do metamodelo ORM (*orm.ecore*)

No projeto podemos identificar ambos os ficheiros, *.ecorediag* e *.ecore* (ver figura 5.7).

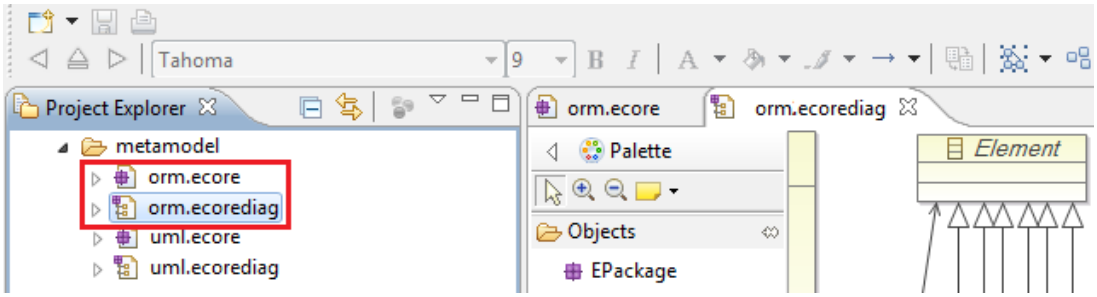


Figura 5. 7 - Diagramas do tipo ecore, orm.ecore e orm.ecorediag

A figura, que se segue, mostra a criação do metamodelo *Ecore* para ORM associado ao ficheiro *.ecorediag (orm.ecorediag)*.

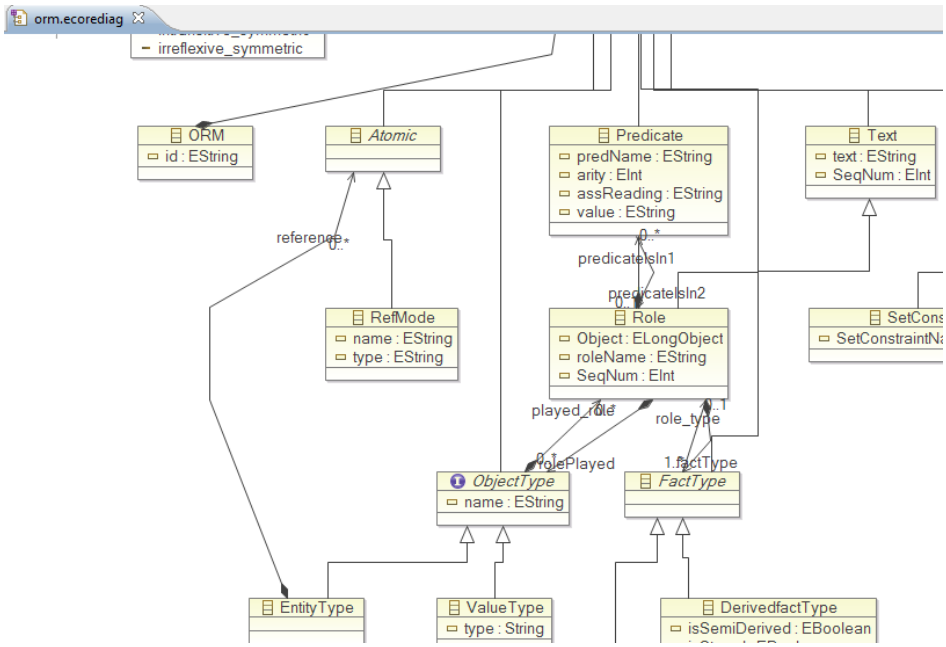


Figura 5. 8 - Criação do metamodelo ecore (orm.ecorediag)

Segue-se o ficheiro *orm.ecore* correspondente (ver figura 5.9).

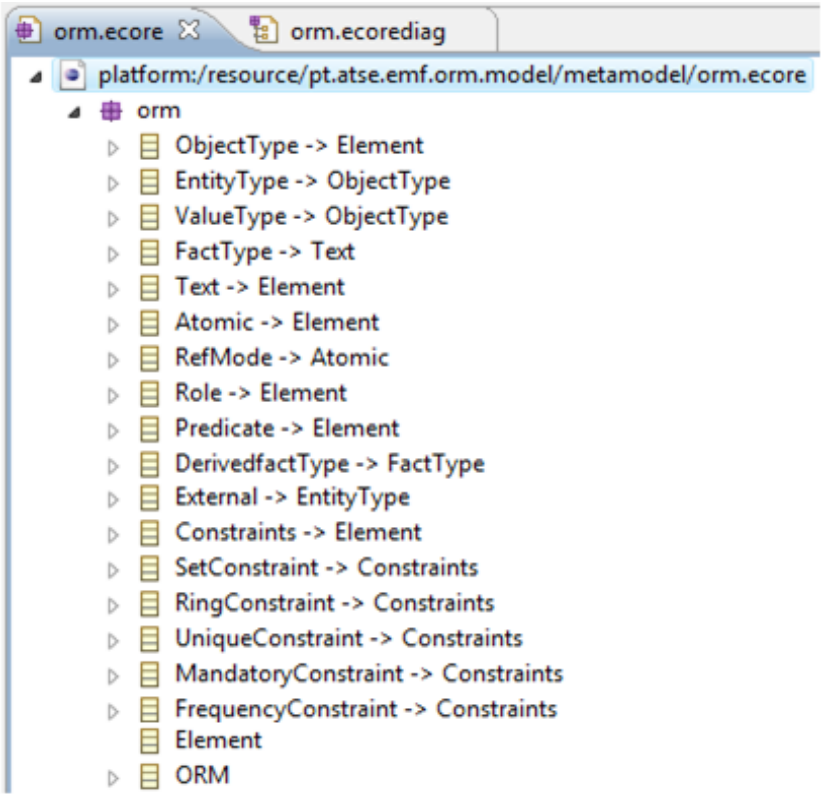


Figura 5. 9 - Ficheiro orm.ecore

Assim foi criado o metamodelo para ORM. Repete-se o mesmo processo para a metamodelo da UML.

Depois de criados os dois metamodelos (ORM, a fonte) e (UML, o alvo), definimos a transformação através das regras de transformação (código na linguagem ATL). A seguir mostra-se o mecanismo de criação do código ATL

5.3.3 O código ATL

A criação do ficheiro ATL segue os passos demonstrados na figura 5.10.

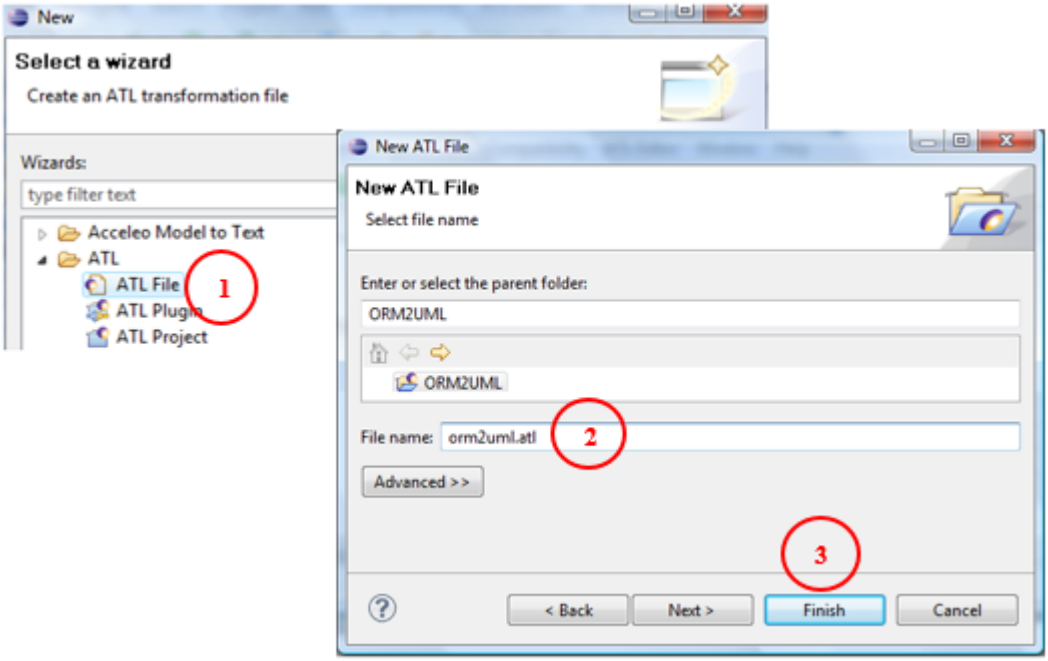


Figura 5. 10 - Criar o ficheiro ATL, *orm2uml.atl*

Apresenta-se de seguida a estrutura do ficheiro *ORM2UML.atl*:

```
-- @path orm=/pt.atse.emf.orm.model/metamodel/orm.ecore
-- @path uml=/pt.atse.emf.orm.model/metamodel/uml.ecore

module ORM2UML;
create OUT : uml from IN : orm;
-----------------------------------------------------------------
-- RUGRAS -----------------------------------------------------------
-- INSTANCIAS -------------------------------------------------------
-----------------------------------------------------------------
rule ORM2Uml {
        from
                input: orm!ORM
        to
                output: uml!UML (
```

```
                    id <- input.id,
                    has <- input.has->select(x | not x.oclIsKindOf(orm!Role)),
                    includes <- input.has->select(x | x.oclIsKindOf(orm!Role))
                )
}
rule Role2Association {
        from
                s: orm!Role(s.oclIsKindOf(orm!Role))
        to
                t2: uml!Association(
                        name <-s.roleName,
                        end <- Set{ae1,ae2}
                ),
                ae1: uml!AssociationEnd(
                        otherEnd <- ae2,
                        lower <- 0,
                        upper <- 1
                ),
                ae2: uml!AssociationEnd(
                        otherEnd <- ae1,
                        lower <- 0,
                        upper <- 1
                )
}
rule Element2Class{
        from
                s: orm!Element( s.oclIsKindOf(orm!ObjectType))
        to
                t1: uml!Class(
                        name <- s.name,
                        features1<- s.reference
                )
}
rule RefMode2Attribute {
        from
                input1 : orm!RefMode(input1.oclIsTypeOf(orm!RefMode))
        to
           output1 : uml!Attribute (
                        name <- input1.name,
                        type <- input1.type
                )
```

Figura 5. 11 - Estrutura do ficheiro de transformação *ORM2UML.atl*

5.3.4 Execução

A execução da transformação no ambiente *eclipse*, e de acordo com a ATL, é feito através de uma execução rápida e controlada. Os passos para execução do programa são:

i. Criar um ficheiro com um determinado modelo ORM expresso em *xmi.* Caso o formato não seja reconhecido pelo ambiente, deve executar a fase 1 do processo que envolve uma transformação XSLT;

ii. Configurar a execução da transformação;

iii. Executar a transformação.

iv. Se pretende visualizar o modelo num diagrama de classes, deverá executar a fase 3 do processo que envolve uma transformação XSLT;

i. Criar um ficheiro com um modelo ORM

De seguida, cria-se o modelo inicial ORM de entrada (*ORM.xmi*). Na criação do metamodelo ORM, foi criada uma *metaclass* ORM com uma referência *(has)* para o supertipo *(Element)* do qual todos os modelos iniciais devem ser criados.

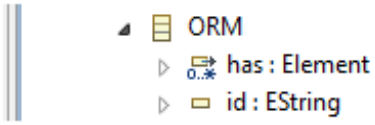


Figura 5. 12 - Estrutura da metaclass ORM

Com esta *metaclass* pode-se, de uma forma simples, criar uma instância do nosso metamodelo, criando um modelo EMF através da opção de criação de instâncias dinâmicas *(Create Dynamic Instance).* Segue-se a demonstração da criação da instância dinâmica do ficheiro *Ecore (orm.ecore)* de entrada, que iremos designar de *ORM.xmi*.

A criação do ficheiro *ORM.xmi* de entrada faz-se com a opção *"Create Dynamic Instance ...."* da seguinte forma:

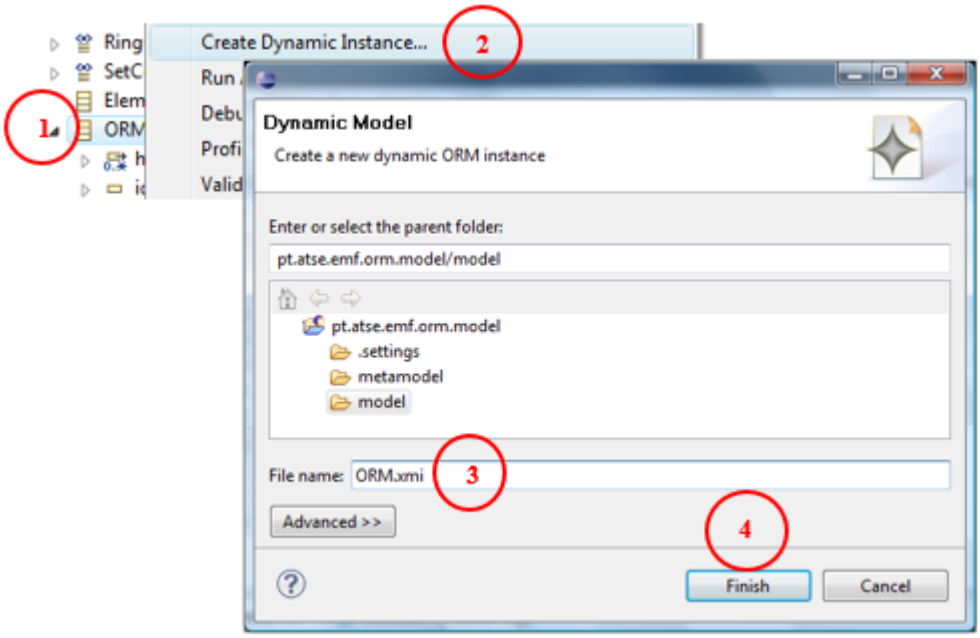


Figura 5. 13 - Criação da instância de um ficheiro ecore (ficheiro ORM.xmi)

Depois de guardar o ficheiro, voltamos a abri-lo usando *"Open with ..."* com a opção *"Sample Reflective Ecore Model Editor"*.

