MARCIO BELO RODRIGUES DA SILVA

Uma Avaliação da Abordagem MDA Através de um Estudo de Caso

Dissertação apresentada ao Curso de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito para obtenção do Grau de Mestre. Área de Concentração: Computação Distribuída e Paralela.

Orientador: Prof. ORLANDO GOMES LOQUES FILHO

Niterói
2005

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

S586 Silva, Marcio Belo Rodrigues da

Uma avaliação da abordagem MDA através de um estudo de caso / Marcio Belo Rodrigues da Silva. – Niterói, RJ : [s.n.], 2005.

134 f.

Orientador: Orlando Gomes Loques Filho.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal Fluminense, 2003.

1. Engenharia de software. 2. Arquitetura de computadores. I. Título.

CDD 005.1

MARCIO BELO RODRIGUES DA SILVA

Uma Avaliação da Abordagem MDA Através de um Estudo de Caso

Dissertação apresentada ao Curso de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito para obtenção do Grau de Mestre. Área de Concentração: Computação Distribuída e Paralela.

Aprovada em março de 2005.

BANCA EXAMINADORA

_____

Prof. Orlando Gomes Loques Filho, Ph.D. – Orientador
Universidade Federal Fluminense

_____

Prof. Maria Luiza d'Almeida Sanchez, D.Sc.
Universidade Federal Fluminense

_____

Prof. Alexandre Sztajnberg, D.Sc.
Instituto de Matemática, UERJ

Niterói
2005

SUMÁRIO

LISTA DE FIGURAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS

AS	Action Semantics
CASE	Computer Computer-Aided Software Engineering
CORBA	Common Object Request Broker
CWM	Common Warehouse Metamodel
DBC	Design By Contract
DTD	Document Type Definition
IDE	Integrated Development Environment
J2EE	Java 2 Enterprise Edition
JMI	Java Metadata Interface
MDA	Model-Driven Architecture
MOF	Meta-Object Facility
mSQL	meta-SQL
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query, Views and Transformations
RMI	Remote Method Invocation
SGBD	Sistema Gerenciador de Banco de Dados
SOAP	Simple Object Access Protocol

SQL	Structured Query Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange

RESUMO

A evolução das linguagens de programação nos mostra um cenário de constante elevação no nível de abstração, estimulada pela necessidade de aumento na produtividade para criação e manutenção de sistemas informatizados cada vez mais complexos. Entretanto, apesar dessa evolução, o processo de desenvolvimento de *software* ainda separa a criação de artefatos de modelagem dos artefatos de codificação.

A abordagem MDA (*Model-Driven Architecture*) busca integrar os esforços de modelagem com os de programação através da eliminação das barreiras existente entre eles. Em MDA, os modelos são o foco central para o desenvolvimento de sistemas, e não apenas meros transmissores de conhecimentos em papel de uma equipe para outra. Ferramentas automatizadas farão o trabalho de conversão dos modelos de especificação e arquitetura da aplicação em artefatos de código executável, na forma mais completa possível.

Neste trabalho, buscamos apresentar, através de um estudo de caso, uma primeira investigação da tecnologia no nosso grupo de trabalho, analisando os resultados obtidos dessa experiência e comparando com as expectativas traçadas pelos proponentes do MDA. Com isso, esperamos apresentar um cenário concreto do estado atual da tecnologia e apontar alguns aspectos que precisam ser melhor elaborados para atingir essas expectativas.

Desta forma, no intuito de promover uma análise mais aprofundada de todos os aspectos do MDA, escolhemos o modelo de persistência de dados do sistema exemplo como alvo principal para a aplicação completa de todas as práticas sugeridas pela tecnologia no seu estágio atual.

ABSTRACT

The evolution of programming languages shows a constant raising in the level of abstraction, stimulated by the need for higher productivity in the creation and maintenance of computer systems. In spite of this evolution, the development of software still makes the difference between model artifacts and the code ones.

The MDA (Model-Driven Architecture) approach seeks the integration of efforts from modeling and programming by eliminating the barriers between them. In MDA, models are central for the development of systems, not only transmitters of knowledge from analysts to programmers. Automated tools will do the work of transforming specification and architecture models to executable code artifacts, in a way as complete as possible.

In this work, we intend to show, through a case of study, one first investigation of the technology in our working group, analyzing the requirements gathered from this experience and comparing with expectations from the MDA proponents. Thus, we expect to show the real state of art of this technology and point out some aspects willing to be more elaborated to reach those expectations.

Thus, in order to promote a deep analysis of all the aspects of MDA, we pick up the persistence data model from the example system as a target to apply the practices dictated by the actual state of art of the technology.

1. INTRODUÇÃO

Comumente, o processo de desenvolvimento de sistemas tradicional rebaixa a importância dos modelos a meros transmissores do conhecimento sobre o quê e como um sistema deve ser desenvolvido. Nesse contexto, uma vez que os desenvolvedores tenham adquirido e utilizado o conhecimento que esses modelos oferecem, eles acabam perdendo sua importância e conseqüentemente se tornam obsoletos em pouco tempo, às vezes antes mesmo que o processo de desenvolvimento completo do sistema tenha sido finalizado.

Esse vazio que existe entre o projeto de *software* tradicional e o *software* executável fomenta a idéia de que o esforço em criar um projeto completo atrasava a entrega do produto final (*software*), e que, portanto, não se deveria perder tempo excessivo com "desenhos". Em [Fra2003] são citados fatores que colaboram para a falta de preocupação com questões de qualidade na indústria de *software*, tais como urgência e visão limitada em relação às implicações dos possíveis prejuízos causados ao negócio por falhas ou omissões.

A proposta do MDA (*Model-Driven Architecture*) é tornar a modelagem do sistema central para o processo de criação do *software* final. Desta maneira, a preocupação dos desenvolvedores deverá ser a criação de modelos do sistema, através da utilização de linguagens com alto nível de abstração e, uma vez que esses modelos tenham sido concebidos, de acordo com os requisitos de negócio para o qual o *software* modelado se

destina, uma ou mais transformações serão realizadas nesses modelos que irão produzir código executável.

A criação de um modelo independente de plataforma preconizada pelo MDA libera o desenvolvedor de restrições tecnológicas geralmente impostas em processos de desenvolvimento de *software*, deixando para ferramentas automatizadas boa parte do trabalho maçante de gerar os artefatos para uma determinada plataforma de execução.

1.1. Objetivos

Os objetivos deste trabalho são:

- Apresentar uma visão da recente abordagem MDA, detalhando os aspectos e elementos mais importantes da proposta;

- Elaborar um estudo de caso que mostre a geração de um sistema informatizado típico, usando a abordagem MDA. Para esse exercício prático, foi selecionado um sistema voltado para controle de algumas atividades necessárias em um ambiente acadêmico. Como resultado, criamos um modelo independente de plataforma e aplicamos as transformações necessárias para gerar código de implementação do sistema doravante denominado Acadêmico;

- Elaborar um modelo de transformação para código de persistência relacional, comparando com abordagens semelhantes e identificando os benefícios das escolhas feitas neste trabalho;

- Analisar o resultados obtidos com a experiência do sistema Acadêmico na investigação da abordagem MDA, consolidando os conhecimentos adquiri-

dos como uma primeira investigação do nosso grupo de trabalho sobre o assunto.

1.2. Organização

Visando cumprir os objetivos traçados neste trabalho, o capítulo 2 inicia com uma análise do cenário atual de desenvolvimento de *software* e a apresentação dos fatores que motivaram o surgimento do MDA. Na seqüência, a proposta MDA propriamente dita é detalhada e os principais padrões e tecnologias envolvidas são estudados.

No capítulo 3 é apresentado o sistema Acadêmico, que será integralmente implementado. O capítulo inicia pela descrição dos aspectos funcionais necessários e decisões de arquitetura feitas para o projeto, seguida da especificação, segundo os preceitos do MDA, do modelo independente de plataforma. De posse dessas especificações, o capítulo aprofunda as transformações necessárias para geração do código executável do sistema acadêmico.

No capítulo 4, a transformação para o modelo de banco de dados relacional será discutida com maior nível de aprofundamento, visando fundamentar as escolhas feitas neste trabalho e apresentando outras alternativas existentes para esse mesmo cenário.

No capítulo 5 é feita uma análise do processo de aplicação do MDA no sistema exemplo buscando comparar os resultados alcançados com os objetivos esperados, conforme as expectativas geradas pelo estado atual da tecnologia MDA.

Finalmente, no capítulo 6, é feita a conclusão deste trabalho traçando as possíveis perspectivas da abordagem MDA e relacionando os pontos que precisam ser melhor elaborados para a concretização das metas esperadas.

2. MDA (MODEL-DRIVEN ARCHITECTURE)

2.1. Introdução

2.1.1. O processo de desenvolvimento de *software*

A figura 2.1, extraída de [KWB2003], mostra o ciclo de desenvolvimento de *software* tradicional.

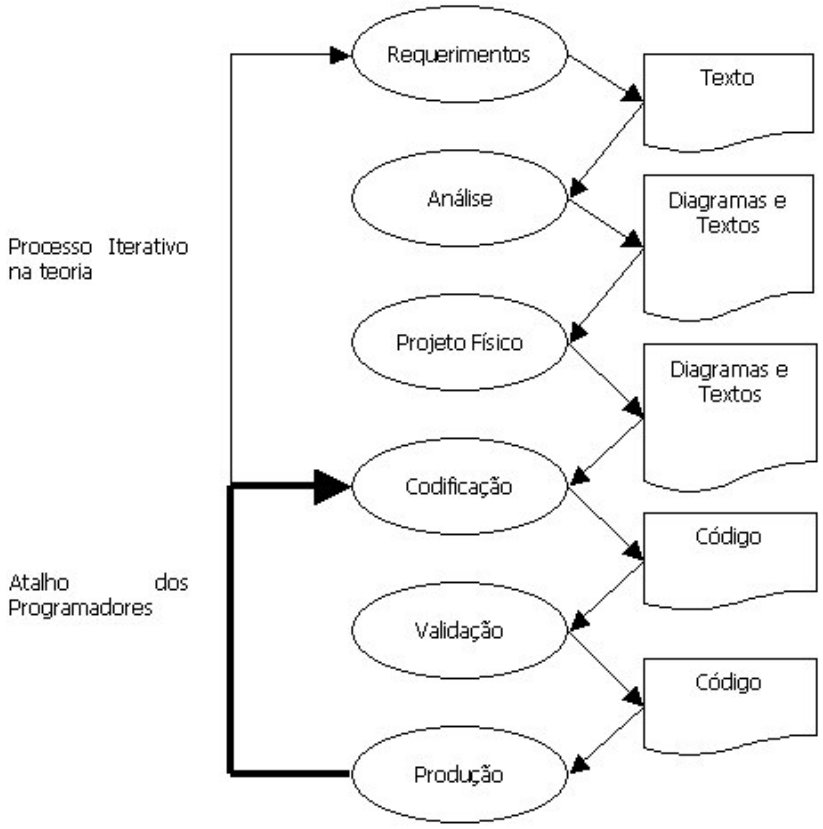


Figura 2.1 Ciclo de Desenvolvimento de *Software* Tradicional

Esse processo de desenvolvimento também é conhecido como processo *waterfall*. Algumas de suas características importantes são ressaltadas e discutidas a seguir:

- Produtividade

A produtividade do processo de desenvolvimento tradicional (seja na sua versão incremental ou interativa) fica seriamente comprometida com a "quebra" existente entre os modelos de alto nível do *software* e os modelos de implementação, conforme sustentado em [KWB2003].

Os modelos de alto nível – modelos de caso de uso, diagramas de classe, diagrama de interação, diagrama de atividades, etc. – acabam sendo enxergados como documentos que apenas transmitem do projetista ao desenvolvedor o conhecimento necessário para que o último possa gerar o que é realmente esperado como produto final do processo: o código executável do sistema. Assim que a fase de codificação é iniciada, os modelos de alto nível perdem rapidamente seu valor, uma vez que todo o esforço do desenvolvimento se volta para a codificação. Embora não fosse esse o objetivo na modelagem tradicional, a prática acabou por produzir essa realidade: todas as mudanças sendo feitas em nível de código, tornando os modelos de alto nível desatualizados e inúteis.

O problema da não atualização dos modelos de alto nível passa a ser sentido geralmente algum tempo depois da entrega do produto inicial, quando a equipe que desenvolveu o sistema é desfeita e novas alterações são demandadas. Outras pessoas que devem manter o sistema – corrigir *bugs*, incluir novas funcionalidades, etc. – acabam tendo como fonte para adquirir conhecimento sobre o sistema apenas o código-fonte, que pode ser composto por centenas ou milhares de linhas de código.

- Portabilidade

Uma das características peculiares da indústria de *software* é a necessidade de acompanhar as novas tecnologias que surgem, motivada por fatores tais como: (i) o atendimento da demanda dos clientes por novos recursos, (ii) a resolução dos problemas mais facilmente, ou (iii) a redução do alto custo associado em manter um produto de *software* desatualizado.

O processo de desenvolvimento tradicional, na sua concepção, gera como produto um sistema específico para uma determinada tecnologia. A migração do produto geralmente implica no desenvolvimento de um novo sistema, perdendo boa parte do que foi desenvolvido anteriormente.

- Interoperabilidade

A grande maioria dos sistemas depende da interação com outros sistemas, que geralmente são desenvolvidos com tecnologias, em plataformas diferentes, para a realização de suas funções.

Além disso, a tendência atual é a da componentização de objetos que interagem entre si, facilitando a atualização seletiva de partes do sistema. Esse processo de componentização amplia ainda mais a diversidade de tecnologias empregadas, visto que tipicamente para cada tipo de componente necessário encontraremos uma tecnologia mais adequada para seu propósito. Essa tendência é totalmente oposta ao que acontecia nos antigos sistemas, construídos inteiramente sob uma mesma plataforma e tecnologia.

- Manutenção e Documentação

A atividade de manutenção é amplamente dificultada pela não atualização dos modelos de alto nível do sistema. Como vimos anteriormente, os desenvolvedores tendem ao sentimento de que o tempo utilizado na construção desses produtos é uma tarefa secundária, tirando o foco do que seria a tarefa principal: codificar.

A ausência de documentação de alto nível, entretanto, poderá gerar grave impacto quando a atualização do sistema se fizer necessária.

2.1.2. As linguagens de programação

Além dos fatores citados sobre o processo tradicional de desenvolvimento de sistemas, a própria evolução da construção de códigos de programação nos mostra a constante tendência de aumento do nível de abstração [MB2003].

No começo, por volta da década de 50, os programas eram construídos diretamente em linguagem de máquina, com código binário. Logo após, em meados da década de 60, apareceram as linguagens de montagem que, embora tivessem uma correlação precisa com as instruções de baixo nível da máquina, representaram um primeiro passo para abstrair detalhes de uma determinada arquitetura de hardware, aproximando a programação da linguagem humana. Com a introdução das linguagens de montagem, apareceram as primeiras ferramentas de *software* voltadas ao auxílio do desenvolvimento de *software* - os montadores – que transformavam os mnemônicos em instruções binárias executáveis.

Algum tempo depois, foi observada uma mudança significativa na evolução da programação de *software*: o aparecimento das linguagens procedurais. Durante o perío-

do de 1965 a 1985, o mundo da programação viveu uma grande revolução. As linguagens procedurais introduziram instruções de alto nível, ainda mais próximas da linguagem humana. Como em qualquer mudança, houve resistências. Alguns alegavam que os compiladores, que transformavam esses programas de alto nível para a linguagem de máquina, dificilmente geravam códigos eficientes. Muitos programadores alegavam serem capazes de produzir, codificando diretamente na linguagem da máquina, melhor código do que aquele gerado pelos primeiros compiladores. Isso era verdade até o aprimoramento dos compiladores, gerando códigos cada vez mais eficientes. Essa mudança de paradigma nos mostrou pela primeira vez a vantagem do aumento do nível de abstração: a portabilidade. Códigos escritos em linguagens como FORTRAN e COBOL podiam ser, com relativa facilidade, compilados para outras plataformas além daquelas para os quais foram originalmente produzidos.

A partir do ano de 1985, surgiram as linguagens de programação orientadas a objetos. A programação orientada a objetos introduziu conceitos que permitiram uma melhor estruturação do código de programação, e, como de praxe, aproximou ainda mais a linguagem para programação da linguagem humana; conceitos como classes de objetos, herança, encapsulamento, entre outros, davam ao programador uma visão mais próxima à forma como os humanos percebiam o mundo do que as linguagens usadas até então. Esse paradigma para programação perdura até hoje e ainda não é amplamente difundido e aceito. Da mesma forma como o ocorrido na transição das linguagens de montagem para as procedurais, a transição para as orientadas a objetos encontrou barreiras de aceitação.

2.1.3. Programando por Modelos

Não é nova a idéia de permitir que um desenvolvedor crie modelos que descrevam a solução de um problema computacional de forma completa. Os trabalhos propostos em [MB2003,LS2004] representam iniciativas concretas nesse sentido. O grande

desafio é tornar um modelo completo o suficiente para que possa ser a fonte para a geração de um produto executável, sem a necessidade de ajustes em nível de código.

A utilização de modelos para a criação de artefatos de *software* estáticos já é amplamente utilizada, como podemos observar nas ferramentas CASE de modelagem de dados. Através delas, podemos modelar usando um nível maior de abstração e gerar, para um determinado produto de banco de dados, todo o código necessário para definir a estrutura e restrições de um banco de dados.

O desafio maior ao desenvolver modelos "executáveis" é torná-lo completo o suficiente para que toda a parte dinâmica do sistema possa ser representada, usando um nível apropriado de abstração. É para essa direção que os esforços do MDA se voltam: a criação de um modelo independente de plataforma usando um alto nível de abstração.

A proposta idealizada pelo MDA é que os modelos produzidos sejam possíveis de serem entendidos não só pelos projetistas do sistema, como ocorre no processo tradicional, mas também por computadores. Dessa forma, os modelos serão os elementos primordiais na elaboração do sistema e, quando submetidos a transformadores, gerarão - total ou parcialmente - o código executável.

Em MDA, a atividade de modelagem é uma atividade de programação. Sendo assim, os papéis de modelador e do programador não são distintos como no processo tradicional de desenvolvimento. A modelagem no contexto do MDA é considerada uma atividade produtiva dentro do processo de construção do *software*.

Dada a complexidade dos sistemas atuais, é inevitável a necessidade de usar no desenvolvimento de um sistema mais de uma ferramenta ou tecnologia. No processo tradicional de desenvolvimento de *software*, as tentativas de limitar a complexidade do sistema passavam pela proposta de adotar uma única linguagem para descrever todo o sistema. Um erro comum entre os iniciantes do MDA é acreditar que ela é mais uma

tentativa nesse sentido, quando, na verdade, não existe essa pretensão. MDA aceita a inevitável realidade de que serão necessárias várias linguagens, cada qual atuando num ponto específico para modelar um determinado aspecto do sistema. A intenção é integrar essas várias linguagens em nível de especificação, mais especificamente através dos seus respectivos meta-modelos.

Um problema fundamental apresentado na maioria das ferramentas, a citar a proposta *Executable UML* [MB2003], é a falta de utilização de padrões consagrados e amplamente aceitos para a especificação do modelo. Nesse sentido, o MDA é um esforço visando estabelecer esses padrões - independentes de qualquer fornecedor de ferramenta de modelagem – de forma que qualquer ferramenta compatível com tais padrões possa ser utilizada no desenvolvimento de um sistema baseado em MDA.

2.1.4. Conclusão

A expectativa em torno do MDA se concentra em um novo passo na evolução do processo de desenvolvimento de *software*: a possibilidade de criar sistemas usando uma linguagem mais genérica, com alto grau de abstração em relação a qualquer restrição imposta por uma plataforma de desenvolvimento específica. A própria história da evolução da engenharia de *software* sustenta a idéia de que brevemente estaremos produzindo *software* num nível de abstração sem qualquer restrição tecnológica.

A proposta do MDA tem a pretensão de consolidar essa idéia através de artefatos genéricos que já são produzidos na maioria dos processos de desenvolvimento de *software*, porém muitas vezes relevados a um papel secundário: os modelos. O MDA eleva a importância dos modelos a um nível vital para o processo de criação do *software*.

2.2. MDA Framework

MDA é um *framework* para desenvolvimento de sistemas proposto pela OMG (*Object Management Group*) [Mda2004]. Ele tem como proposta melhorar as práticas de desenvolvimento de projetos de *software* e estender a vida útil deles, permitindo sua adaptabilidade às novas tecnologias que surgem.

Para atender seus objetivos, MDA emprega uma série de padrões – alguns ainda em fase de maturação – ilustrado pela figura 2.2 retirada de [Mda2004].