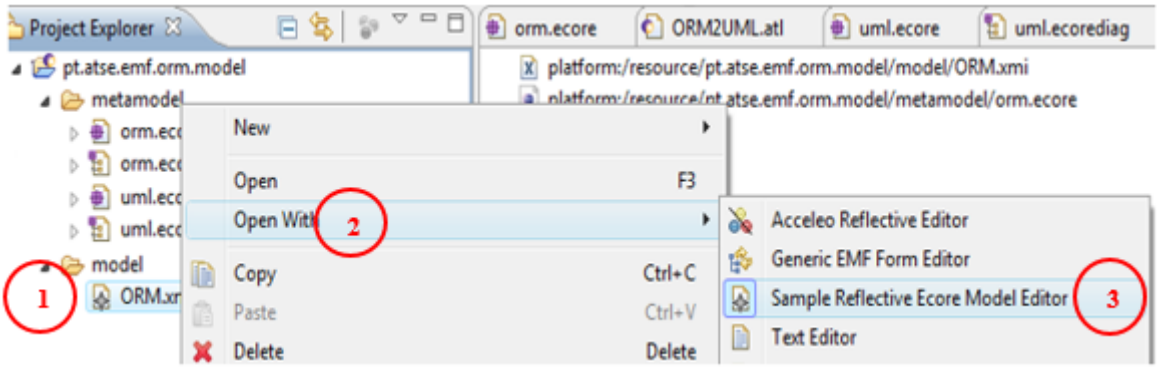


Figura 5. 14 - Como abrir o ficheiro *ORM.xmi* em modo gráfico.

Aparecendo a seguinte figura:

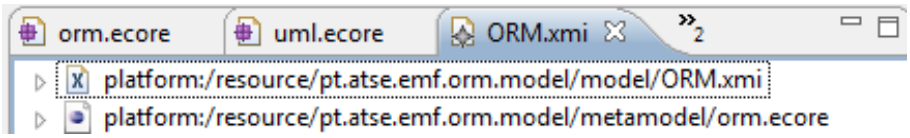


Figura 5. 15 - Abertura do ficheiro ORM.xmi

Carregue na seta para expandir o ficheiro *ORM.xmi* (primeira Linha: *platform:/resource/pt.atse.emf.orm.model/model/ORM.xmi*), e passa a:

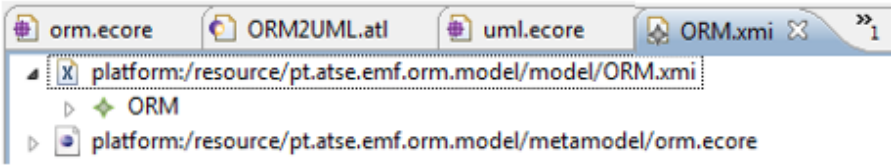


Figura 5. 16 - Ficheiro ORM.xmi

Para inserir os elementos, carrega-se sobre o ORM, e Insere-se por exemplo, os tipos de entidade *(Has Entity Type) Professor* e *Disciplina* do nosso exemplo.

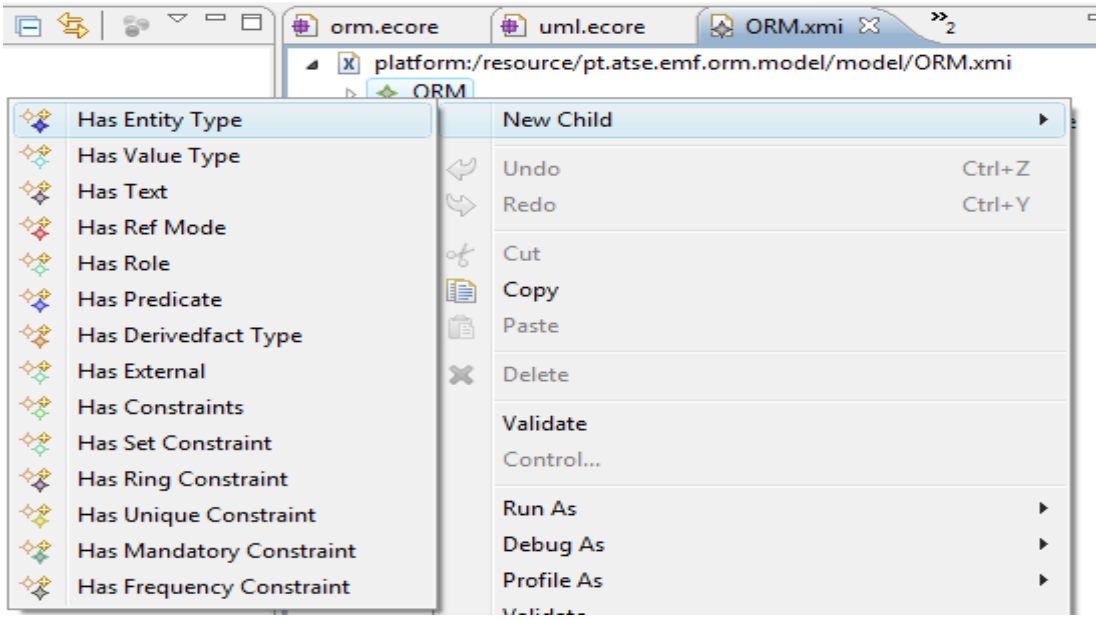


Figura 5. 17 - Inserir elementos no modelo de entrada *ORM.xmi*

Obtemos o seguinte:

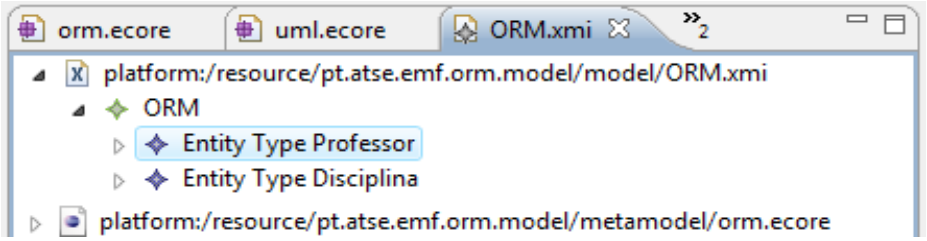


Figura 5. 18 - Tipo de entidades Professor e Disciplina inseridos no modelo ORM.xmi

Insere-se o esquema de referência *Ambrósio* para o tipo de entidade *Professor* e o esquema de referência *TICs* para o tipo de entidades *Disciplina* e o papel desempenhado pelas duas entidades *(leciona)*.

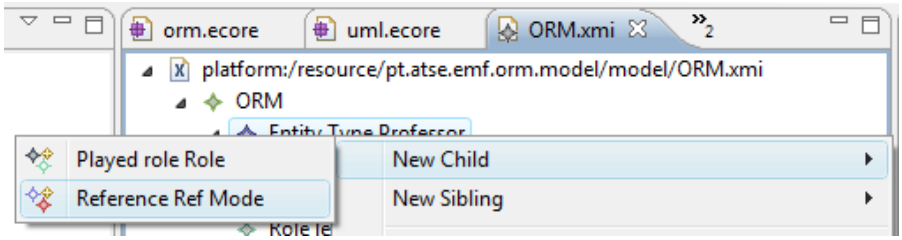


Figura 5. 19 - Inserção do esquema de referência e tipo de papel no modelo ORM

No final temos o modelo de entrada ORM do nosso exemplo:

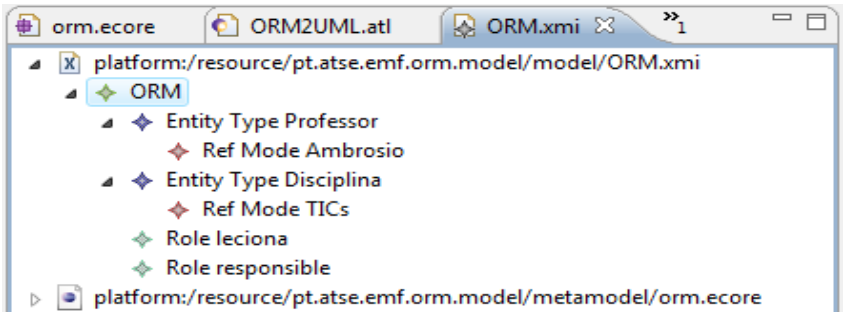


Figura 5. 20 - Modelo ORM de entrada

O trecho abaixo ilustra o ficheiro em modo de texto do modelo de entrada ORM *(ORM.xmi)*.

```
1<?xml version="1.0" encoding="ASCII"?>
2<ORM:ORM xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://
3  <has xsi:type="ORM:EntityType" name="Professor">
4    <reference xsi:type="ORM:RefMode" name="Ambrosio" type="String"/>
5  </has>
6  <has xsi:type="ORM:EntityType" name="Disciplina">
7    <reference xsi:type="ORM:RefMode" name="TICs" type="string"/>
8  </has>
9  <has xsi:type="ORM:Role" roleName="leciona"/>
10 <has xsi:type="ORM:Role" roleName="responsible"/>
11</ORM:ORM>
```

Figura 5. 21 - Ficheiro de texto do modelo de entrada ORM

ii. Configuração da Execução

Depois de efetuados todos os passos de implementação de transformação, já se pode proceder à execução. A figura 5.22 mostra os passos para configurar a execução. Começa-se por selecionar o ficheiro ATL *(orm2uml.atl)*, e depois procede-se à configuração *(Run Configurations...)*.

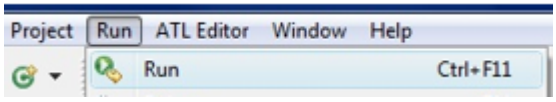


Figura 5. 22 - Configuração da execução do projeto ATL

Será exibida uma caixa de diálogo com várias informações já preenchidas: o módulo de ATL (o nosso ficheiro de transformação, *orm2uml.atl*), e os metamodelos (*orm.ecore* e *uml.ecore*). É necessário adicionar (navegando na área de trabalho *"Workspace"*) o modelo de entrada. O modelo de *fonte* (**IN**:, está de acordo com metamodelo de entrada, orm) é o modelo que queremos transformar, isto é, o nosso ficheiro *ORM.xmi*. O modelo *alvo* (**Out**:, está de acordo com metamodelo de saída, uml) é o modelo a ser gerado, navegasse na área de trabalho *"Workspace"* para encontrar o projeto e nomeia-se o ficheiro, neste caso, *"UML.xmi"*.

A figura 5.23 mostra o ecrã para a configuração da execução do projeto, segundo a sequência numérica.

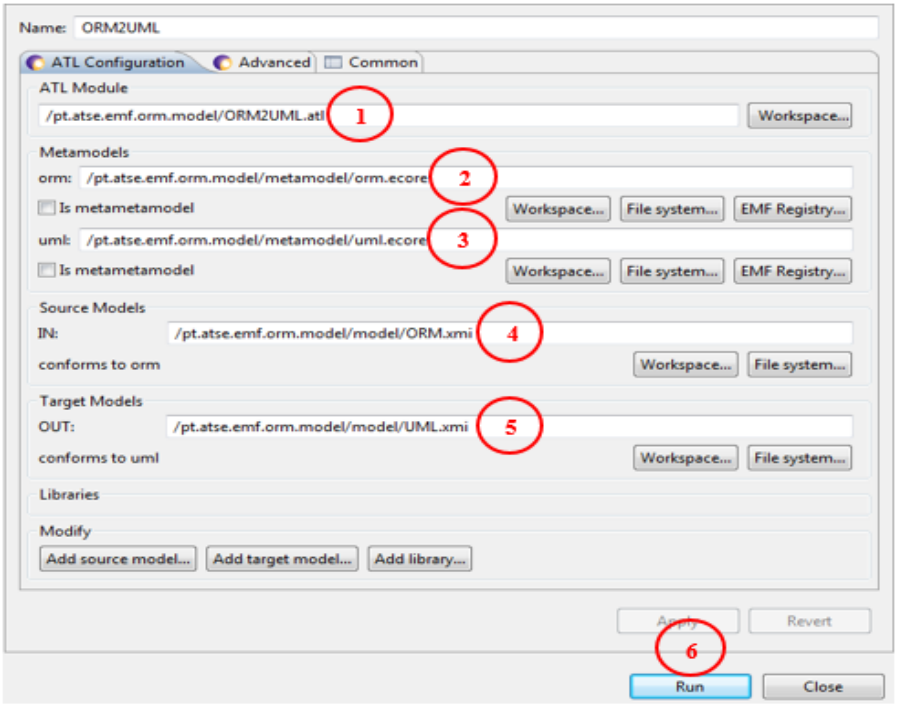


Figura 5. 23 - Correr a transformação

(1) Seleciona-se o ficheiro ATL, *ORM2UML.atl*;
(2) Seleciona-se o metamodelo de entrada, *orm.ecore*;
(3) Seleciona-se o metamodelo de saída, *uml.ecore*;
(4) Seleciona-se o ficheiro com o modelo de entrada, *ORM.xmi*;
(5) Define-se o nome do ficheiro para o modelo de saída, *UML.xmi*;

(6) Finalmente corre-se a transformação.

iii. Executar a Transformação

O ficheiro gerado, *UML.xmi*, contém a transformação dos elementos ORM (***tipo de entidades***, ***esquema de referências***, …) em elementos UML (***classe***, ***atributo***, …) do nosso exemplo. Se tudo estiver de acordo com os metamodelos e as regras, o programa é executado e gera o modelo de saída.

A estrutura do ficheiro *UML.xmi* será a seguinte:

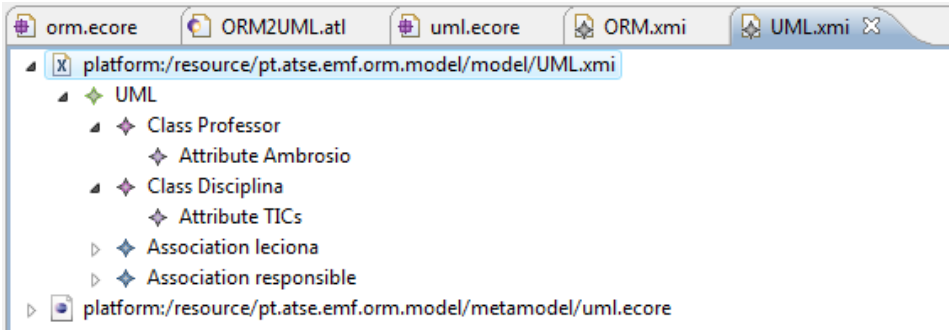


Figura 5. 24 - Modelo de saída em formato ecore (UML)

O trecho abaixo ilustra o ficheiro em modo de texto do modelo de saída da UML *(UML.xmi)*

```
1<?xml version="1.0" encoding="ISO-8859-1"?>
2<uml:UML xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://
3  <has xsi:type="uml:Class" name="Professor">
4    <features1 xsi:type="uml:Attribute" name="Ambrosio" type="String"/>
5  </has>
6  <has xsi:type="uml:Class" name="Disciplina">
7    <features1 xsi:type="uml:Attribute" name="TICs" type="string"/>
8  </has>
9  <includes name="leciona">
10   <end otherEnd="//@includes.0/@end.1" upper="1"/>
11   <end otherEnd="//@includes.0/@end.0" upper="1"/>
12 </includes>
13 <includes name="responsible">
14   <end otherEnd="//@includes.1/@end.1" upper="1"/>
15   <end otherEnd="//@includes.1/@end.0" upper="1"/>
16 </includes>
17</uml:UML>
```

Figura 5. 25 - Ficheiro de texto do modelo de saída UML

Capítulo 6 – Caso de estudo

Nos capítulos anteriores, apresentamos as linguagens de modelação ORM e UML, as abordagens de transformação, bem como a estratégia metodológica de transformação. Neste capítulo, e no seguimento, pretende- -se validar o trabalho realizado ao aplicar o método de transformação de modelos proposto para num caso de estudo. Deste modo, este capitulo este estruturado de seguinte forma: a secção 6.1 descreve o caso de estudo. A secção 6.2 apresenta o modelo ORM no domínio da solução. A secção 6.3 a criação da representação conceptual para o sistema do caso de estudo. A secção 6.4 descreve a implementação do modelo de entrado do caso de estudo. A secção 6.5 apresenta a implementação do caso de estudo com a ferramenta adotada. Para concluir, a secção 6.6 apresenta os resultados obtidos.

6.1 Caso de estudo: Dados relativos a Professores/Alunos nos serviços académicos

Para que seja possível a integração de ambientes de transformação de modelos, no processo de desenvolvimento de software, é necessário, em primeiro lugar, definir o domínio específico do problema. Assim, o caso de estudo analisado nesta dissertação é, um sistema para gestão dos serviços académicos, particularmente no que diz respeito à informação sobre professores e/ou alunos.

No domínio de gestão dos serviços académicos vai ser possível ter informações sobre o professor, designadamente, o nome, sobrenome, a sua morada, o seu género, e o departamento a que pertence, etc. Igualmente, pode-se ter informação sobre o aluno, como por exemplo, o nome, sobrenome, morada, o género, e a que ano esta inscrito.

Após a conclusão da definição do domínio do problema relacionado com do caso de estudo, segue-se o desenho do modelo no domínio da solução. Este trabalho consiste na transformação de modelos ORM para modelo UML então, neste caso, o nosso modelo inicial será expresso em ORM.

6.2 Desenho do Modelo fonte

O modelo inicial servirá de base à definição do modelo de entrada do caso de estudo. Este modelo define os *tipos de factos*, as *entidades* e *restrições* do domínio da solução. Este modelo expresso em ORM foi desenhado na ferramenta *Microsoft Visual Studio 2010* com *plugin NORMA*, conforme é ilustrado a seguir na figura 6.1. A figura 6.1 apresenta o modelo ORM, onde estão representados os vários tipos de entidade, tipos de valor, os seus tipos de relacionamento e restrições, dos quais podemos destacar:

- **Cada** *Professor* **lecciona** *Disciplina* – Tipo de entidade Professor e Disciplina
- **Cada** *Pessoa* **tem um** *sobrenome* – Tipo de valor (*sobrenome*)
- **Cada** *Professor* **lecciona** *Disciplina* – Papel no relacionamento **(lecciona)**
- **Cada** *Pessoa* **tem no máximo um** *sobrenome* – restrição de Unicidade e de Obrigatoriedade.
- **Cada** *Pessoa* **é de um** determinado *Género* (masculino "**M**" ou feminino "**F**") – restrição de Subtipo.
- **Cada** *Aluno* **é uma** *Pessoa* **que é do** *Género* (masculino "**M**" ou feminino "**F**") - restrição de Subtipo.
- **Cada** *Professor* **é uma** *Pessoa* **que é do** *Género* masculino "**M**" ou feminino "**F**") - restrição de Subtipo.
- **Cada** *Professor* **é chefiado no máximo** por **dois** Professores - restrição de Frequência (≤2).
- **Qualquer** *Professor* que **chefia** um *Departamento* deve **ser um membro** desse *Departamento* - restrição de Subconjunto.
- **Se** o *Professor* ***Ambrósio*** for chefiado pelo *Professor* ***Hugo*** e o *Professor* ***Hugo*** for chefiado pelo *Professor* ***Sérgio***, então o *Professor* ***Ambrósio*** não é chefiado pelo *Professor* ***Sérgio*** - restrição de Anel, Intransitiva.
- **Cada** *Pessoa* **não pode ser** *Aluno* **é** *Professor* **ao mesmo tempo -** restrição de Anel, Exclusiva.

Nas subsecções seguintes, prossegue-se o processo de transformação do modelo fonte ORM para diagramas de classe da UML.

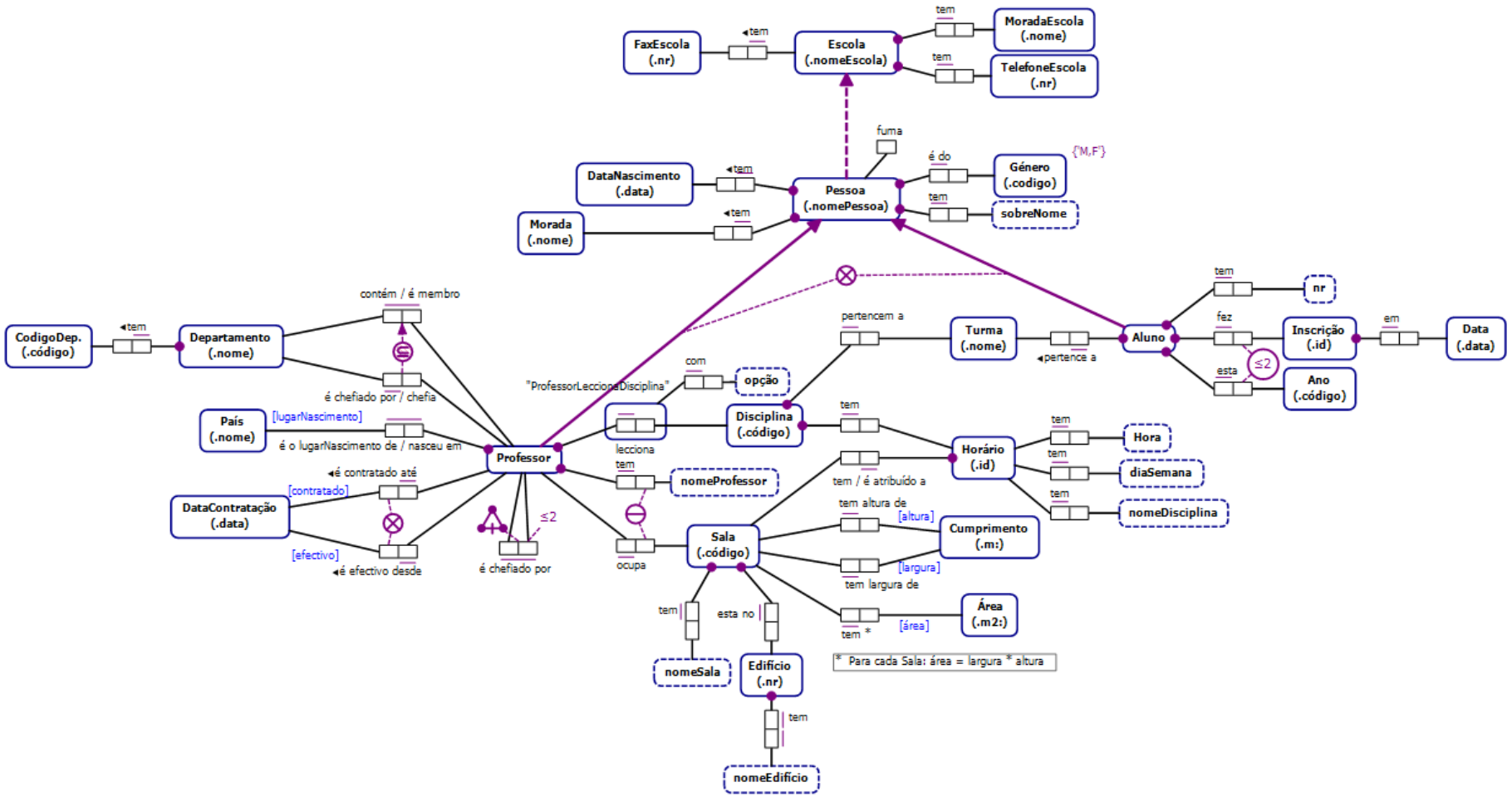


Figura 6. 1 - Metamodelo fonte (ORM)

6.3 Execução

Os passos para a execução, do caso de estudo segundo um projeto ATL, previamente criado, são:

1. Criação do modelo com os conceitos da ORM;
2. Conversão do modelo ORM para o formato xmi aceite pela ferramenta;
3. Execução da transformação por execução das regras de transformação e criação do modelo UML no formato xmi;
4. Conversão do modelo UML no formato xmi para o formato aceite pela ferramenta de modelação.

Nas subseções que se seguem, apresenta-se uma descrição dos vários passos.

6.3.1 As representações conceptuais

Para este caso de estudos iremos desenhar dois modelos, ORM e UML. Nesta secção apresentamos exemplos de entidades de ambos os modelos. Temos o modelo ORM de entrada, (figura 6.2 a)), que tem **Professores** e **Disciplinas**. Cada **Professor** tem um **nome**, e cada **Disciplina** contém um **código** da disciplina. O objetivo é realizar uma transformação a partir de um modelo em ORM para o correspondente em UML. Isso significa que a cada elemento do modelo de entrada irá corresponder a um elemento no modelo de saída (UML). No final, temos apenas um modelo UML correspondente (figura 6.2 b)).

Tipo de entidade: Professor
Modo de Referência: .nome.
Tipo de Factos:
Professor tem nome.

Tipo de entidade: Disciplina
Modo de Referência: código.
Tipo de Factos:
Disciplina tem código.

Tipo de Factos:
Professor lecciona Disciplina.

Restrição de Unicidade:
Cada Professor lecciona **uma** Disciplina.

Restrição de Obrigatoriedade:
Cada Professor lecciona **exatamente uma** Disciplina.

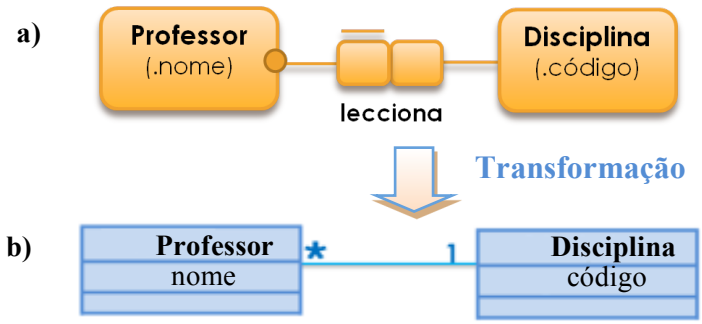


Figura 6. 2 – Modelo ORM a) e Modelo b)

Classe: Professor
Atributo: nome.

Classe: Disciplina
Atributo: código.

Restrição de Multiplicidade: Muitos-para-Um (n:1)
Varios Professor lecciona **uma** Disciplina.

As tabelas abaixo descrevem a correspondência entre os conceitos das linguagens ORM e da UML referente às estruturas e respetivas restrições.

Instâncias / Estruturas	
ORM	**UML**
Tipo de Entidade (Professor)	Classe (Professor)
Tipo de Entidade (Disciplina)	Classe (Professor)
Esquema de referência (nome)	Atributo (nome)
Esquema de referência (código)	Atributo (código)
Associação de tipo de objeto	Classes de Associação

Tabela 6. 1 - Correspondência básica entre os conceitos conceptuais da ORM e UML

Restrições	
ORM	**UML**
Unicidade Interna	Multiplicidade de...1 §
Regra de Obrigatoriedade Simples	Multiplicidade de 1...(muitos–para-Um (n:1))

§ = Correspondência incompleta dos conceitos correspondentes

Tabela 6. 2 - Correspondência básica das restrições da ORM e UML

O processo de transformação foi previamente detalhado no capítulo 5, o qual envolve uma sequência de procedimentos de transformação de modelos, descreve o conceito de implementação, apresentando os vários passos presentes na transformação, tendo como exemplo *"Professor* leciona *uma Disciplina"*, retirado do caso de estudo.

Nas próximas secções apresentamos um exemplo da transformação do modelo ORM aplicado no caso de estudo. Começamos com a criação do modelo de entrada.