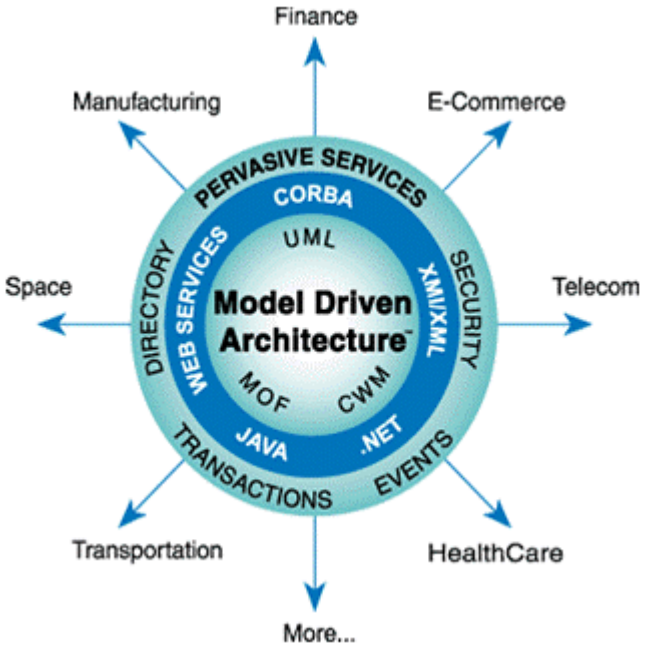


Figura 2.2 Padrões do MDA *Framework*

No núcleo da figura 2.2 estão os padrões principais que formam o *Framework* do MDA. Alguns desses padrões foram adotados pela OMG mesmo antes do padrão MDA ter sido proposto. É o caso do UML, o mais conhecido dos padrões OMG. A UML é uma linguagem notacional – que emprega elementos visuais e textuais - amplamente adotada na especificação de sistemas, principalmente naqueles que seguem práticas orienta-

das a objetos. Outro caso de padrão surgido antes do MDA é o padrão MOF. O MOF é uma linguagem padronizada para definição de linguagens de modelagem ou, como dito no contexto MDA, uma meta-linguagem. Já o CWM [Cwm2003] é um conjunto de meta-modelos padronizados pela OMG que representam modelos, em geral no nível da modelagem de dados, usualmente aplicados na especificação de sistemas. Esses padrões serão discutidos em mais detalhes neste trabalho.

No anel intermediário da figura 2.2 são mostradas plataformas recentes para implementação e especificação de sistemas, como *Web Services* e CORBA, essa última sendo um padrão mantido também pela OMG. Cabe ressaltar que as tecnologias citadas não visam limitar a aplicação do MDA apenas a essas plataformas ou especificações. Como veremos adiante sobre as características do MDA, a independência de plataforma e a adaptabilidade às novas tecnologias são pontos vitais para a abordagem. A separação dessas tecnologias do núcleo central do MDA visa justamente representar a independência de plataformas de implementação e especificação.

No anel mais externo na figura 2.2 vemos alguns aspectos não-funcionais freqüentemente empregados em sistemas informatizados, como segurança e transações. A abordagem MDA estimula o emprego adequado dessas propriedades na elaboração e implementação dos sistemas.

Finalmente, as setas que saem dos anéis na figura 2.2 representam as diversas áreas em que o MDA pode ser empregado, não limitado a elas, como podemos deduzir a partir do texto 'outros' da figura.

Um dos elementos centrais para realização da prática proposta pelo MDA é a confecção de um modelo chamado de PIM. O PIM (*Platform Independent Model*) é um modelo independente de plataforma que especifica de forma completa todos os requisitos funcionais do sistema. A partir dele, usando ferramentas de transformação automatizadas, podemos gerar um ou mais modelos PSM (*Platform Specific Model*), que são

modelos voltados para uma plataforma de *software* específica. A realização das transformações de modelos é um ponto vital para obter a percepção das vantagens do MDA.

Embora o modelo PIM (*Platform Independent Model*) possa ser representado em qualquer linguagem suportada pelo MDA, a OMG aponta o UML como a linguagem ideal para construir esse modelo. Uma das justificativas dessa escolha encontra-se no fato da ampla aceitação dessa linguagem para especificação de sistemas, conforme visto em [FS2000]. Além disso, UML suporta uma das características fundamentais dos modelos PIM: alto grau de abstração e independência de qualquer plataforma de *software*. No tópico 2.4.1, os aspectos positivos e algumas deficiências da linguagem UML com relação ao suporte a MDA serão abordados.

Uma vez construído o modelo PIM, um processo de transformação é executado gerando um ou mais PSM (*Platform Specific Model*). Cada PSM, por sua vez, quando submetido a outro processo de transformação, gera código executável para a plataforma na qual ele se baseia. A figura 2.3 - retirada de [KWB2003] - ilustra os processos de transformações em MDA de forma simplificada.

Figura 2.3 Modelos e Transformações Típicas Empregadas em MDA

2.3. Principais benefícios do MDA

- Produtividade

Uma vez definido um PIM seguindo todas as especificações propostas pelo MDA, o passo seguinte será gerar modelos PSM usando ferramentas e linguagens de transformação que automatizam esse processo. O grande problema nesse contexto é

selecionar – ou até mesmo criar - a linguagem de transformação adequada para essa transformação PIM para PSM. Entretanto, uma vez definida a ferramenta e a linguagem a ser utilizada num determinado contexto, essa transformação poderá ser repetida automaticamente diversas vezes e reaproveitada, eventualmente, em outros sistemas que necessitem da mesma transformação.

Sendo todas as implementações direcionadas para o PIM, o desenvolvedor não precisa se preocupar com detalhes específicos da plataforma para a qual o sistema será gerado, focando a atenção nos aspectos de negócio da aplicação. Esse foco no negócio, por conseqüência, produz como efeito positivo a maior possibilidade de que a especificação do analista esteja coerente com os requisitos demandados pelos usuários.

- Manutenibilidade e Documentação

Como em MDA o foco está no desenvolvimento do modelo PIM que, conforme já citado, é um modelo de alto nível de abstração para criação do *software*, ele representa a melhor documentação do sistema. A grande vantagem que a abordagem MDA oferece está no fato de que qualquer alteração necessária, motivada por novas implementações ou por correções da funcionalidade existentes, será implementada diretamente no PIM, mantendo a documentação do sistema sempre atualizada. Diferente do processo tradicional de desenvolvimento, o modelo PIM não é abandonado após sua criação [KWB2003].

A evolução das ferramentas possibilitará também que qualquer alteração feita em nível dos modelos PSMs ou dos códigos gerados seja refletida na atualização do modelo PIM que os gerou.

- Portabilidade

A própria característica de independência do modelo PIM é o ponto forte para suporte a portabilidade. A partir do PIM, podem ser gerados modelos PSMs para as mais diversas plataformas de *software*. A dificuldade, como já citado, será produzir a definição de transformação adequada para cada plataforma alvo. Para as plataformas mais populares, como Java, já existem transformações automatizadas disponíveis. Para as plataformas menos convencionais, será necessário especificar como a transformação será realizada, usando uma ferramenta que dê suporte à criação da definição de transformações. Essa especificação personalizada será certamente custosa e pouco viável para a maioria dos projetos de construção de *software*. Entretanto, uma vez definida essa transformação, podemos reutilizá-la tantas vezes quanto for necessário para o projeto e até para outros projetos.

O surgimento de transformações, caso a idéia da OMG para o MDA se realize, será natural e ocorrerá junto com as novas plataformas. Por força da demanda de mercado, pode-se esperar que os fornecedores de soluções de *software* venham a disponibilizar as devidas definições de transformações para suas ferramentas. Valorizando essa expectativa, a propaganda do MDA coloca que ela é uma proposta "à prova de futuro", conforme citado em [KWB2003]. Essa longevidade pretendida pela adoção do MDA poderá ser alcançada principalmente pelo estímulo no emprego de padrões abertos de implementação, como por exemplo, linguagens e protocolos padronizados como ANSI SQL, SOAP, C#, etc.

Outra forma de pensarmos em portabilidade seria na utilização de ferramentas diversas que utilizem os mesmos modelos PIM. MDA não é a primeira iniciativa de integração de meta-modelos (ou meta-dados, para usar o termo mais comum antes do MDA). Várias tentativas sem sucesso foram realizadas no final da década de 80 e início da de 90. Percebendo a miríade de tipos de meta-dados empregados em um único sistema, foram propostas várias abordagens independentes, como por exemplo a

da ferramenta *PowerDesigner* [Syb2005], que visavam unificá-los em um único e definitivo meta-modelo. Ainda não era reconhecido que cada aspecto do sistema seria melhor implementando usando a linguagem mais adequada para cada caso. A falha dessas primeiras tentativas desuniu esses proponentes. MDA e MOF surgem, então, nesse contexto.

2.4. Padrões envolvidos

2.4.1. UML (*Unified Modeling Language*)

A UML é uma linguagem para modelagem adotada pela OMG. Combinado com o padrão MOF, a UML provê os elementos básicos para a proposta MDA [Uml2004].

Inicialmente, a linguagem UML foi adotada para modelagem de sistemas sem a pretensão de que os modelos resultantes pudessem servir de base para a geração automática dos artefatos de *software* necessários para a execução do sistema. Algum tempo depois, alguns trabalhos voltados para criação de modelos executáveis surgiram e, devido à ampla aceitação da linguagem UML no mercado, adotaram a UML como linguagem base para confecção dos modelos executáveis. Um exemplo concreto desse tipo de proposta é a *Executable UML* [MB2003], que estende a linguagem UML dando-lhe o suporte semântico necessário para que a execução do modelo fosse possível. Um dos problemas principais dessa proposta é a não padronização.

As versões iniciais da UML enfocavam principalmente as notações gráficas chamadas de representações concretas (*Concrete Syntax*). Com a adoção do padrão pela OMG e sua orientação para o suporte a MDA, cresceu a preocupação com a representação abstrata (*Abstract Syntax*) em UML. Como representação abstrata podemos entender os conceitos formais subjacentes a cada uma das representações concretas que a linguagem representa. Por exemplo, a notação gráfica de classe – um retângulo conten-

do três divisões horizontais – não teria nenhum significado caso não fosse convencionado que ela representa uma abstração de interesse ao nosso sistema. Para formalizar essa representação abstrata, a OMG definiu o meta-modelo da UML.

O meta-modelo UML é um modelo escrito na linguagem padronizada pelo MOF [Mof2002] que confere significado aos elementos abstratos da linguagem UML. O modelo da figura 2.6 representa o meta-modelo simplificado da linguagem UML. O padrão MOF será explicado detalhadamente no próximo tópico.

O formalismo nas representações abstratas dos elementos da linguagem UML permite também que ela possa ser representada em uma linguagem textual e estruturada. Essa linguagem é baseada no XML e foi padronizada pela OMG, sendo batizada de XMI [Xmi2003]. O padrão XMI será abordado ainda neste capítulo.

Outro elemento forte que favorece a adoção da linguagem UML para a construção de modelos PIM na arquitetura MDA é a sua característica extensível. Essa extensibilidade é fornecida pelos *UML Profiles*. Através dos *Profiles* podemos especializar a linguagem UML para algum aspecto não contemplado na mesma e que seja de interesse num contexto em particular. Como veremos posteriormente, os *UML Profiles* têm um importante papel na complementação semântica dos modelos PIM que favorecem a automatização de suas transformações para modelos PSMs.