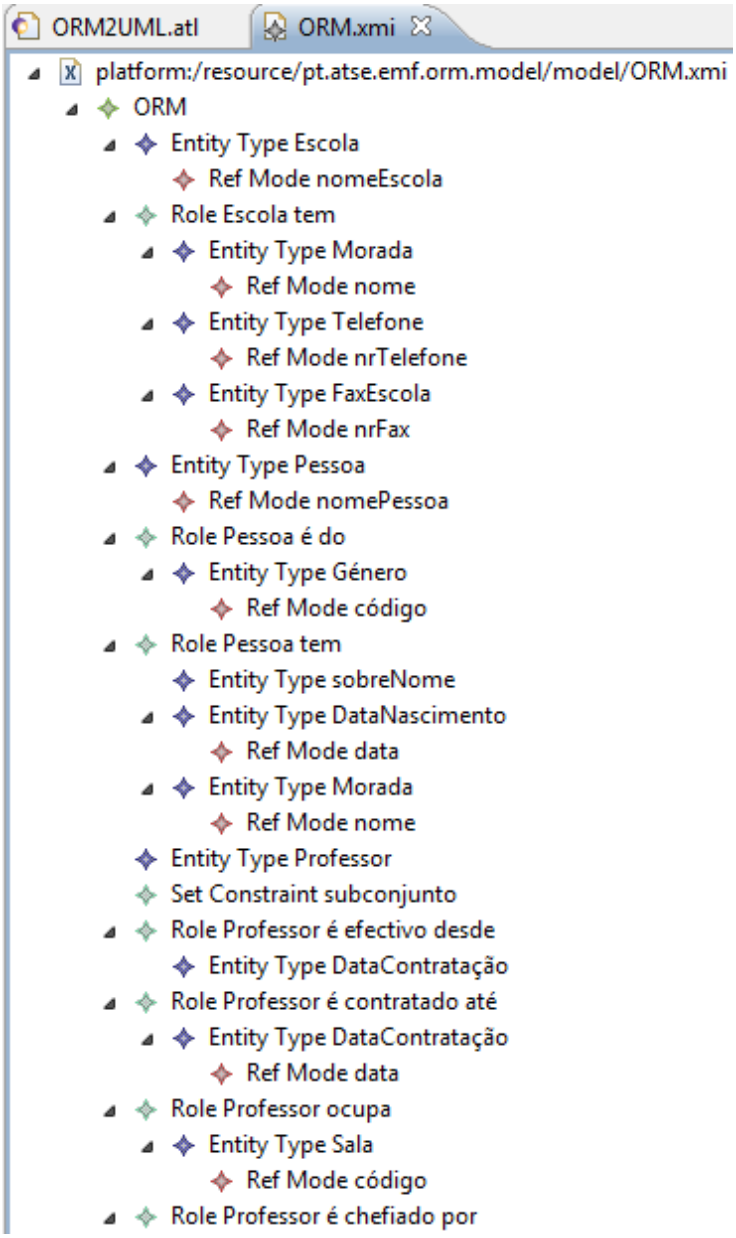
6.3.2 Modelo ORM de entrada

O modelo ORM anterior da figura 6.1, não está no formato aceite pela ferramenta de transformação de modelos, terá de ser convertido para o formato XML adequado. Através de uma conversão XSLT convertemos os dados numa gramática XML para a gramática aceite pela ferramenta ATL. A figura 6.3 apresenta um exemplo de aplicação da linguagem XLST para transformar documentos no formato XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <xsl:stylesheet version="1.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
xmlns="http://metamodel/1.0">
<xsl:output method="xml" indent="yes"/>
 <xsl:template match="/ecore:EPackage">
  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="http://metamodel/1.0">
   <xsl:for-each select="eClassifiers[not(@xsi:type='ecore:EEnum')]">
     <xsl:element name="{@name}">
     <xsl:for-each select="./eStructuralFeatures[@xsi:type='ecore:EAttribute']">
       <xsl:if test="@eType = 'ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString'">
            <xsl:attribute name="{@name}">String</xsl:attribute>
             </xsl:if>
<xsl:if test="@eType = 'ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EDate'">
            <xsl:attribute name="{@name}">Date</xsl:attribute>
             </xsl:if>
<xsl:if test="@eType = 'ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EDouble'">
            <xsl:attribute name="{@name}">Double</xsl:attribute>
             </xsl:if>
<xsl:if test="@eType = 'ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInteger'">
          <xsl:element name="{@name}">
            <xsl:attribute name="type">Integer </xsl:attribute>
          </xsl:element>
        </xsl:if>
          </xsl:for-each>
        </xsl:element>
   </xsl:for-each>
   </xmi:XMI>
 </xsl:template>
</xsl:stylesheet>
```

Figura 6. 3 - folha de estilo XSLT

Após a transformação XLST, obtemos um modelo no formato aceite pela ferramenta de transformação *(ORM.xmi)*, ou seja, um modelo EMF. A figura 6.4 apresenta um exemplo de um modelo ORM ao qual se aplicará a transformação de modelos e que corresponde a uma instanciação do metamodelo ORM.


ORM2UML.atl
ORM.xmi
platform:/resource/pt.atse.emf.orm.model/model/ORM.xmi
ORM
Entity Type Escola
Ref Mode nomeEscola
Role Escola tem
Entity Type Morada
Ref Mode nome
Entity Type Telefone
Ref Mode nrTelefone
Entity Type FaxEscola
Ref Mode nrFax
Entity Type Pessoa
Ref Mode nomePessoa
Role Pessoa é do
Entity Type Género
Ref Mode código
Role Pessoa tem
Entity Type sobreNome
Entity Type DataNascimento
Ref Mode data
Entity Type Morada
Ref Mode nome
Entity Type Professor
Set Constraint subconjunto
Role Professor é efectivo desde
Entity Type DataContratação
Role Professor é contratado até
Entity Type DataContratação
Ref Mode data
Role Professor ocupa
Entity Type Sala
Ref Mode código
Role Professor é chefiado por

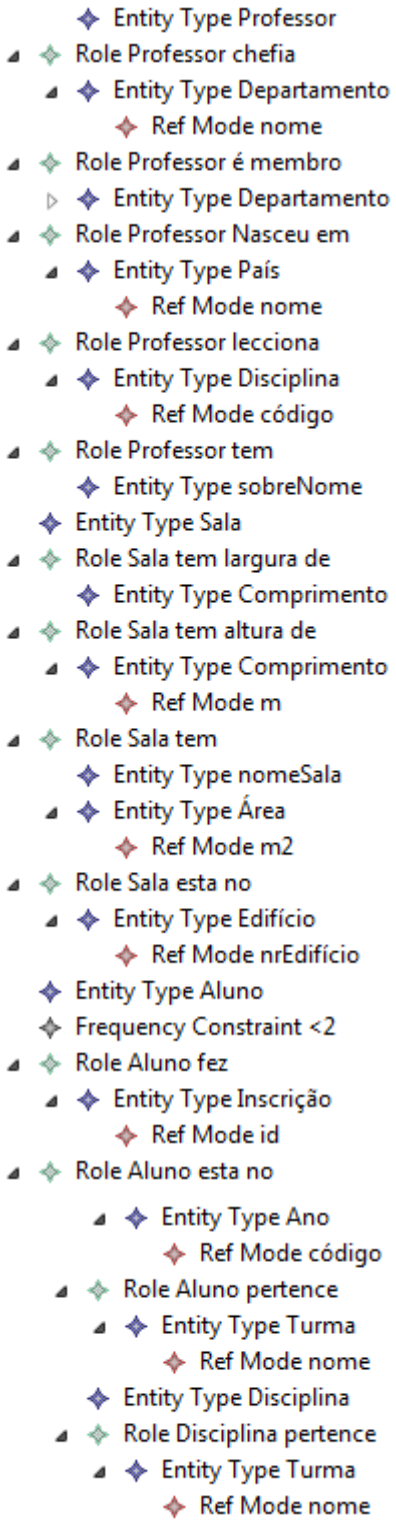


Figura 6. 4 - Modelo inicial ORM de entrada na norma xmi, ORM.xmi

6.3.3 Transformação específica entre ORM e UML

O código, que se apresenta de seguida, realiza a transformação de ORM para UML e foi aplicado no exemplo do caso de estudo. O código foi escrito recorrendo apenas aos

recursos declarativos de ATL. Nesta seção, vamos explicar a lógica de algumas regras do programa relacionadas com o exemplo.

```
-- @path orm=/pt.atse.emf.orm.model/metamodel/orm.ecore
-- @path uml=/pt.atse.emf.orm.model/metamodel/uml.ecore

1. module ORM2UML;
2. create OUT : uml from IN : orm;
-------------------------------------------------------------------
--
-- RUGRAS -----------------------------------------------------------
--
-- INSTANCIAS -------------------------------------------------------
--
-------------------------------------------------------------------
--
3. rule ORM2Uml {
4.    from
5.          input: orm!ORM
6.    to
7.          output: uml!UML (
8.                id <- input.id,
9.                has <- input.has->select(x | not
10. x.oclIsKindOf(orm!Role)),
11.               includes <- input.has->select(x |
12. x.oclIsKindOf(orm!Role))
13.         )
14.}
15. rule Role2Association {
16.   from
17.         s: orm!Role(s.oclIsKindOf(orm!Role))
18.   to
19.         t2: uml!Association(
20.               name <-s.roleName,
21.               end <- Set{ae1,ae2}
22.         ),
23.         ae1: uml!AssociationEnd(
24.               otherEnd <- ae2,
25.               lower <- 0,
26.               upper <- 1
27.         ),
28.         ae2: uml!AssociationEnd(
29.               otherEnd <- ae1,
30.               lower <- 0,
31.               upper <- 1
32.         )
33. }
34. rule Element2Class{
35.   from
36.         s: orm!Element( s.oclIsKindOf(orm!ObjectType))
37.   to
38.         t1: uml!Class(
39.               name <- s.name,
40.               features1<- s.reference
41.         )
42. }
43. rule RefMode2Attribute {
```

```
44.   from
45.         input1 : orm!RefMode(input1.oclIsTypeOf(orm!RefMode))
46.   to
47.       output1 : uml!Attribute (
48.               name <- input1.name, -- name - é o nome do atributo
da
49. classe
50.               type <- input1.type
51.         )
52.}
…
…
```

Figura 6. 5 - Algumas regras do código ATL

A transformação específica de transformação entre ORM e UML pode ser dividida em duas partes lógicas. A primeira parte que contém o cabeçalho obrigatório *(header)*, e os auxiliares *(helper)*. A segunda parte com as regras de transformação *(rules)*. Uma vez que a especificação da transformação é declarativa, não existe ordem de execução explícita entre as partes. (Jouault & Kurtev, 2006)

O código das linhas 1-2, do *header*, identifica o módulo do programa ATL *(ORM2UML)* e declara os modelos fonte (ORM) e alvo (UML).

A regra **`rule ORM2Uml`** transforma o modelo inicial ORM em modelo da UML (linhas 3-14). A regra **`rule Role2Association`** transforma o *tipo de factos* com mais de dois papéis *(n> 2)* para uma associação de classe da UML, com cardinalidade correspondente nas extremidades da associação (linhas 15-33). Se houver uma restrição de unicidade *(uniqueness)* sobre o tipo de papel, esta regra cria uma restrição de multiplicidade em UML de "*{n:1,1:n,1:1, m:n.}*". O mesmo ocorrerá se, sobre o tipo de papel, houver uma restrição de obrigatoriedade de papel, esta regra origina uma restrição de multiplicidade de "*{n:1,1:n,1:1, m:n.}*". A regra **`rule Element2Class`** transforma um elemento em ORM, que pode ser tipos de entidades ou tipo de valor, num elemento em UML que será classificado como Classe (linhas 34-42).

A regra **`rule RefMode2Attribute`** transforma o esquema de referência de um tipo de entidade num atributo primário da classe no modelo UML "*Attribute {P}*" (linhas 43-52).

6.3.4 Resultados

Depois de efetuado o processo de transformação do modelo ORM produzindo o modelo UML no formato xmi, conforme mostra a figura abaixo.

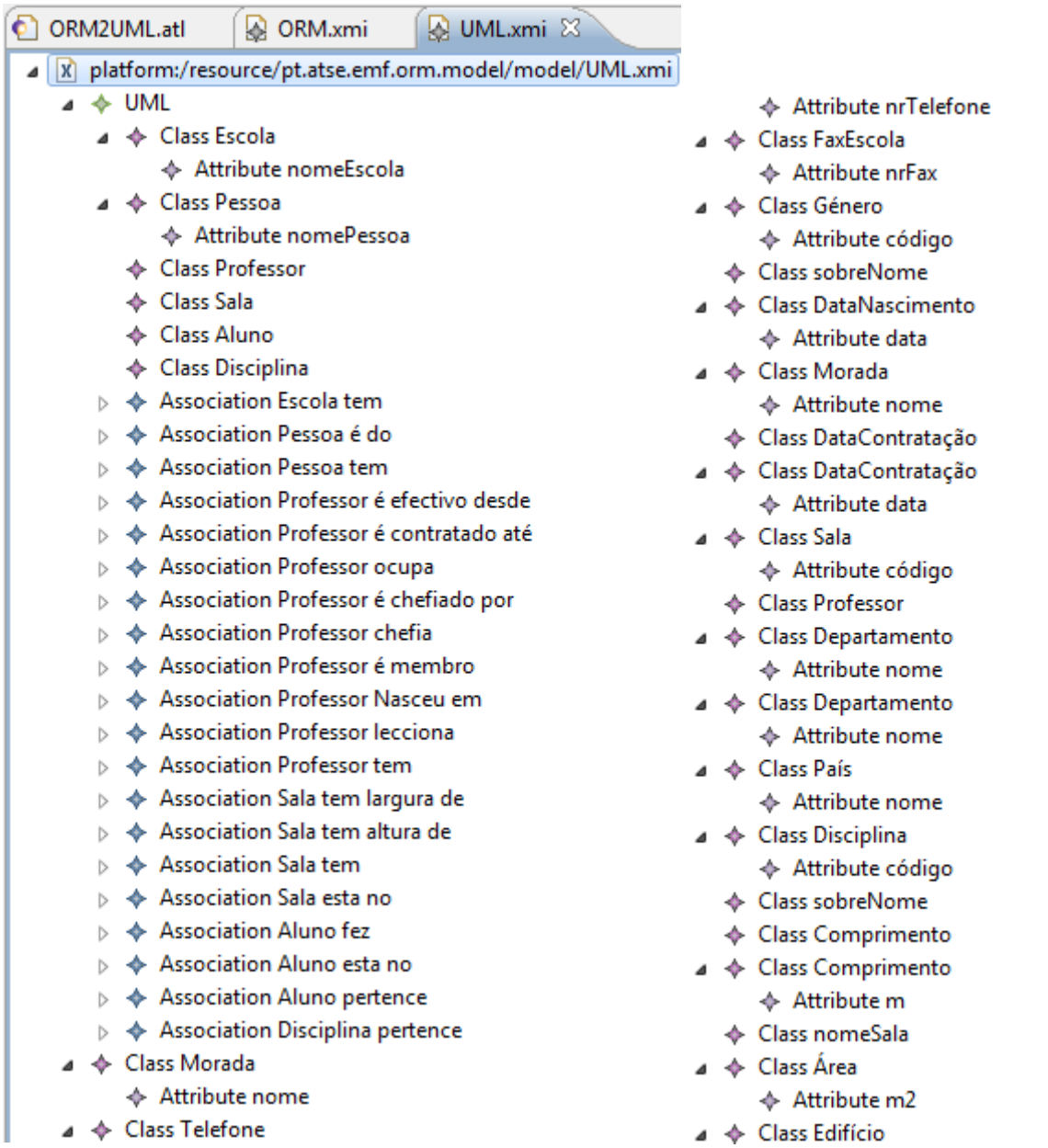


Figura 6. 6 - Trecho do ficheiro de saída UML.xmi

O modelo UML gerado em seguida pode ser transformado em modo texto. O resultado pode ser vido na figura abaixo.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:uml="platform:/resource/pt.atse.emf.orm.model/metamodel/uml.ecore">
  <uml:UML>
    <has xsi:type="uml:Class" name="Escola">
      <features1 xsi:type="uml:Attribute" name="nomeEscola"/>
    </has>
```

```
    <has xsi:type="uml:Class" name="Pessoa">
      <features1 xsi:type="uml:Attribute" name="nomePessoa" type="string"/>
    </has>
    <has xsi:type="uml:Class" name="Professor"/>
    <has xsi:type="uml:Class" name="Sala"/>
    <has xsi:type="uml:Class" name="Aluno"/>
    <has xsi:type="uml:Class" name="Disciplina"/>
    <includes name="Escola tem">
      <end otherEnd="/0/@includes.0/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.0/@end.0" upper="1"/>
    </includes>
    <includes name="Pessoa é do">
      <end otherEnd="/0/@includes.1/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.1/@end.0" upper="1"/>
    </includes>
    <includes name="Pessoa tem">
      <end otherEnd="/0/@includes.2/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.2/@end.0" upper="1"/>
    </includes>
    <includes name="Professor é efectivo desde">
      <end otherEnd="/0/@includes.3/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.3/@end.0" upper="1"/>
    </includes>
    <includes name="Professor é contratado até">
      <end otherEnd="/0/@includes.4/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.4/@end.0" upper="1"/>
    </includes>
    <includes name="Professor ocupa">
      <end otherEnd="/0/@includes.5/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.5/@end.0" upper="1"/>
    </includes>
    <includes name="Professor é chefiado por">
      <end otherEnd="/0/@includes.6/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.6/@end.0" upper="1"/>
    </includes>
    <includes name="Professor chefia">
      <end otherEnd="/0/@includes.7/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.7/@end.0" upper="1"/>
    </includes>
    <includes name="Professor é membro">
      <end otherEnd="/0/@includes.8/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.8/@end.0" upper="1"/>
    </includes>
    <includes name="Professor Nasceu em">
      <end otherEnd="/0/@includes.9/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.9/@end.0" upper="1"/>
    </includes>
    <includes name="Professor lecciona">
      <end otherEnd="/0/@includes.10/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.10/@end.0" upper="1"/>
    </includes>
    <includes name="Professor tem">
      <end otherEnd="/0/@includes.11/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.11/@end.0" upper="1"/>
    </includes>
    <includes name="Sala tem largura de">
      <end otherEnd="/0/@includes.12/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.12/@end.0" upper="1"/>
    </includes>
    <includes name="Sala tem altura de">
      <end otherEnd="/0/@includes.13/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.13/@end.0" upper="1"/>
    </includes>
    <includes name="Sala tem">
      <end otherEnd="/0/@includes.14/@end.1" upper="1"/>
```

```
      <end otherEnd="/0/@includes.14/@end.0" upper="1"/>
    </includes>
    <includes name="Sala esta no">
      <end otherEnd="/0/@includes.15/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.15/@end.0" upper="1"/>
    </includes>
    <includes name="Aluno fez">
      <end otherEnd="/0/@includes.16/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.16/@end.0" upper="1"/>
    </includes>
    <includes name="Aluno esta no">
      <end otherEnd="/0/@includes.17/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.17/@end.0" upper="1"/>
    </includes>
    <includes name="Aluno pertence">
      <end otherEnd="/0/@includes.18/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.18/@end.0" upper="1"/>
    </includes>
    <includes name="Disciplina pertence">
      <end otherEnd="/0/@includes.19/@end.1" upper="1"/>
      <end otherEnd="/0/@includes.19/@end.0" upper="1"/>
    </includes>
  </uml:UML>
  <uml:Class name="Morada">
    <features1 xsi:type="uml:Attribute" name="nome" type="string"/>
  </uml:Class>
  <uml:Class name="Telefone">
    <features1 xsi:type="uml:Attribute" name="nrTelefone"/>
  </uml:Class>
  <uml:Class name="FaxEscola">
    <features1 xsi:type="uml:Attribute" name="nrFax"/>
  </uml:Class>
  <uml:Class name="Género">
    <features1 xsi:type="uml:Attribute" name="código" type=""/>
  </uml:Class>
  <uml:Class name="sobreNome"/>
  <uml:Class name="DataNascimento">
    <features1 xsi:type="uml:Attribute" name="data"/>
  </uml:Class>
  <uml:Class name="Morada">
    <features1 xsi:type="uml:Attribute" name="nome"/>
  </uml:Class>
  <uml:Class name="DataContratação"/>
  <uml:Class name="DataContratação">
    <features1 xsi:type="uml:Attribute" name="data"/>
  </uml:Class>
  <uml:Class name="Sala">
    <features1 xsi:type="uml:Attribute" name="código"/>
  </uml:Class>
  <uml:Class name="Professor"/>
  <uml:Class name="Departamento">
    <features1 xsi:type="uml:Attribute" name="nome" type="string"/>
  </uml:Class>
  <uml:Class name="Departamento">
    <features1 xsi:type="uml:Attribute" name="nome" type="string"/>
  </uml:Class>
  <uml:Class name="País">
    <features1 xsi:type="uml:Attribute" name="nome" type="string"/>
  </uml:Class>
  <uml:Class name="Disciplina">
    <features1 xsi:type="uml:Attribute" name="código"/>
  </uml:Class>
  <uml:Class name="sobreNome"/>
  <uml:Class name="Comprimento"/>
  <uml:Class name="Comprimento">
```

```
    <features1 xsi:type="uml:Attribute" name="m"/>
  </uml:Class>
  <uml:Class name="nomeSala"/>
  <uml:Class name="Área">
    <features1 xsi:type="uml:Attribute" name="m2"/>
  </uml:Class>
  <uml:Class name="Edifício">
    <features1 xsi:type="uml:Attribute" name="nrEdifício"/>
  </uml:Class>
  <uml:Class name="Inscrição">
    <features1 xsi:type="uml:Attribute" name="id"/>
  </uml:Class>
  <uml:Class name="Ano">
    <features1 xsi:type="uml:Attribute" name="código"/>
  </uml:Class>
  <uml:Class name="Turma">
    <features1 xsi:type="uml:Attribute" name="nome" type="string"/>
  </uml:Class>
  <uml:Class name="Turma">
    <features1 xsi:type="uml:Attribute" name="nome" type="string"/>
  </uml:Class>
</xmi:XMI>
```

Figura 6. 7 - Trecho do ficheiro de saída UML.xmi em modo texto, gerado a partir do Eclipse v. Galileo

Capítulo 7 – Conclusão

Este capítulo descreve as considerações finais deste trabalho. Este capítulo está estruturado da seguinte forma. A secção 7.1 apresenta a conclusão de uma forma geral, a secção 7.2 incide sobre as conclusões mais específicas, concretamente focados nos objetivos alcançados e as dificuldades deparadas, a secção 7.3 descreve as intenções de trabalhos futuros nesta área.

7.1 Conclusões Gerais

No contexto atual em que existem inúmeras aplicações isoladas que interagem em si de uma forma global, cada vez mais se torna necessária a utilização de mecanismos de transformação que permitam a integração destas aplicações.

O envolvimento com o universo das transformações de modelos traz algumas implicações das quais se podem destacar alguns pontos que resumem os aspetos gerais desse universo de transformação de modelos:

- A transformação de modelos utiliza os mapeamentos, mas também contém outras informações como, por exemplo, a condição no modelo fonte para permitir essa transformação, a linguagem do modelo fonte, a linguagem do modelo alvo, linguagem de modelação, linguagem de transformação, a ferreamente de transformação, etc.
- A construção de uma linguagem de transformação de modelos, que exige a familiarização com conceitos, tais como, a especificação e implementação, a transformação de modelos e suas particularidades.
- O processo de transformação de modelos que não só se limita a linguagem de transformação e vai mais além, pois considera ainda a máquina de transformação *(engine)* com características muito especificas: ter a capacidade de aceder modelos a partir de repositórios com a especificação MOF, aceder e navegar fácil na estrutura do metamodelo, capacidade de usar os relacionamentos definidos na linguagem de transformação e, ser capaz de gerar novos modelos a partir de modelos origem, de acordo com especificações definidas.

- O processo de transformação de modelos no contexto da arquitetura de software baseada no MDA *(Model Driven Architecture)*. Este alinhamento com a arquitetura MDA, para além de definir o uso das especificações MOF *(Meta Object Facility)*, UML *(Unified Modeling Language)* e CWM *(Common Warehouse Metamodel)*, como padrões básicos para a representação dos modelos, separa a especificação da implementação. Isto permite através de transformação de modelos produzir modelos de aplicações conceptual, lógica e a geração de código para uma plataforma específica.

7.2 Conclusões Específicas

Nesta secção são apresentadas as "metas parciais" que, ao serem alcançadas conjuntamente, acabarão construindo uma solução para o objetivo geral desta dissertação. Assim sendo, a seguir iremos apresentar os objetivos alcançados em cada fase do processo de transformação de modelos ORM para modelos UML.

7.2.1 Objetivos alcançados

Estes objetivos cobrem alguns aspetos específicos do objetivo geral que foram apresentados no capítulo 1.

Assim, os objetivos alcançados para o objetivo geral são:

- Foram apresentados dois modelos de referência (ORM e UML) que permitiram comparar e identificar as características comuns e/ou especificas de ambas (Capítulos 2 e 3 respectivamente).
- Foi descrita uma metodologia que definiu o processo de transformação de modelos (ver figura 5.1) padrão, e a sequência de procedimentos utilizados para fazer a transformação de um modelo fonte para um modelo alvo. Com base nesse processo, foi definido uma estrutura (ver Figura 5.4) aplicada na transformação de modelos com respeito às linguagens de modelação estudadas neste trabalho, com apenas um único objetivo ORM → UML (capítulo 5).
- Para aplicar o processo de transformação de forma automática foi escolhido o ambiente *eclipse* na versão *Galileo*, que incorpora ferramentas de modelação EMF, e a linguagem de transformação ATL. A ferramenta recebe como entrada o modelo conceptual ORM no formato *xmi* e gera o modelo lógico de saída UML em *xmi*.

Grande parte das linguagens de transformação de modelos trabalha de forma unidirecional, não permitindo a definição de transformações que possam ser executadas em ambas as direções, tanto no sentido direto *(PIM→PSM)* quanto no sentido inverso *(PIM←PSM)*. A ATL é uma destas linguagens, por esse motivo neste trabalho não foi implementada o processo de transformação inverso *(ORM←UML)*. Este aspeto não fez parte dos objetivo principais da Dissertação. Todavia, todo o processo de conversão dos principais tipos de dados, relacionamentos e restrições de ORM para UML foi alcançado.

É importante ressalvar que nem todo o tipo de restrições no modelo fonte foram implementadas devido a não correspondência direta, ou outras formas de representação no modelo alvo.

A solução apresentada é apenas uma entre várias soluções possíveis para o caso estudo.

Contudo, em qualquer trabalho é natural que exista algum tipo de dificuldade associado na elaboração do mesmo. Assim sendo, a secção seguinte apresenta as dificuldades encontradas neste trabalho.

7.2.2 Dificuldades

Em geral, as dificuldades deparadas se referem especificamente devido à alta complexidade do processo de meta-modelação do EMOF (versão simplificada do MOF). Esta complexidade obriga a familiarização com conceitos como, metadados, modelo, metamodelo e meta-metamodelo. Exige um conhecimento profundo da estrutura MOF, e ainda o domínio da forma como o MOF representa estes metamodelos.

Outra dificuldade evidente foi de não conhecer a linguagem de transformação ATL, a linguagem para especificação de restrições de objetos (OCL), que são linguagens declarativas para descrever as regras que se aplicam aos modelos UML padrão. Estas linguagens exigem mais do que uma simples abordagem, é necessário um conhecimento específico da sua estrutura e sintaxe.