A independência de plataforma que a UML emprega por natureza torna-se vital para sua adoção em MDA [MB2003]. Embora tenha essa independência, UML pode ser formalizada e precisa o suficiente para guiar as transformações automatizadas requeridas em MDA. Esse formalismo é conseguido pelo emprego de uma linguagem adicional a UML chamada OCL [Ocl2003]. Um exemplo da combinação UML/OCL será apresentado no tópico 2.5.3.3.

2.4.2. MOF (*Meta-Object Facility*)

O MDA exige a utilização de modelos escritos numa linguagem bem-definida, ou seja, que possa ser interpretada não só por seres humanos, mas também por computadores. Uma linguagem bem-definida é aquela que obedece a um conjunto de regras formais. MOF [Mof2002] é uma linguagem para descrição de meta-modelos que, por sua vez, irão definir regras sobre como os modelos de nossas aplicações devem ser escritos para serem considerados bem-definidos.

No contexto do MDA, MOF classifica os modelos que utilizamos em quatro níveis de modelagem. A figura 2.4 mostra cada um desses níveis.

Meta Nível	Descrição	Elementos
M3	Nível MOF, ou conjunto de construções usado para definir meta-modelos.	Classes MOF, Atributos MOF, Associações MOF, etc.
M2	Meta-modelos, consistindo de instâncias das construções MOF	Classes UML, Associações UML, Atributos UML, Estados UML, etc.
M1	Modelos, consistindo de instâncias dos meta-modelos no nível M2	Classe "Consumidor", Classe "Conta" Tabela "Empregados", Tabela "Vendedores", etc.
M0	Objetos e dados, isto é, instâncias dos modelos no nível M1	Consumidor Jane Smith, Consumidor Joe Jones, etc.

Figura 2.4 Níveis MOF de Modelagem

No nível M0 encontramos os objetos sobre qual nosso sistema tem interesse. Esses objetos são instâncias de classes definidas no meta-modelo do nível M1. Considerando o sistema Acadêmico, usado como exemplo e apresentado no capítulo 3 deste trabalho, *João* seria um elemento em nível M0 que, por sua vez, é uma instância da classe *Aluno* do meta-modelo subjacente em nível M1.

O modelo no nível M1 especifica o modelo da aplicação, no sentido tradicional. Nesse nível encontramos objetos que são instâncias de classes definidas no meta-modelo do nível M2 (modelo MOF). Considerando o sistema exemplo Acadêmico, *Aluno* seria um elemento em nível M1 que, por sua vez, é uma instância da classe MOF *Class* do meta-modelo MOF do nível M2.

O nível M3 representa o meta-meta-modelo do MOF. Embora pareça bastante estranho compreender a necessidade de um meta-modelo de um meta-modelo, é justamente dessa característica que o MOF se vale para permitir a integração de várias linguagens diferentes na definição de um sistema. Imagine um sistema simples que use um modelo relacional e um modelo para objetos. Nesse exemplo simples, temos dois meta-modelos: um para o modelo relacional e outro para o modelo de objetos. O meta-meta-modelo do nível M3 do MOF definirá elementos básicos que permitirão a integração dos elementos (instâncias do elementos do nível M3, para ser mais preciso) dos dois meta-modelos do nível M2. É nesse nível que as linguagens de meta-modelagem e os meta-modelos operam, definindo os conceitos mais simples requeridos para capturar a representação deles e possibilitando o intercâmbio desses artefatos entre ferramentas.

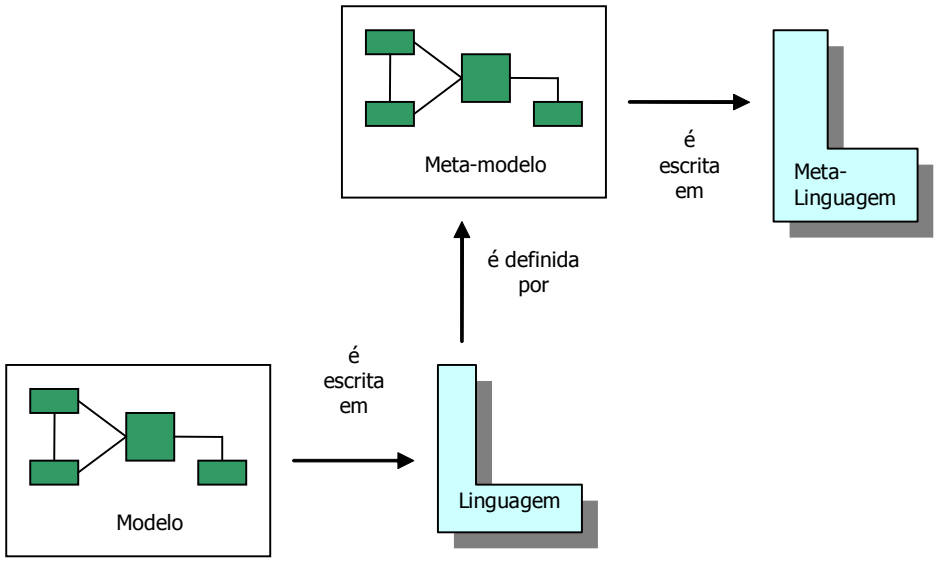


Figura 2.5 Modelos, linguagens, meta-modelos e meta-linguagens

Na figura 2.5, que ilustra o cenário de aplicação do MOF, retirada de [KWB2003], vemos que um modelo é escrito através das definições impostas por uma linguagem de modelagem. Esta, por sua vez, é definida por um meta-modelo, que determina como a linguagem de modelagem deve ser escrita. Finalmente, esse meta-modelo é escrito em uma linguagem padronizada pelo MOF. Como essa linguagem é utilizada para definir linguagens, ela é considerada uma meta-linguagem.

MOF se baseia na premissa de que mais de um tipo de modelo existirá na modelagem de sistemas, portanto deverá existir mais de uma linguagem de modelagem. O objetivo é ter a capacidade de utilizar diversos tipos de modelos para representar os vários aspectos do nosso sistema, cada um desses modelos sendo apropriado para definir uma linguagem de abstração especializada para um determinado propósito.

O MOF, combinado com o *UML Profile*, é o mecanismo que dá ao MDA a capacidade de definir diversas linguagens através de modelos.

Por exemplo, o modelo de classes da UML é um modelo comumente encontrado na especificação de sistemas orientados a objetos. Como um modelo, devemos ter a capacidade de representá-lo formalmente – usando MOF – através de um meta-modelo. A figura 2.6, retirado de [Fra2003], define uma meta-modelagem para o modelo de classes da UML.

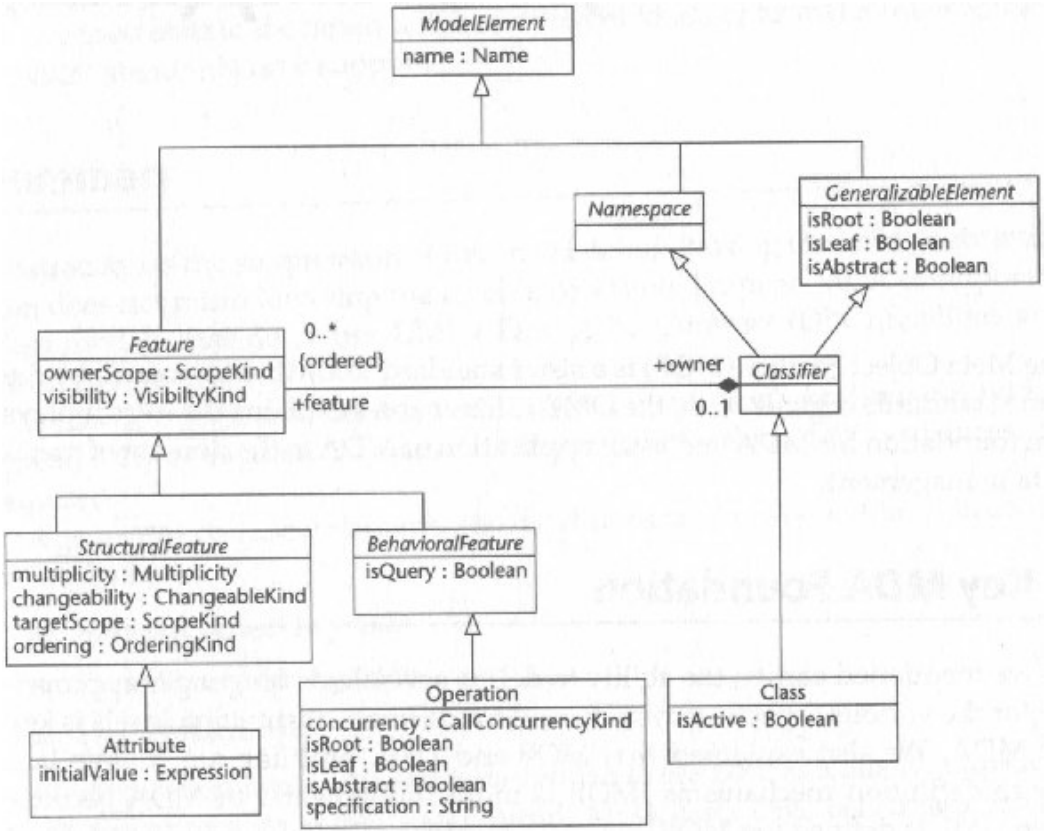


Figura 2.6 Fragmento do Meta-modelo do Diagrama de Classes UML

No meta-modelo para modelo de classes da UML, percebe-se que o MOF utiliza o próprio modelo de diagramas de classes da UML para descrever o modelo formal, tirando proveito da característica flexível que essa linguagem de modelagem oferece. Podemos utilizar os elementos de modelagens formais, sejam eles próprios da UML ou oferecidos pela OCL, para complementar a semântica do meta-modelo.

No meta-modelo da figura 2.6, podemos notar a definição formal de objetos que aparecerão em modelos derivados – instâncias do modelo de classes UML – e como esses elementos estão relacionados. Por exemplo, a classe MOF *Class* representa a definição de várias ocorrências de uma classe no modelo derivado. Usando um exemplo real do modelo Acadêmico, podemos dizer que a classe *Aluno* é uma instância da classe MOF *Class* do seu respectivo meta-modelo.

Neste ponto fica clara a importância do MOF para a abordagem MDA. MOF é o padrão necessário para que o MDA possa capacitar os engenheiros de *software* na integração dos vários modelos necessários para a definição de um sistema.

Por exemplo, outro modelo usualmente encontrado na especificação de operações realizadas por um sistema em UML é o modelo de estados. O modelo de diagrama de estados é definido em MOF pelo meta-modelo da figura 2.7, retirado de [Fra2003].

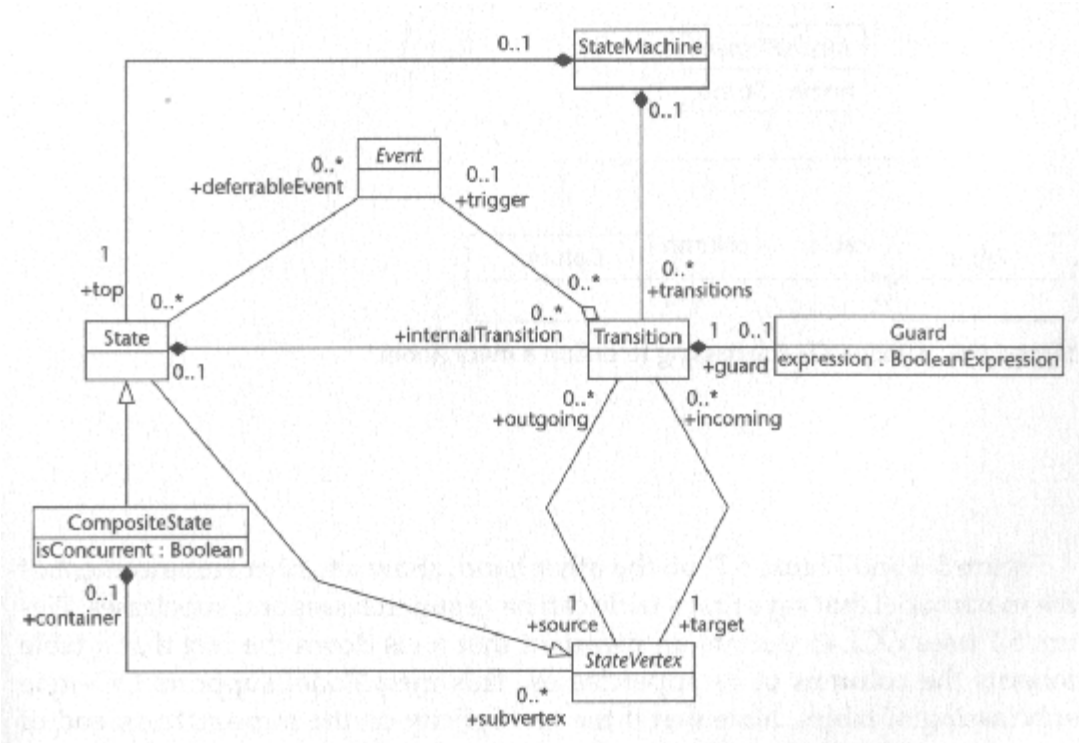


Figura 2.7 Meta-modelo do Diagrama de Estados UML

Os exemplos de meta-modelos citados anteriormente podem dar a falsa impressão de que o MOF é utilizado apenas para modelagem orientada a objetos. Na verdade, MOF pode ser usado para a representação de meta-modelos legados, ou não orientados a objetos. Como exemplo, considere o meta-modelo ilustrado na figura 2.8 para o tradicional modelo de um banco de dados relacional. Nesta figura vemos representados apenas os elementos principais desse tipo de modelo: tabelas e colunas. Outros detalhes foram omitidos por motivos de simplificação.

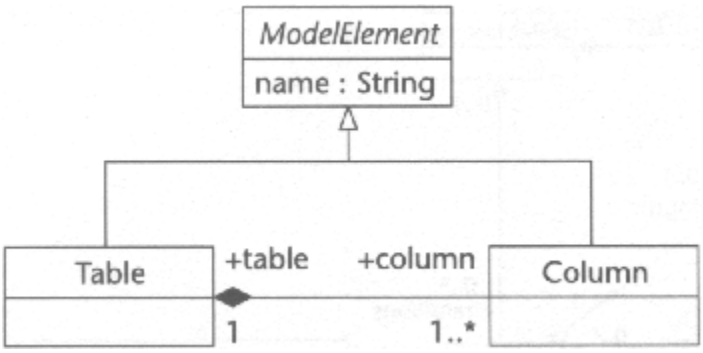


Figura 2.8 Meta-modelo Relacional Simplificado

Como já foi dito, é de fundamental importância refinar a semântica dos meta-modelos definidos com o MOF. Como o diagrama de classes tem limitações – como aliás terá qualquer outro modelo visual – para representar todos os aspectos e restrições que um modelo comumente deve impor, necessitamos de algo mais para definir as regras que irão dar o formalismo necessário para o modelo. Isto é feito utilizando uma linguagem formal para especificar regras que não são contempladas pelos elementos gráficos fornecidos pelo diagrama de classes UML. A formalidade na descrição dessas regras é necessária para que sejam entendidas e processadas por ferramentas automatizadas. A especificação da UML define uma linguagem própria para esse objetivo, que é a linguagem OCL (*Object Constraint Language*) [Ocl2003].

Para exemplificar a necessidade de completar semanticamente um meta-modelo, consideremos o meta-modelo simplificado para modelos relacionais descrito na figura 2.8. Poderíamos ter, para um determinado ambiente, a necessidade de especificar que uma coluna não pode ter o mesmo nome da tabela que a contém. Como pode ser observado, o modelo definido na figura não tem essa regra especificada e nem meios para fazê-lo visualmente. O código 2.1 em OCL complementaria o referido modelo para impor, automaticamente, a regra que queremos definir.

```
context Table inv:
 column -> forAll ( col | col.name <> self.name)
```

Código 2.1 Regra OCL do Meta-Modelo Relacional Simplificado

O código 2.1 inicia a definição da regra estabelecendo que o elemento *Table*, definido no meta-modelo da figura 2.8, será o escopo para a regra. O termo *column* refere-se ao *'association end'* da associação entre os elementos *Table* e o *Column* do mesmo meta-modelo. A instrução *forAll* indica que a regra a ser definida entre parênteses valerá para todos os elementos associados, que no caso são as colunas de uma tabela. A regra então indica que, de todas as colunas de uma tabela, nenhuma poderá ter o mesmo nome (propriedade *name* do meta-elemento *column*) que o da tabela (representada pelo palavra reservada *self* seguida da propriedade *name*).

O MOF também define um padrão para armazenamento e obtenção de modelos chamado MOF *Repository*. O objetivo desse repositório é definir uma maneira padronizada de armazenar modelos – baseados em MOF – para que várias ferramentas de modelagem possam utilizá-los de forma harmônica. Um exemplo de interface para MOF *Repository* já padronizada pela OMG é a MOF-CORBA. Existe, entretanto, outros padrões de interface para acesso a esse repositório, como é o caso do JMI, baseado em Java. Em [PJ2004] é apresentada uma proposta de linguagem chamada mSQL para consulta a repositórios MOF.

2.4.3. XMI (*XML Metadata Interchange*)

XMI [Xmi2003] é um padrão baseado em XML que define o formato para troca de documentos sobre modelos e meta-modelos. Essa padronização, como é a regra em dialetos escritos em XML, é feita através da definição de um documento com regras de validação. Na versão 2.0 do XMI, essas regras podem ser escritas em documentos DTDs ou XML *Schemas*. O padrão XMI faz parte do padrão MOF, pois o último formaliza os elementos sintáticos básicos de um modelo de tal forma que eles possam ser exportados para elementos XML válidos segundo as regras do XMI.

Através da padronização estabelecida pelo XMI, várias ferramentas de modelagem que aderem ao MDA poderão trocar modelos e cooperarem no processo de criação de modelos ou nas transformações necessárias para a geração de código executável. As ferramentas de modelagem que se auto proclamam compatíveis com MDA devem permitir que os modelos por eles gerados sejam formatados em XMI.

Colocando de maneira mais prática, podemos considerar XMI um padrão para serialização de modelos. Para serializar um modelo, precisamos do meta-modelo no qual ele está baseado. Para exemplificar a serialização realizada pelo XMI, considere-se parte do modelo UML do aplicativo Acadêmico – mostrando apenas a classe *Disciplina* - na figura 2.9. O código em 2.2 é o modelo UML serializado usando o padrão XMI.

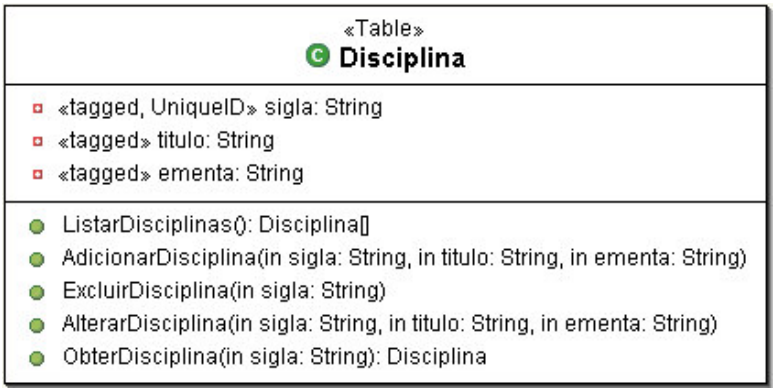


Figura 2.9 Parte do modelo PIM do sistema Acadêmico

```
<?xml version="1.0" encoding="UTF-8"?>
<editmodel:ClassDiagramEditModel xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:editmodel="editmodel.xmi" xmlns:options="options.xmi" name="PIM"
size="825,731" id="." metadata="nsuml-1.4" initialized="true" scrolledX="-3"
scrolledY="-2" tag="5" key="6D697373696E6720416361646564D69636F">
  <children xsi:type="editmodel:ClassEditModel" name="Disciplina"
location="2,203" size="381,195" stereotype="Table" id="academico/Disciplina"
packageIndication="1" runTimeClassModel="">
    <children xsi:type="editmodel:CompartmentEditModel" size="198,108">
      <children xsi:type="editmodel:AttributeEditModel" name="sigla"
stereotype="tagged, UniqueID" taggedValues="{Size=10}"
id="academico/Disciplina#sigla"/>
      <children xsi:type="editmodel:AttributeEditModel" name="titulo"
taggedValues="{Size=40}" id="academico/Disciplina#titulo"/>
      <children xsi:type="editmodel:AttributeEditModel" name="ementa"
taggedValues="{Size=8000}" id="academico/Disciplina#ementa"/>
    </children>
</editmodel:ClassDiagramEditModel>
```

Código 2.2 Fragmento de Código XMI do Modelo PIM do Acadêmico

No código em 2.2, podemos notar como alguns elementos do modelo PIM foram representados dentro do código XML, obedecendo à gramática ditada pelo padrão XMI. A classe *Disciplina*, por exemplo, está destacada em negrito e faz parte do elemento *children* do XML. Os atributos da classe também estão destacados em negrito e, como podemos observar, são declarados da mesma forma com o elemento *children*, só que aninhados ao elemento referente à classe a qual pertencem.