A familiarização com as ferramentas de modelação conceptual baseada em ORM (*NORMA)*, que era de todo desconhecida, as ferramentas de modelação *Eclipse*, *Enterprise Architect v.7.5* para os modelos UML.

7.3 Trabalhos Futuros

As intenções de trabalhos futuros podem ser resumidas com a pretensão de um aprofundamento deste tema, que estaria caracterizado em expandir as capacidades das linguagens de transformação e das máquinas de transformação, em aspetos como:

- Evolução da linguagem ATL, onde seriam implementadas novas variabilidades possibilitando o processo de transformação bidirecional (conversão nos dois sentidos).
- A implementação de um algoritmo simples, compacto e funcional, que apresente uma estrutura de conversão com novas formas de correspondência entre as linguagens de modelação ORM→UML.

Bibliografia

Anneke G. Kleppe, Jos B. Varmer, Wim Bast. 2003. *MDA Explained: the model driven architecture: practice and promise.* s.l. : Person Education, Inc., 2003.

ATL. 2011. ATL. *ATL - A Model Transformation Technology.* [Online] 2011. [Cited: 2011 28-Outubro.] http://www.eclipse.org.

ATLAS, Group. 2006. ATL: Atlas Transformation Language. *ATL User Manual.* 2006 Fevereiro, Vol. VII.

Bollen, P. 2002. *A formal transformation from Object Role Models to UML class diagrams.* Toronto : s.n., 2002.

Bourret, R. 2007. Consulting, writing, and research in XML and databases. *XML and Databases.* [Online] 2007. http://www.rpbourret.com/xml/ProdsXMLEnabled.htm.

Budinsky, Frank. 2003. *Eclipse modeling framework:.* Boston : Addison Wesley, 2003.

Chen, P. P. 1976. The entity - relationship model - toward a unified view of data. *ACM Transactions on database systems.* 1976, Vol. I, pp. 9-36.

Eclipse Modeling Framework. 2011. EMF. *Eclipse Modeling Framework Project (EMF).* [Online] 2011. http://www.eclipse.org/modeling/emf/.

Firesmith, D., Graham, I., & Henderson-Sellers, B. 1998. *Open Modeling Language (OML) - Reference Manual.* SIGS.

Graphical Modeling Framework. 2013. GMF. *Graphical Modeling Framework (GMF) Runtime.* [Online] 2013. http://www.eclipse.org/modeling/gmf/.

Halpin, T. 2008. *Information Modeling and Relational DataBase.* Second Edition. USA : Academic Press, 2008.

Halpin, T. and Cuyler, D. 2003. *Metamodels for Object-Role Modeling.* USA : s.n., 2003.

Halpin, T. e Bloesch, A. 1999. Data modeling in UML and ORM: a comparison. Oct-Dec de 1999, Vol. 10, pp. 4-13.

Halpin, T. 1998. *Object-Role Modeling (ORM/NIAM).* USA : Microsoft Corporation, 1998.

Halpin, T. 2005. *ORM 2 Graphical Notation.* 2005 01-September.

Halpin, T. 2007. Fact-Oriented Modeling: Past, Present and Future. *Fact-Orinted Modeling: Past, Present and Future.* USA : Utah, 2007, pp. 20-36.

Halpin, T. 2013. *Object Role Modeling.* [Online] http://www.orm.net/pdf/ORM2.pdf.

Halpin, T. 2009. The ORM Foundation. *ORM.* [Online] 2009. http://www.ormfoundation.org/.

Jouault, F. and Kurtev, I. 2006. *Transforming Models with ATL.* [prod.] University of Nantes) ATLAS Group (INRIA & LINA. Nantes : s.n., 2006.

Jouault, Frédéric. 2009. *Model Transformation with ATL.* Nantes, França : s.n., 2009.

Miller, Joaquin and Mukerji, Jishnu. 2003. *MDA Guide V1.0.1.* 2003 12-Junho.

Mellor, Stephen J., et al. 2004. *MDA Distilled Principles of Model-Driven Architecture.* U. S : Addison Wesley, 2004.

Meta Object Facility. 2013. MetaObject Facility (MOF). *Meta Object Facility.* [Online] 2013. http://www.omg.org/mof.

Model Driven Architecture. 2001. OMG Papers on the MDA. *MDA Presentations and Papers.* [Online] 2001 01-07. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

Model Driven Architecture. 2013. Model Driven Architecture (MDA). *Model Driven Architecture.* [Online] 2013. http://www.omg.org/mda/.

Object Management Group, Inc. 2010. Object Management Group. *OMG "Object Management Group".* [Online] 4.01, Object Management Group, Inc., 2010. http://www.omg.org/.

Object Management Group, Inc. 2011. About OMG. *OMG Document.* [Online] 2011. http://www.omg.org/gettingstarted/gettingstartedindex.htm.

Object Management Group, Inc. 2011. Documents Associated with MOF/XMI Mapping. *OMG XMI 2.4.* [Online] 2.4 - Beta 2, 2011 Março. [Cited: 2012 10-Janeiro.] http://www.omg.org/spec/XMI/2.4/Beta2/.

Object Management Group, Inc. 2012. Object Management Group. *OMG "Object Management Group".* [Online] 4.01, Object Management Group, Inc., 2012. http://www.omg.org/.

Object Management Group, Inc. 2012. OMG Object Constraint Language (OCL). *OMG OCL 2.3.1.* [Online] 2012 Janeiro. http://www.omg.org/spec/OCL/2.3.1.

Object Management Group, Inc. 2004. Metamodel and UML Profile for Java. *OMG Specifications.* [Online] 2004 2-Fevereiro. [Cited: 2010 9-12.] http://www.omg.org/cgi-bin/apps/doclist.pl. formal/04-02-02.

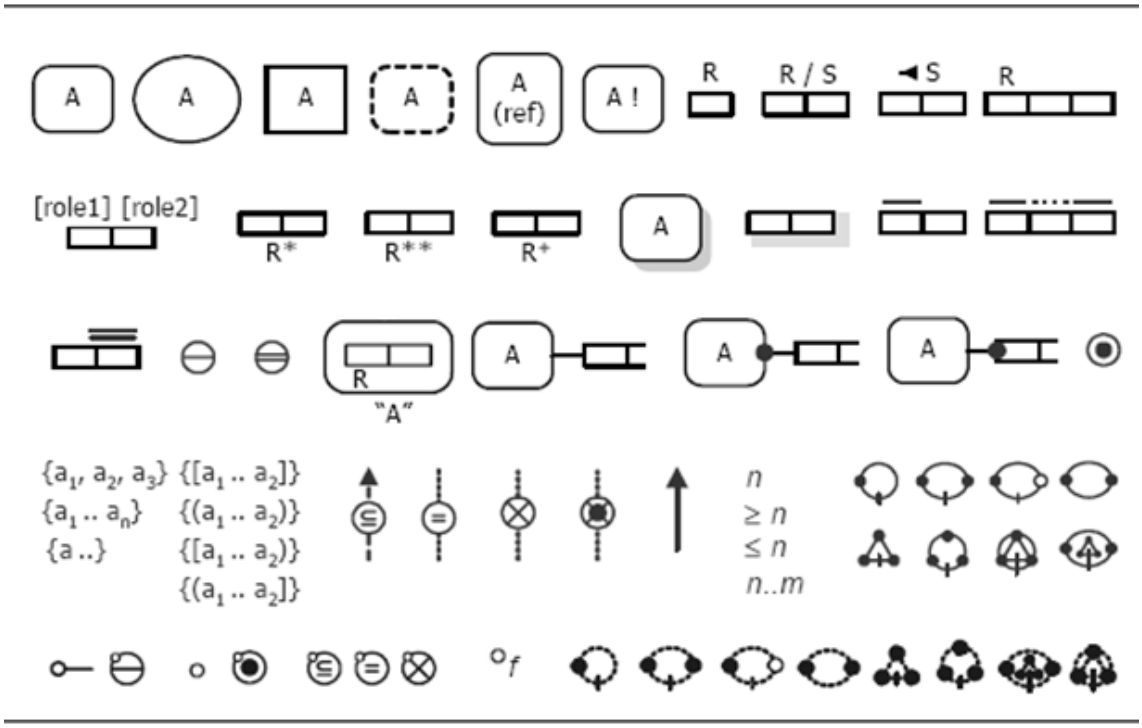
Rumbaugh, J., Jacobson, I. and Booch, G. 1999. *The Unified Modeling Language Reference Manual.* Massachusetts, USA : Addison Wesley Longman, Inc, 1999.

Silva, Alberto M. R. and Videira, Carlos A. E. 2001. *UML, Metodologias e Ferramentas CASE.* [ed.] 1ª. Porto : Centro Atlântico, 2001.

SPARX. 2008. XML Schema Generation. *SPARX Systems.* [Online] 2008. http://www.sparxsystems.com.au/resources/xml_schema_generation.html.

ANEXOS

Anexo A: Os Principais símbolos de Modelação em ORM2



Principais símbolos de Modelação ORM2 (Halpin T. , 2005)

Anexo B: Algoritmo de Transformação ORM-para-UML

B.1 – Algoritmo de transformação para a configuração de fato Individual

Algorithm 1: ORM-Unary-to-UML-class-attribute(ftx)
BEGIN
*Get the entity type that plays the role in the unary fact type **ftx** {A}*
*Get the reference scheme for the entity type **A** {anr}.*
*Get the predicate of the unary fact type **ftx** {xy}.*
*IF no object class exists with name **A***
*THEN Create a UML object class with name **A**.*
*Create the first class attribute for the object class **A** having name **anr**.*
*Add the qualification **{P}** to the attribute **anr**.*
ENDIF
*Create a class attribute for object class **A** that has the name **xy**.*
*Add the attribute type **boolean** to the class attribute **xy**.*
*IF entity type **A** does not play a mandatory role in fact type **ftx***
*THEN assign an attribute multiplicity of [0..1] to the attribute **xy***
ENDIF
END

Algorithm 2: ORM-Unary-to-UML-sub type(ftx)
BEGIN
*Get the entity type that plays the role in the unary fact type **ftx** {A}*
*Get the reference scheme for the entity type **A** {anr}.*
Get the predicate of the unary fact type {xy}.
*IF no object class exists with name **A***
THEN
*Create a UML object class with name **A**.*
*Create the first class attribute for the object class **A** having name **anr**.*
*Add the qualification **{P}** to the attribute **anr**.*
ENDIF
*Create a sub type of object class **A** that has the name **xy**.*
END

Algorithm 3: ORM-Binary-to-UML-attribute(ftx,A)[6]
BEGIN
*IF the object class with name **A** does NOT exist*
*THEN Create a UML object class with name **A***
*Get the reference scheme for the entity type **A** {anr}*
*Create the first class attribute for the object class having name **anr***
*Add the qualification **{P}** to the attribute **anr***
ENDIF
*Add an attribute to object class **A** which has as name the appropriate predicate of fact type **ftx** {ax}*
IF the object type in the other role {B} is a value type
*THEN assign this value type as the data (attribute) type of attribute **ax**.*
*ELSE append the name of object type **B** to the name of attribute **ax**.*
*IF object type **B** has an abbreviated reference scheme*
THEN get the reference scheme for object type B {bnr}
*Assign **bnr** as the data (attribute) type of attribute **ax**.*
ENDIF
ENDIF

*IF there exists a uniqueness constraint defined on the role of fact type played by object type **A***
*THEN assign a maximum attribute multiplicity to **ax** of **[..1]***
*ELSE assign a maximum attribute multiplicity to **ax** of **[..*]***
ENDIF
*IF there exists a uniqueness constraint defined on the role played by object type **B***
*THEN IF object type **A** plays a(n) implied mandatory role in **ftx** AND a uniqueness constraint defined on the role of fact type **ftx** played by object type A exists*
*THEN place {U} behind the attribute **ax***
*ELSE assign the following textual constraint to object class **A**: {each element of {ax} refers to at most one A}.*
ENDIF
ENDIF
IF there exists a uniqueness constraint defined on both roles
*THEN assign a maximum attribute multiplicity to **ax** of **[..*]***
ENDIF
*IF (entity type **A** is independent) OR (**A** does not play a(n) implied mandatory role in **ftx** in the global schema)*
*THEN add a lower multiplicity of **[0..** to attribute **ax***
*ELSE add a lower multiplicity of **[1..** to attribute ax*
ENDIF
IF the final attribute multiplicity for attribute ax = [1..1]
THEN leave out the multiplicity because it is the UML default
ENDIF
END

Algorithm 4: ORM-Binary-to-UML-association(ftx)
BEGIN
WHILE still roles in ftx
DO take next role {rx}
get the object type that plays role ***rx*** *{ex}*
IF there does not exist a classifier for the object type ***ex***
THEN
IF ***ex*** *is an entity type*
THEN create a UML object class with name ***ex***
Get the reference scheme for the entity type ***ex*** *{enr}*
Create the first class attribute for the object class ***ex*** *having*
name ***enr***
Add the qualification ***{P}*** *to the attribute* ***enr***
ELSE Create a UML data type with name ***ex***
ENDIF
ENDIF
ENDWHILE
Create a binary association (line) {bx} between the object class(es)[7] *and or data type that play the roles in the binary fact type* ***ftx.***
Give the association ***bx*** *a name that is equal to the appropriate predicate of* ***ftx***
IF role names exist in the ORM fact type ***ftx***
THEN Match the role names in fact type ***ftx*** *with the association end names in the association* ***bx.***
ENDIF
IF there is a uniqueness constraint defined on ***ftx*** *that covers both roles*
THEN assign an upper multiplicity of ***..*]*** *to both association ends in the association* ***bx.***
ENDIF
WHILE still roles in ftx

DO take next role {nrx}
IF there is a uniqueness constraint defined exclusively on role nrx
THEN assign an upper multiplicity of ***..1]*** *to the opposite association end in the corresponding association* ***bx.***
ELSE assign an upper multiplicity of ***..*]*** *to the opposite association end in the corresponding association bx.*
ENDIF
IF (a mandatory role is defined on role) ***nrx*** *OR (****nrx*** *is the only role that is played by the object type and the object type is NOT independent}*
THEN assign a lower multiplicity of ***[1..]*** *to the opposite association end in the corresponding association* ***bx.***
ELSE assign a lower multiplicity of ***[0..]*** *to the opposite association end in the corresponding association* ***bx.***
ENDIF
IF the final association end multiplicity for the opposite association end = [1..1]
THEN replace the association end multiplicity for the opposite association end by ***1***
ENDIF
IF the final association end multiplicity for the opposite association end = [0..]*
*THEN replace the association end multiplicity for the opposite association end by **
ENDIF
ENDWHILE
END

Algorithm 5: ORM-Nary-to-UML-association(ftx)
BEGIN
WHILE still roles in ftx
DO take next role {rx}
*Get the ORM object type that plays role **rx** {ex}*
*IF there does not exist a classifier for the object type **ex***
THEN
*IF **ex** is an entity type*
*THEN create a UML object class with name **ex***
*Get the reference scheme for the entity type **ex** {enr}*
*Create the first class attribute for the object class **ex** having name **enr***
*Add the qualification **{P}** to the attribute **enr***
*ELSE Create a UML data type with name **ex***
ENDIF
ENDIF
ENDWHILE
*Create a N-ary association {bx} between the object class(es) that play the roles in the N-ary fact type **ftx** by connecting the object classes one time for every role in **ftx** to the diamond in **bx**.*
*Give the association **bx** a name that is equal to the predicate of **ftx***
*IF role names exist in the ORM fact type **ftx***
*THEN Match the role names in fact type **ftx** with the association end names in the association **bx**.*
ENDIF
IF there is a uniqueness constraint defined on ftx that covers all roles
*THEN assign an upper multiplicity of **..*]** to all association ends in the association **bx**.*
ENDIF
*WHILE still roles in **ftx***
DO take next role {nrx}
IF there is a uniqueness constraint defined on the other [N-1] roles
*THEN assign an upper multiplicity of **..1]** to the corresponding association end **anrx** in the association **bx**.*

*ELSE assign an upper multiplicity of **..*]** to the corresponding association end **anrx** in the association **bx**.*
ENDIF
*IF a mandatory role is defined on role **nrx** OR role **nrx** is the only role that is played by the object type*
*THEN connect a textual constraint: {'Each **ex** must play this role'} to the line in the class diagram that connects the role to the association diamond*
*Assign a lower multiplicity of **[0..]** to the corresponding association end **{anrx}** in the association **bx**.*
*ELSE assign a lower multiplicity of **[0..]** to the corresponding association end **{anrx}** in the association **bx**.*
ENDIF
*IF the final association end multiplicity for this **anrx** = [1..1]*
THEN replace the association end multiplicity for this role by 1
ENDIF
*IF the final association end multiplicity for this **anrx**= [0..*]*
*THEN replace the association end multiplicity for this role by **
ENDIF
ENDWHILE
END

Algorithm 6: ORM nested object type-onto-UML-association class (nt)
BEGIN
*Get the defining fact type for the nested object type **nt** {fnt}*
*Get the UML association for **fnt** {as}*
*Add an object class with name **nt***
*Add a hyphenated line from the line or diamond to object class **nt***
*Remove the name from the association **as***
END

Algorithm 7: co-referenced-object-type-onto-UML-attributes (D)
BEGIN
*Get the co-referenced object type **D***
IF no UML object class D exists
*THEN Create a UML object class with name **D***
ENDIF
WHILE still fact types left on which an external uniqueness (co-reference) constraint is defined
DO take next fact type {nfx}
IF object type that plays the role on which the co-reference constraint is defined is an entity type
THEN get this entity type {ex}
*get the abbreviated reference scheme for **ex** {anx}*
*add an attribute to the object class **D** with attribute name of the fact type predicate combined with the name of the entity type **ex** and as attribute type the name of the value type **anx** {enx}*
ELSE get the value type {anx}
*add an attribute to the qualifier with attribute the name of the fact type predicate and as attribute type the name of the value type **anx** {enx}*
ENDIF
*add the qualification {P} to the attribute **enx**.*
ENDWHILE
END

Algorithm 8: ORM subtype-onto-UML-subtype (SUBT)
BEGIN
*Get the ORM super type of **SUBT** {SUPERT}*
*IF there does not exist an object class for **SUPERT***
*THEN create a UML object class with name **SUPERT***
*IF an abbreviated reference scheme exists for **SUPERT***
*THEN Get the reference scheme for the entity type **SUPERT** {snr}*
*Create the first class attribute for the object class having name **snr***
*Add the qualification {P} to the attribute **snr***
ENDIF
ENDIF
*IF there does not exist a UML object class for **SUBT***
*THEN Create a UML object class with name **SUBT***
ENDIF
*Create the UML subtype by drawing a subtype triangle under object class **SUPERT** and a connecting line between this super type triangle and **SUBT***
*IF a subtype defining rules exists for **SUBT***
THEN
For each entity type (sub or super type) in the subtype defining ORM rule draw a hyphenated line to the corresponding object class or attributes.
Connect all hyphenated lines to the sub typing defining rule in brackets. Copy the subtype-defining rule as a textual constraint attached to the hyphenated lines
ENDIF
END

B.2 – Transformation algorithms for complete ORM conceptual schema to complete UML class

Algorithm 9a: ORM-onto-UML-initial class diagram (ORM conceptual schema, unary encoding preference)
BEGIN
Get all entity types, nested object types and functional value types[8] in the ORM conceptual schema {entity types left}
Get all fact types in the ORM conceptual schema {fact types left}
WHILE still entity types or nested object types in ***entity types left***
DO get next entity type or nested object type ***{ex}***
*IF [(**ex** is an independent entity type OR **ex** plays two or more roles of which at least one role is mandatory)AND (**ex** has an abbreviated reference scheme)]*
THEN get value type from abbreviated reference scheme ***{vnrx}***
*model **ex** as an explicit object class **{ocx}** and model the value type **vnrx** of the abbreviated reference scheme as the attribute name of the identifier attribute **{ocanr}***
*add the qualification {P} after attribute **ocanr***
*Remove **ex** from **entity types left***
ELSE IF ex= nested object type
*THEN get constituting fact type for **ex** {ftx}*
*IF arity of constituting fact type **ftx** >2*
*THEN algorithm 5 (**ftx, lower multiplicity mode**)*
*ELSE algorithm 4 (**ftx**)*
ENDIF
*algorithm6(**ex**)*
*remove **ex** from **entity types left***
*remove **ftx** from **fact types left***
*ELSE IF **ex** is a super type AND **ex** has an abbreviated reference scheme*
*THEN model **ex** as an explicit object class **{ocx}** and model the value type **vnrx** of the abbreviated reference scheme as the attribute name of the identifier attribute **{ocanr}***
*add the qualification {P} after the attribute **ocanr***
*remove **ex** from **entity types left***
*ELSE IF **ex** is a subtype*
*THEN algorithm8(**ex**)*
*remove **ex** from **entity types left***
ENDIF
ENDIF
ENDIF
ENDIF
*IF (**ex** plays a role in at least one ternary or higher order fact type {FTY} AND **ex** is not a nested object type) OR*
*(**ex** plays two roles in at least one binary fact type{FTY}) OR*
*(**ex** plays a role in at least one unary fact type {FTY})*
THEN add the fact types in FTY that are not defining fact types for a nested object type to the set of fact types to be created in this sub transformation ***{naryset}***
ENDIF
ENDWHILE
WHILE still fact types in ***naryset***
DO get next fact type ***{ftx}***
IF arity of ftx>2
THEN

```
    Algorithm 5(ftx,lower multiplicity mode)
ENDIF
IF arity of ftx=1
THEN IF unary encoding preference=subtype
        THEN algorithm 2(ftx)
        ELSE algorithm 1(ftx)
       ENDIF
ENDIF
IF arity of ftx=2
THEN Algorithm 4(ftx)
ENDIF
Remove fact type ftx  from fact types left
ENDWHILE
WHILE still value types left
DO get next value type{vx}
    IF a value constraint is defined on the value type vx
    THEN create an enumeration vx for that value type and list the values in the value
          type as enumeration constants of the enumeration vx
    ENDIF
ENDWHILE
END
```

```
Algorithm 9b: UML initial class diagram-onto-UML-second class diagram (ORM conceptual schema,
UML class diagram, entity types left, fact types left)
BEGIN
WHILE still object types in entity types left
DO get next entity  type {ex}
IF [(ex plays one role in one binary fact type) AND (ex is not independent) AND (ex has an abbreviated
        reference scheme OR ex is a functional value type)]AND ( the other role in that fact type in which
        ex plays the role is played by an entity type that is already encoded as an object class that is not
        an association class).
THEN    Get the other entity type {oex}
         get fact type predicate {fty}
        IF fty is in fact types left
        THEN algorithm3(fty,oex)
                remove fty from fact types left
        ENDIF
        remove ex from entity types left
ELSE IF ex is a co-referenced entity type
      THEN IF at least one of the entity types in the functional roles of the constituting
           fact types in the co-reference is modeled as an explicit object class
           THEN apply the nest/co-reference theorem.
                Get the defining fact type of the nested object type {ftx}
                  IF arity of ftx>2
                  THEN   Algorithm 5(ftx,lower multiplicity mode)
                  ELSE   Algorithm 4(ftx)
                  ENDIF
                  Algorithm 6(ex)
                  Remove ex from entity types left
        ELSE    Algorithm 7 (ex)
                Remove ex from entity types left
```

```
        ENDIF
    ENDIF
ENDIF
ENDWHILE
END
```

Algorithm 9c: UML second class diagram –onto-UML third class diagram (ORM conceptual schema, UML class diagram, entity types left, fact types left)

```
BEGIN
WHILE  still object types in entity types left
        DO get next object type {ex}
        IF ex plays one role each  in only binary fact type(s) AND no mandatory role is defined on those
        roles AND at least one of the  other entity types in the  binary fact type(s)  are encoded as object
        classes
        THEN get name from the abbreviated reference scheme for ex {vnx}
                Get the binary fact types in which ex plays a role {fcx}
         WHILE still fact types in fcx
                Get next fact type {fxy}
                DO      get the object type in the other role {ocx}
                        Get the fact type predicate of  fxy
                IF (ocx is a nested object type AND the configuration of fxy=2.4, 2.6 or 2.8) OR (ocx is
                an entity type AND the configuration of fxy=2.1, 2.2, 2.4, 2.5, 2.6, 2.8 or 2.9)
                THEN Algorithm 3 (fxy, ocx)
                      Remove fxy from fact types left
                ELSE IF there does not exist a classifier for the object type ex
                       THEN

                          IF ex is an entity type
                          THEN create a UML object class with name ex
                         Get the reference scheme for the entity type ex {enr}
                        Create the first class attribute for the object class ex having
                          name enr
                          Add the qualification {P} to the attribute enr
                          ELSE Create a UML data type with name ex
                          ENDIF
                     ENDIF
              ENDIF
         ENDWHILE
      ENDIF
ENDWHILE
END
```

Algorithm 9d: UML third class diagram-onto-UML-fourth class diagram (ORM conceptual schema, UML class diagram, entity types left, fact types left)
BEGIN
Get the remaining object types that are no value types from ***entity types left*** *that are not yet encoded as an object class {REST}*
WHILE still entity types in REST
DO take next object type {ex}
Calculate the number of roles in ***fact types left*** *in which* ***ex*** *is involved {nex} AND in which the other object type is not yet encoded as an object class or association class*
IF ***nex****=0 THEN remove* ***ex*** *from* ***entity types left*** *ENDIF*
ENDWHILE
Rank the entity types in REST according to the values for nex. Make a list with the entity type that has the highest next on top and so forth until the entity type with the lowest nex is on the bottom of the list {LIST}
WHILE still entity types in LIST
DO take next highest entity type in LIST (>0) {nel}
WHILE still binary fact types in which ***nel*** *is involved*
DO take next fact type ein which ***nel*** *is involved from* ***fact types left****{fnel}*
Get the other object type in ***fnel*** *{oel}*
Get the fact type predicate of ***fnel*** *{fny}*
*Algorithm 3 (****fny, nel****)*
Remove ***fny*** *from* ***fact type list***
Subtract 1 from the value for ***oel*** *in the list*
ENDWHILE
Remove ***nel*** *from LIST and from* ***entity types left***
ENDWHILE
END

Algorithm 9e: UML fourth class diagram-onto-Final UML model (ORM conceptual schema, UML class diagram, unary encoding, fact types left)
BEGIN
WHILE still fact types in ***fact types left***
*DO take next fact type {****ftx****}*
IF arity ***ftx****=2*
THEN
Algorithm 4 (ftx)
Remove ***ftx*** *from* ***fact types left***
ENDIF
ENDWHILE
Get all entity types/nested object types in the ORM model {OBTYP}
WHILE still entity types/nested object types in ***OBTYP***
*DO take next entity type/nested object type {****eox****}*
*IF[(****eox*** *is encoded as an object class and is NOT independent) AND (****eox*** *does not play a mandatory role in the ORM conceptual schema) AND(eox plays at least two roles)] OR [On a subset of the roles that eox plays a disjunctive mandatory role constraint is defined]*
THEN assign the following textual constraint to the object class eox and all associations and attributes that are played by ***eox*** *or on which the disjunctive mandatory role constraint is defined: {every instance of object class eox should participate in at least one of the associations} OR {eox.attr1 is not null or ….or eox.attrN is not null}*[9]
ENDIF
ENDWHILE
END

Anexo C: Processo de Desenho do Esquema Conceptual (CSDP)

(Modelação de processos em ORM)