2.5. Transformações

2.5.1. Introdução

As transformações possuem um papel fundamental na abordagem MDA, pois são através delas que as vantagens prometidas ficam evidentes.

O objetivo da transformação, dentro do contexto do MDA, é transformar um modelo expresso em um determinado nível de abstração em outro modelo, usualmente expresso em um menor nível de abstração, ou seja, em um modelo mais próximo da aplicação do sistema em uma determinada tecnologia. Esse tipo de transformação é o mais comum dentro do contexto MDA e é chamado de transformação vertical em [MSUW2004].

Em [Fra2003], as transformações são simplificadas em dois passos colocados pelo autor como vitais num processo aplicando o MDA: a transformação PIM para PSM e a transformação PSM para código. Entretanto, podem-se utilizar transformações intermediárias ou transformações que até mesmo mantenham o nível de abstração do modelo origem e destino. Essas transformações que mantém o nível de abstração podem, a princípio, parecer desnecessárias dentro de um processo MDA, mas são úteis para representar uma visão do sistema sobre um outro contexto ou usando uma representação diferente. Esse tipo de transformação é chamada de transformação horizontal em [MSUW2004].

Outra vantagem que pode ser obtida com transformações automatizadas é o eventual emprego de *Design Patterns*. Uma boa ferramenta de transformação poderá aplicar um *Design Pattern* consagrado que resolva a implementação de alguma funcionalidade do sistema projetado. Em outros casos, o projetista poderá identificar um ce-

nário onde caiba a aplicação de um *Design Pattern* e, através do provimento de parâmetros, guiar a ferramenta de transformação para a aplicação dele.

Para que a transformação possa ser realizada, alguns elementos básicos devem existir. O próximo tópico procura esclarecer esses pontos.

2.5.2. Elementos básicos para a transformação

A figura 2.10, extraída de [KWB2003], ilustra o processo de transformação de um modelo PIM para um PSM e cada um dos elementos necessários para sua realização.

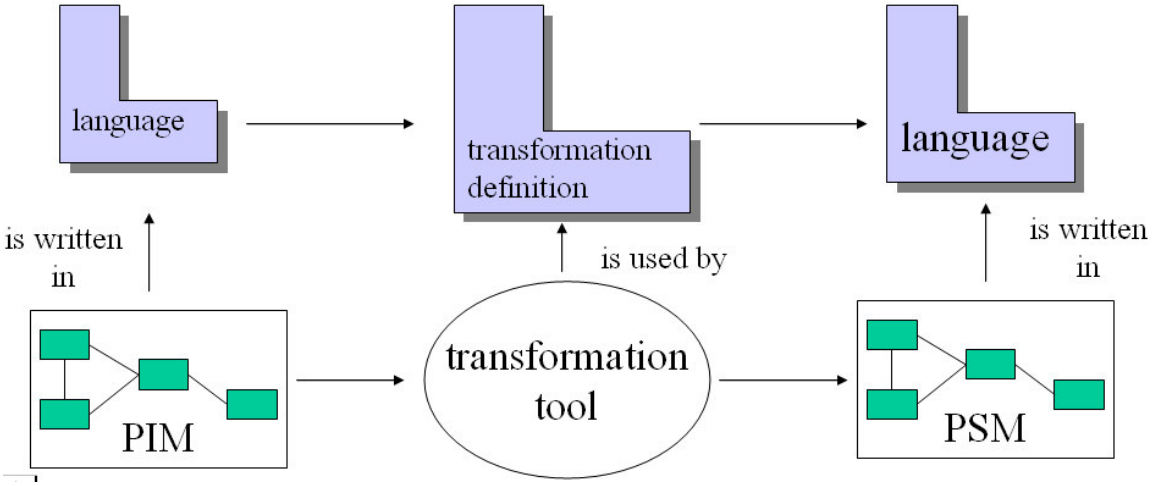


Figura 2.10 Elementos Básicos para Transformação de Modelos em MDA

Inicialmente devemos entender a importância das linguagens que descrevem os modelos para o processo de transformação. No exemplo da figura 2.10, o modelo PIM – que nas práticas de MDA geralmente é expresso em UML – é escrito numa linguagem formalizada através de um meta-modelo. Esse meta-modelo é um modelo que segue o padrão MOF e que proporciona o formalismo necessário para que o processo de transformação possa acontecer. De forma análoga, o modelo PSM gerado também precisa ter o suporte formal para que a ferramenta de transformação possa inequivocamente determinar o que deve ser gerado como resultado de sua "execução". A definição da transformação relaciona a meta-classe de um elemento no modelo de origem com uma ou mais meta-classes de elementos no modelo de destino.

Generalizando o processo de transformação da figura 2.10, podemos identificar, então, quatro elementos básicos: o modelo de origem e o modelo de destino, ambos

embasados em uma linguagem que lhes confere o formalismo necessário, uma ferramenta de transformação e finalmente uma linguagem específica para instruir a ferramenta sobre como a transformação deve ocorrer – a chamada definição da transformação.

O processo de padronização da linguagem a ser usada para escrever as definições de transformação está em andamento na OMG. Embora em fase de embrionária, essa linguagem está sendo chamada pelo nome *Query, Views And Transformations* (QVT).

Tendo em vista a lacuna ainda existente pela falta de uma linguagem que descreva a definição da transformação, Kepple et al em [KWB2003] propõe uma linguagem de transformação que, segundo os próprios autores, embora não tenha a pretensão de antecipar o que a linguagem QVT será, deverá se parecer com a linguagem final que a OMG adotará, em linhas gerais. Outras propostas, como a apresentada em [JMK2004], estão colaborando para delinear a linguagem final que será adotada.

2.5.3. Adicionando semântica ao modelo

Um dos desafios ao criar um modelo que possa ser submetido a uma ferramenta para que ela o transforme em outro modelo (executável ou não) é conferir riqueza de detalhes suficiente ao modelo fonte para que todas as decisões de transformação possam ser realizadas inequivocamente. Como foi dito no tópico anterior, a definição da transformação necessitará de subsídios para escolher qual transformação melhor se adequa para cada um dos casos de transformação.

Os tópicos a seguir detalham as três principais abordagens para aumentar a riqueza semântica dos modelos:

2.5.3.1. Linguagem de Ação

Na versão 1.4 da linguagem UML, a parte estrutural no sistema pode ser completamente representada no modelo. Entretanto, essa versão apresenta deficiências com relação à especificação da parte dinâmica do sistema devido à ausência de precisão semântica na modelagem comportamental, como podemos verificar no trabalho de [Mash2004]. A OMG já identificou essa necessidade e espera resolvê-la na versão 2.0, que está em fase final de aprovação pela entidade.

Em [MB2003], é proposta uma extensão para a UML que preenche a lacuna de especificação comportamental que esta apresenta. Nesse trabalho, os autores propõem a utilização de UML puro adicionado a uma linguagem de especificação chamada de *Action Semantics* (AS). Essa linguagem permite especificar o comportamento de cada uma das ações do sistema com precisão suficiente para gerar código executável. Ou seja, para cada operação do diagrama de classes, deve-se codificar – usando AS – o comportamento dinâmico dessa operação. O grande problema apresentado por essa proposta está no fato da linguagem AS assemelhar-se a uma linguagem computacional, com bai-

xo nível de abstração. No trabalho em [Mash2004], o autor expõe as deficiências dessa abordagem no contexto MDA que se propõe a especificar o sistema em um alto nível de abstração.

2.5.3.2. Extensão da Linguagem UML

Uma das soluções propostas para o complemento semântico do modelo comportamental é apresentado em [Fra2003]. Essa proposta aponta duas abordagens que podem ser combinadas para a solução do problema: o uso da meta-modelagem e *UML Profiles*.

Na meta-modelagem – chamada no trabalho em [Fra2003] de *Heavyweight UML Metamodel Extension* – devemos especializar o meta-modelo dos modelos fonte e destino de maneira que todas as alternativas de transformação possam ser representadas. Essa abordagem, como o próprio trabalho de Frankel [Fra2003] aponta, apresenta deficiências na reutilização desses meta-modelos para criação de modelos de propósito semelhante.

Os arquitetos responsáveis pela especificação da UML, ao invés de tentar definir uma linguagem que tivesse a pretensão de abranger todas as possíveis necessidades que as pessoas teriam ao usá-la, equiparam-na com um mecanismo de extensão, os *UML Profiles* - chamada em [Fra2003] de *Lightweight Extension*. *UML Profile* é uma forma de estender a semântica de um modelo através de dois tipos de marcadores: estereótipos e *tagged values*.

Em [MSUW2004] esses *Profiles* são chamados de *Marks*, qualificados como extensões não intrusivas para modelos (sem poluí-los) que capturam informações requeridas para as transformações. Esse mesmo trabalho aponta quatro principais fornecedores de *UML Profiles*: organizações de padronização (como a OMG), fornecedores de ferramentas, definidores de metodologias e, por final, os desenvolvedores. A idéia defendi-

da pelos autores é que estaremos mais freqüentemente utilizando *Profiles* prontos do que definindo nossos próprios. O trabalho em [WJ2004] evidencia a importância que bibliotecas de *Profiles* terão em MDA.

- Estereótipo: é um marcador aplicado a determinados tipos de elementos do modelo que lhes confere alguma adição de significado. Visualmente um estereótipo é colocado ao lado de um elemento do modelo do sistema delimitado pelos sinais *<<estereótipo>>*. A figura 2.11 mostra uma classe do modelo do sistema Acadêmico (ver capítulo 3) e o estereótipo *<<Table>>*, usado para identificar que uma determinada classe do PIM gerará uma tabela no PSM relacional. Esse e outros estereótipos relacionados serão discutidos com maior profundidade no capítulo 4.

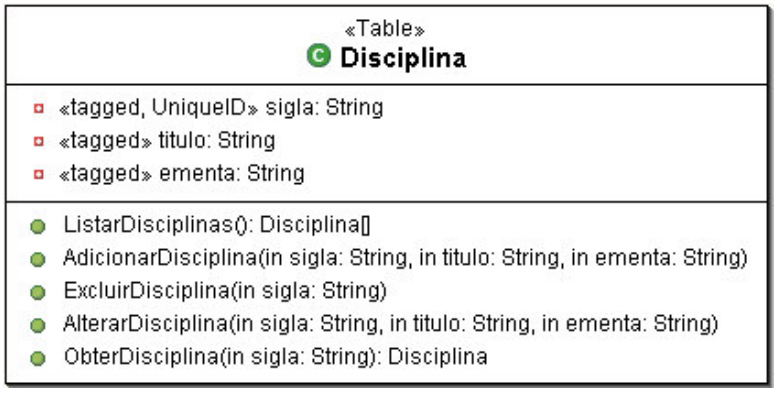


Figura 2.11 Fragmento Disciplina do Modelo PIM do Acadêmico

- *Tagged value*: é outro tipo de marcação usado pelo *UML Profiles*. Esse tipo de marcação pode ser utilizado junto com um estereótipo ou de forma livre, associado a qualquer outro elemento do meta-modelo UML, como atributos ou métodos.

  Embora a ferramenta utilizada neste trabalho não mostre explicitamente (ela apenas mostra que o atributo possui um *tagged value* através do indicador

<<tagged>> que o precede), o atributo *titulo* da classe *Disciplina* mostrado na figura 2.11 possui o *tagged value Size*, cujo objetivo será estipular, na geração do modelo relacional, quantos caracteres o atributo suportará. Outros *tagged values* relacionados à transformação para o modelo relacional serão abordados no capítulo 4.

Os *UML Profiles* são definidos formalmente por um modelo usando uma notação gráfica associada a regras que irão complementá-lo, como acontece com qualquer outro modelo. Em [MSUW2004] esses modelos são chamados de modelos de marcação. Eles guardam a mesma relação com o *Profile* que os meta-modelos fazem com os modelos. A figura 2.12 mostra a definição do *profile* utilizado nos exemplos acima.

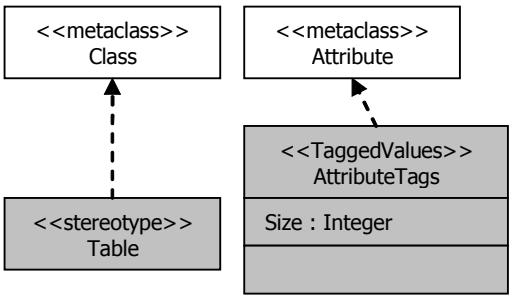


Figura 2.12 Fragmento do Modelo do *UML Profile* Relacional

2.5.3.3. *Design By Contract* (DBC)

Outra forma de apurar a precisão do modelo, além do uso das técnicas de meta-modelagem e *UML Profiles*, é através da utilização da linguagem OCL para definir regras e contratos para elementos do nosso modelo. Essa técnica é conhecida como *Design By Contract* (DBC) e explicada com maior profundidade no trabalho de [Fra2003].

O DBC enfoca na especificação de contratos explícitos e formais da interface de um *software*, colaborando não só no aprimoramento da qualidade do sistema, mas

permitindo, com o uso de ferramentas adequadas, o aumento da produtividade através da geração automática de códigos que obedecem às regras estabelecidas. Existem duas maneiras básicas para especificar contratos:

- Pré-condições e Pós-condições

  As pré-condições e pós-condições são especificações aplicadas a operações. Uma pré-condição estabelece uma regra que deve ser verdadeira antes do início da execução da operação. A pós-condição, em contrapartida, estabelece uma regra que deve ser verdadeira após o término da execução da operação.

- Invariantes

  A invariante é uma asserção feita sobre o estado do sistema que deve ser verdadeira durante toda a execução do mesmo, exceto durante a execução de operações. Durante a execução de uma operação, a invariante pode ser violada momentaneamente devido a algum estado temporário normal ao fluxo de execução. As invariantes, ao contrário das pré e pós-condições, não se aplicam a uma operação específica do *software*. Elas estão associadas ao sistema como um todo. Usualmente, como apontado em [MSUW2004], as invariantes são verificadas ao término da execução de cada uma das operações do sistema.

A linguagem OCL [Ocl2003] faz parte do padrão UML e possibilita desenvolvedores de *software* escreverem regras e consultas sobre modelos de objetos. Essas regras, escritas de maneira formal, permitem refinar a precisão de regras de negócio e enriquecer semanticamente os diversos tipos de modelos empregados no MDA, favorecendo a geração automática de código.

Além de aplicada a modelos – incluídos aí os meta-modelos - a linguagem OCL também é utilizada em *UML Profiles* para refinar e contextualizar a aplicação do *profile*.

Uma regra pode ser escrita em OCL para restringir a aplicação de um estereótipo apenas a alguns elementos do meta-modelo.

O objetivo no uso do OCL vai além do mero refinamento nas regras de validação de um modelo. Os códigos OCL podem orientar as ferramentas de transformação para geração de código de implementação. Esses códigos podem estar voltados para dois propósitos básicos: garantia das regras estabelecidas para o modelo – como no exemplo do código 2.1 - ou para a completa geração de código que implemente um determinado aspecto dinâmico do sistema.

Para exemplificar a aplicação de código OCL para complementação semântica de regras e para dirigir a completa geração de um código de implementação em uma linguagem qualquer, como Java ou C#, consideremos o modelo simples para uma aplicação bancária da figura 2.13. Nesse modelo, encontramos um cenário comum nesse domínio de aplicação: uma conta bancária pode ser do tipo poupança ou conta corrente, representado pela herança feita a partir da classe genérica *Conta*. Qualquer conta deve ter uma identificação (atributo *id*), um saldo consolidado (atributo *saldo* usando o tipo *Double* definido para UML) e um titular (definido pelo tipo *Cliente* omitido do modelo por simplificação). O modelo generaliza a operação *TransfereFundo*, que realiza a transferência de um tipo de conta para outra. Para não poluir o modelo, os parâmetros da referida operação foram omitidos.

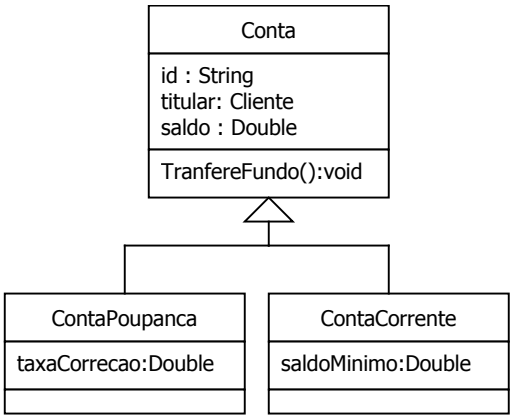


Figura 2.13 Modelo Aplicação Bancária

Uma regra de negócio típica nessa aplicação seria indicar que nenhuma conta corrente pode ter saldo menor que o saldo mínimo determinado pela respectiva propriedade. A regra OCL do código 2.3 define uma invariante que garante essa restrição de negócio. O texto precedido de dois traços indica um comentário na linguagem OCL e é ignorado pelo compilador.

```
context ContaCorrente inv:

-- Saldo não pode ser menor que o saldo mínimo definido
saldo >= saldoMinimo
```

Código 2.3 Invariante OCL Para a Classe ContaCorrente do Modelo Bancário

A invariante definida no código 2.3 poderia orientar a transformação para gerar, por exemplo, um gatilho no banco de dados que verifique a regra sempre que o saldo for modificado. Caso a regra não seja atendida, a operação de alteração seria cancelada e uma exceção gerada pela aplicação.

```
context Conta::TransfereFundo(de:ContaCorrente,para:ContaPoupanca,valor:Double):void

pre:
 -- O saldo da conta corrente origem deve ser suficiente
 de.saldo >= valor

pre:
 -- A conta de origem e a conta de destino devem ser do mesmo correntista
 de.titular = para.titular
```

```
post:
 -- O saldo da conta corrente é reduzido do valor transferido
 de.saldo = de.saldo@pre - valor

post:
 -- O saldo da conta poupança é acrescido do valor transferido
 para.saldo = para.saldo@pre + valor
```

Código 2.4 Definição da Operação TranfereFundo do Modelo Bancário

O código 2.4 define em OCL a operação *TransfereFundo* do modelo bancário da figura 2.13. As duas pré-condições são bem diretas e garantem que o saldo deve ser maior que o valor sendo transferido e apenas entre contas de mesma titularidade. As pós-condições definem a transferência propriamente dita. O indicador *@pre* indica que o valor da propriedade em questão se refere ao anterior. Em [KW1998], é oferecido um aprofundamento nos detalhes da linguagem OCL.

Repare que, embora seja uma definição em alto nível de abstração, o código OCL poderá, em muitos casos, ser suficiente para a completa geração de um código de implementação.

2.5.4. Problemas de Sincronização

A sincronização é uma necessidade típica de sistemas CASE que possuem algum nível de automatização para geração de código. Ao transformar um modelo em código, este poderá necessitar de ajustes que devem ser preservados. Um processo de desenvolvimento exige diversas interações e revisões no modelo PIM e no código gerado durante o refinamento do sistema.

Neste contexto de preocupação, consideramos duas abordagens extremas: engenharia ida-e-volta completa (*Full Round-Trip Engineering*) e a engenharia unidirecional (*Forward Engineering Only*).

No primeiro tipo de sincronização, qualquer alteração realizada no modelo deve ser refletida no código sem que possíveis refinamentos feitos anteriormente por um programador ou analista sejam perdidos. De forma inversa, qualquer inclusão ou alteração no código deve ser refletida no modelo PIM. Esse tipo de sincronização, apesar de ser mais flexível, é de difícil implementação e uso na prática. Nenhuma ferramenta atualmente implementa este tipo de sincronização, como dito em [Fra2003]. O diagrama de estados da figura 2.14, retirada de [Fra2003], representa este tipo de sincronização.

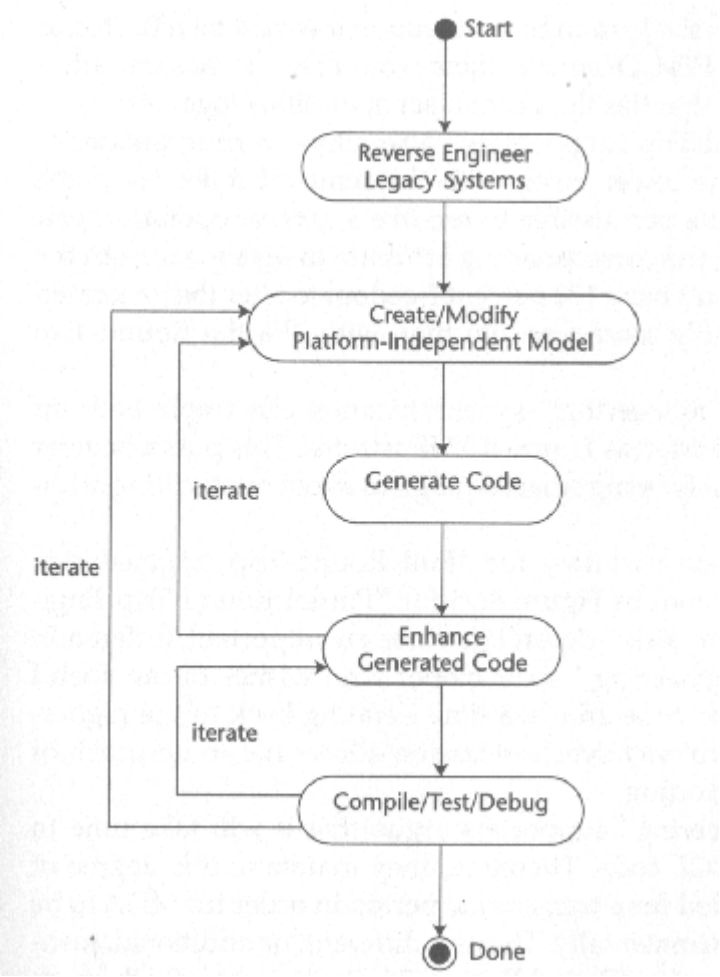


Figura 2.14 Sincronização Ida-e-Volta Completa

Na sincronização unidirecional, alterações podem ser feitas apenas em nível do modelo PIM. Este tipo de sincronização é obviamente o mais simples e considerado o mais puro na concepção MDA, visto que desejamos implementar todo e qualquer aspecto do sistema em nível de modelo. A figura 2.15, retirada de [Fra2003] , mostra o diagrama de estados representando as interações durante o desenvolvimento usando essa abordagem.