C.1 Passo 1 – Fatos elementares

- Escola EB 2,3 Dr. Jesus Neves Júnior tem Morada da Escola na Av …
- Escola EB 2,3 Dr. Jesus Neves Júnior tem Telefone da Escola nº 289 …
- Escola EB 2,3 Dr. Jesus Neves Júnior tem Fax da Escola nº 289 …
- Ambrósio Alves tem sobre Nome Soares.
- Ambrósio Alves Soares tem Data de Nascimento 27/02/1978.
- Ambrósio Alves Soares é do Gébero 'M'.
- Ambrósio Alves Soares nasceu em País.Moçambique
- Ambrósio Alves Soares ocupa Sala Sala SI1
- Sala SI1 tem nome da Sala. Sala de Informática 1
- Sala SI1 esta no Edifício Edifício nº. 1
- Sala SI1 tem altura de Cumprimento 4
- Sala SI1 tem largura de Cumprimento 4
- Sala SI1 tem Área 4*4
- Ambrósio Alves Soares ocupa Sala. SI2
- Moçambique é o lugarNascimento de Ambrósio Alves Soares
- Ambrósio Alves Soares é chefiado por Paula
- Departamento de Informática contém Professor Ambrósio Alves Soares
- Professor Ambrósio Alves Soares é membro Departamento de Informática
- Departamento de Informática tem Código do Dep DI
- Ambrósio Alves Soares lecciona Disciplina de TICs
- Professor é contratado até DataContratação.
- Professor é efectivo desde DataContratação.
- Ambrósio Alves Soares tem nr 20982
- Ambrósio Alves Soares esta no 2º Ano.
- Anibal pertence a Turma TIC

C.2 Passo 2 – Desenhar os tipos de fatos

- Escola tem MoradaEscola.
- Escola tem TelefoneEscola.
- Escola tem FaxEscola.
- Pessoa tem sobreNome.
- Pessoa tem DataNascimento.
- Pessoa é do Género.
- Pessoa tem Morada.
- Professor nasceu em País.
- Professor tem nomeProfessor.
- Professor ocupa Sala.
- País é o lugarNascimento de Professor
- Professor é chefiado por Professor.
- Departamento contém Professor.
- Professor é membro Departamento
- Departamento tem CodigoDep.
- Professor lecciona Disciplina.
- Professor é contratado até DataContratação.
- Professor é efectivo desde DataContratação.
- Aluno tem nr.
- Aluno fez Inscrição.
- Inscrição em Data.
- Aluno esta Ano.
- Aluno pertence a Turma.
- Disciplina pertencem a Turma.
- Sala tem nomeSala.
- Sala tem altura de Cumprimento.
- Sala tem largura de Cumprimento.

C.3 Passo 3 – Identifica os tipos de fatos derivados

- Sala tem Área.
  Cada Sala tem **no máximo uma** Área.
  é possível que mais de uma Sala tem **a mesma** Área.

C.4 Passo 4 – Restrições de unicidade

- **Cada** Escola tem **uma** MoradaEscola.
- **Cada** Escola tem **um** TelefoneEscola.
- **Cada** Escola tem **um** FaxEscola.
- **Cada** Pessoa tem **um** sobreNome.
- **Cada** Pessoa tem **um** DataNascimento.
- **Cada** Pessoa é do **no máximo um** Género.
- **Cada** Pessoa tem **uma** Morada.
- **Cada** Professor nasceu em **num** País.
- **Cada** Professor tem **um** nomeProfessor.
- **Cada** Professor ocupa **uma** Sala.
- **Um** País é o lugarNascimento de **mais do que um** Professor
- **Cada** Departamento tem **um** CodigoDep.
- **Cada** Professor lecciona **uma** Disciplina.
- **Cada** Professor é contratado até **uma** DataContratação.
- **Cada** Professor é efectivo desde **uma** DataContratação.
- **Cada** Aluno tem **um** nr.
- **Cada** Aluno fez **uma** Inscrição.
- **Cada** Inscrição em **uma** Data.
- **Cada** Aluno esta **num** Ano.
- **Cada** Aluno pertence a **uma** Turma.
- **Cada** Disciplina pertencem a **uma** Turma.
- **Cada** Sala tem **um** nomeSala.
- **Cada** Sala tem altura de **um** Cumprimento.
- **Cada** Sala tem largura de **um** Cumprimento.

C.5 Passo 5 – Restrições de obrigatoriedade

- **Cada** Escola tem **exatamente uma** MoradaEscola.
- **Cada** Escola tem **exatamente uma** TelefoneEscola.
- **Cada** Pessoa tem **exatamente uma** sobreNome.
- **Cada** Pessoa tem **exatamente uma** DataNascimento.
- **Cada** Pessoa é do **exatamente um** Género.
- **Cada** Pessoa tem **exatamente uma** Morada.
- **Cada** Professor nasceu em **exatamente um** País.
- **Cada** Professor tem **exatamente uma** nomeProfessor.
- **Cada** Departamento tem **exatamente um** CodigoDep.
- **Cada** Professor lecciona **exatamente uma** Disciplina.
- **Cada** Aluno tem **exatamente um** nr.
- **Cada** Aluno fez **exatamente uma** Inscrição.
- **Cada** Inscrição em **exatamente numa** Data.
- **Cada** Aluno esta **exatamente num** Ano.
- **Cada** Aluno pertence a **exatamente uma** Turma.
- **Cada** Disciplina pertencem a **exatamente uma** Turma.

C.6 Passo 6 – Restrições de valor

- Pessoa é do Género.
  Cada Pessoa é do **exatamente um** Género.
  O valor possível do Código do género **é** 'M,F'.

C.7 Passo 7 – Restrições de Anel

- **Para cada** Pessoa, **no máximo, uma das seguintes se verifica:**
  Que uma Pessoa é **um** Aluno;
  Que uma Pessoa é **um** Professor.

- **Para cada** DataContratação, , **no máximo, uma das seguintes se verifica:**
  Que um Professor é efectivo desde **uma** DataContratação;
  Que um Professor é contratado até **uma** DataContratação.

- **Para cada** Sala **e** nomeProfessor,
  Ano máximo, um Professor ocupa **a** Sala **e**
  tem **esse** nomeProfessor.

- **Cada** $Professor_2$ **em "**$Professor_1$ é chefiado por $Professor_2$**" ocorre, no máximo** 2 **vezes**.

- **Se** $Professor_1$ é chefiado por $Professor_2$ **e** $Professor_2$ é chefiado por $Professor_3$
  então é impossível que $Professor_1$ é chefiado por $Professor_3$.

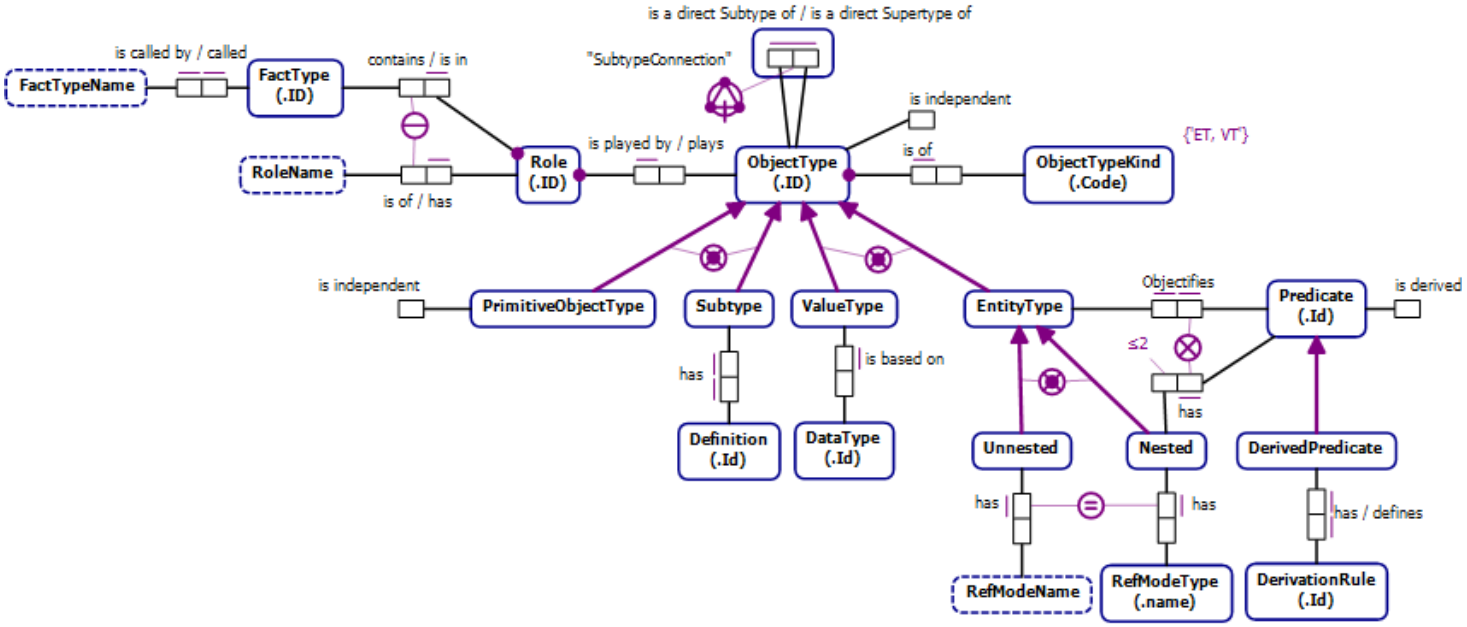
- **Para cada** Sala **e** nomeProfessor,
  no máximo um Professor ocupa **essa** Sala **e**
  tem **esse** nomeProfessor.

- **Para cada** Ano **e** Inscrição,
  essa combinação ocorre, no máximo 2 **vezes neste contexto**.

Anexo D: Metamodelo ORM.

Modelação em ORM (NORMA)

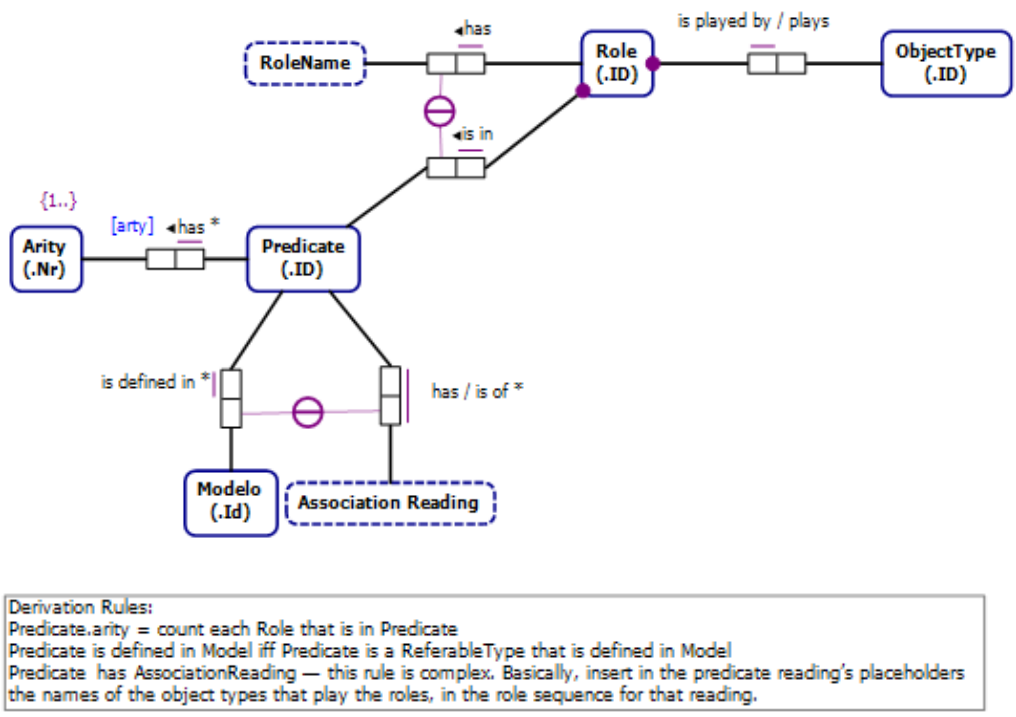
D.1 – Tipos principais



Subtype Definitions:
Each PrimitiveObjectType is an ObjectType that is a subtype of no ObjectType
Each Subtype is an ObjectType that is a subtype of an ObjectType
Each ValueType is an ObjectType that is of ObjectTypeKind 'VT'
Each EntityType is an ObjectType that is of ObjectTypeKind 'ET'
Each NestedEntityType is an EntityType that objectifies a Predicate
Each UnnestedEntityType is an EntityType that objectifies no Predicate
Each DerivedPredicate is a Predicate that is derived

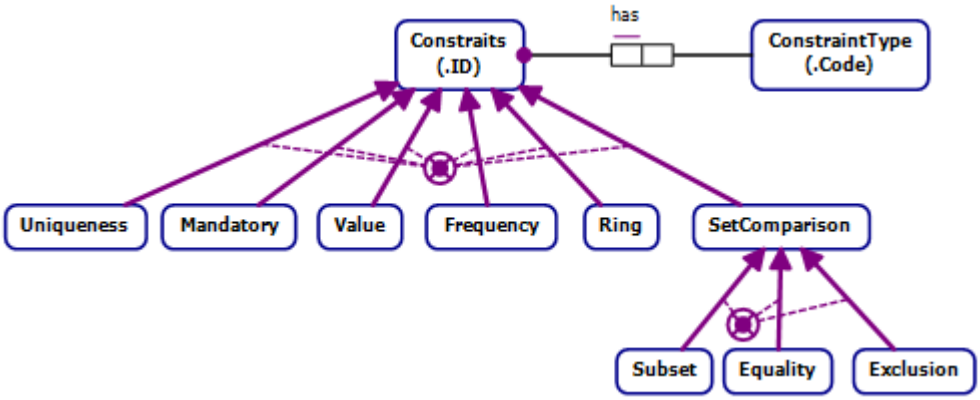
Principais tipos da ORM (Halpin & Cuyler, 2003)

D.2 – Relacionamentos



Relacionamentos da ORM (Halpin & Cuyler, 2003)

D.3 – Restrições



Restrições da ORM (Halpin & Cuyler, 2003)