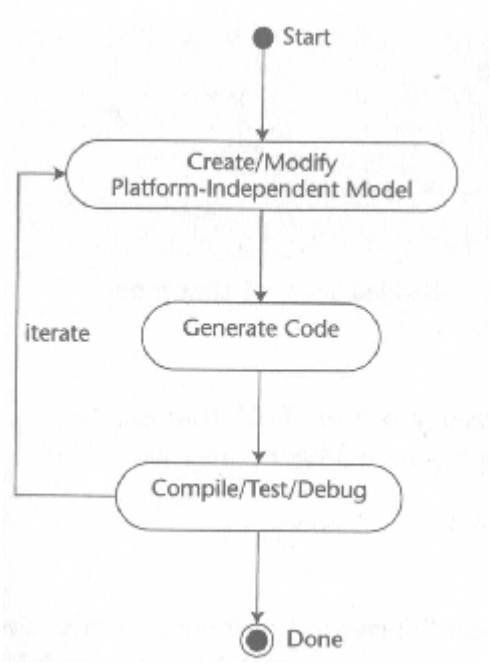


Figura 2.15 Sincronização Unidirecional

Em [AP2004] é apresentado um estudo interessante sobre o impacto dos problemas de sincronização e os aspectos que devem ser considerados pelas ferramentas de transformação para lidar com essas questões.

2.6. Conclusão

O MDA é uma abordagem ousada que propõe utilizar linguagens de modelagem como linguagens de programação, ampliando ainda mais o nível de abstração do projeto de arquitetura de *software* para sua implementação em uma determinada plataforma hardware ou sistema operacional.

Entretanto, os padrões que visam atingir os objetivos do MDA ainda estão em estágio de amadurecimento e muitas mudanças podem ocorrer nos próximos anos.

Sem dúvida, o que irá de fato impulsionar a proposta do MDA será o aparecimento de ferramentas de modelagem que suportem os padrões propostos por ela e que produzam de fato os resultados esperados, ou seja, a partir de um modelo independente de plataforma (PIM), um modelo para uma plataforma específica (PSM) seja gerado automaticamente. Entretanto, da mesma forma que os padrões aguardam amadurecimento, as ferramentas ainda implementam apenas alguns detalhes que MDA sugere. Já existem hoje algumas ferramentas que implementam boa parte das transformações necessárias para a realização completa de um sistema, sendo um exemplo de ferramenta apresentado em [WJ2004]. Outras ferramentas de transformação foram avaliadas em [Mash2004].

Entretanto, ainda existe muito a ser feito e padronizado no contexto das transformações para que os objetivos traçados pelo MDA possam ser alcançados. Um dos padrões fundamentais que ainda se encontra em processo de elaboração é a linguagem QVT.

3. APLICAÇÃO EXEMPLO

3.1. Introdução

Com o objetivo de pôr em prática as propostas do MDA, experimentando vários aspectos da abordagem, elaboramos um sistema típico de três camadas – apresentação, negócio e persistência – com um tema recorrente na área de informatização.

Como o objetivo primordial desse trabalho é explorar a aplicação do MDA, não é esperado que o resultado do sistema em estudo contemple todos os aspectos que um sistema informatizado típico para área se proporia a resolver.

O sistema acadêmico, que ora é proposto como estudo de caso, visa informatizar parte das atividades necessárias para uma instituição de ensino. No tópico seguinte serão descritos os requisitos do sistema.

3.2. Requisitos funcionais da aplicação

Para definir uma melhor visualização dos requisitos do sistema Acadêmico, foi elaborado o Diagrama de Casos de Uso mostrado na figura 3.1.

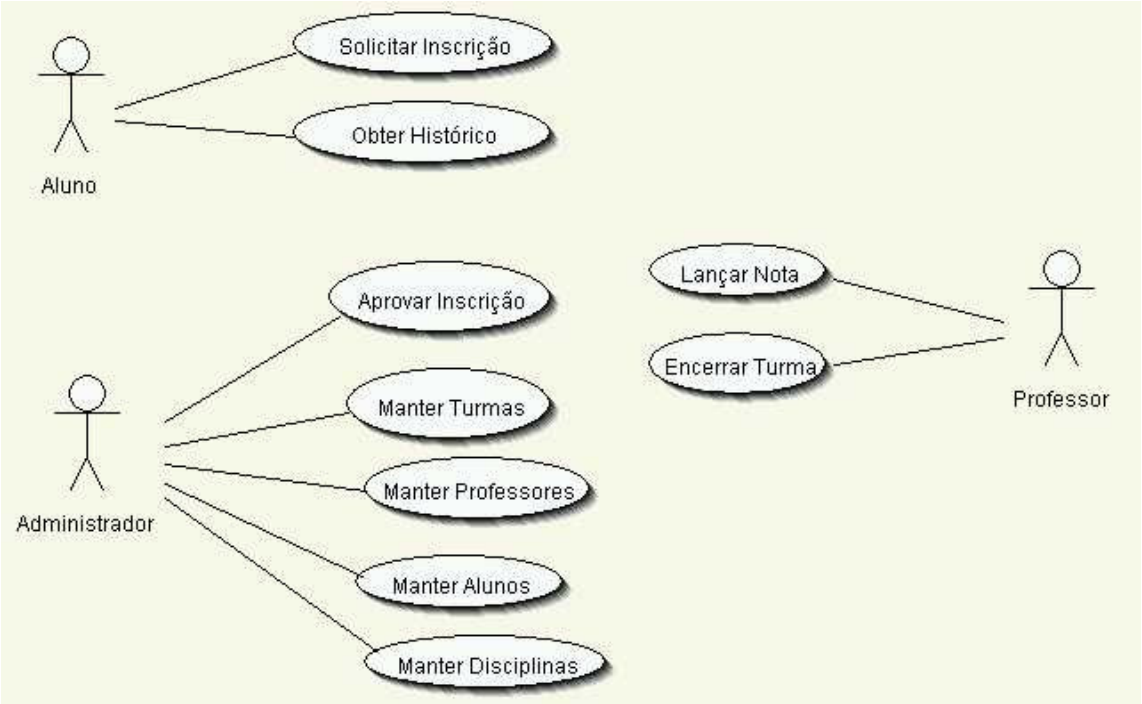


Figura 3.1 Diagrama de Casos de Uso do Sistema Acadêmico

A seguir são explicados com detalhes cada um dos casos de uso do sistema Acadêmico:

- Caso de Uso: Manter Disciplinas

Responsável: Administrador

Descrição: envolve a inclusão, alteração e exclusão das disciplinas a serem aplicadas em um determinado período, podendo abranger vários períodos.

- Caso de Uso: Manter Alunos

Responsável: Administrador

Descrição: envolve a inclusão, alteração e exclusão de alunos que cursam as disciplinas oferecidas pela instituição, através de turmas, durante um período de vigência da inscrição do aluno.

- Caso de Uso: Manter Professores

Responsável: Administrador

Descrição: envolve a inclusão, alteração e exclusão de professores que lecionam disciplinas oferecidas pela instituição. Um professor poderá lecionar em uma ou mais turmas de disciplinas diferentes. Não é escopo do sistema informatizado registrar quais disciplinas o professor está apto a lecionar.

- Caso de Uso: Manter Turmas

Responsável: Administrador

Descrição: envolve a inclusão, alteração e exclusão de turmas que aplicam uma e somente uma disciplina. A turma ocorre em um período letivo definido e é aplicada por um professor da instituição. O administrador também determina, de forma livre, os dias, horários e salas da instituição onde serão lecionadas as aulas das turmas. O sistema informatizado não contemplará questões sobre alocação na grade de horário ou utilização de ambiente físico.

- Caso de Uso: Solicitar Inscrição

Responsável: Aluno

Descrição: o sistema exibe ao usuário uma lista de turmas com inscrições em aberto e cujas disciplinas ainda não foram cursadas pelo solicitante. O aluno, então, submete ao sistema a solicitação de inscrição em uma turma.

- Caso de Uso: Aprovar Inscrição

Responsável: Administrador

Descrição: o sistema exibe ao Administrador uma lista de solicitações de inscrições de alunos em disciplinas. O Administrador, após a análise dessa solicitação, aceita ou exclui do sistema essa solicitação. Caso aceite, o aluno está regularmente inscrito na instituição para cursar a disciplina.

- Caso de Uso: Lançar Nota

Responsável: Professor

Descrição: o sistema exibe ao Professor todas as turmas em que ele está lecionando no período letivo em curso. O Professor seleciona uma das turmas e o sistema exibe a lista de alunos inscritos nessa turma. O Professor seleciona um aluno e pode então lançar uma nota, atribuindo à avaliação um código de identificação.

- Caso de Uso: Obter Histórico

Responsável: Aluno

Descrição: sistema exibe ao Aluno uma lista de ocorrências, onde cada ocorrência indica uma data e disciplina que o aluno cursou e a situação final de avaliação que o aluno obteve. As situações finais possíveis são Aprovado – média final maior ou igual a 6 - ou Reprovado – média final menor que 6.

- Caso de Uso: Encerrar Turma

Responsável: Professor

Descrição: sistema exibe ao Professor a lista de turmas na qual ele está lecionando. O Professor solicita ao sistema então que encerre uma determinada turma. O sistema calcula a situação final de cada aluno, baseado nas notas obtidas por ele, e atribui as situações Aprovado – média das notas maior ou igual a 6 – ou Reprovado – média das notas menor que 6. A turma fica registrada no sistema como encerrada e passa a fazer parte do histórico dos alunos que cursaram aquela turma.

3.3. Modelo PIM

O modelo PIM é a base para especificação do sistema usando a abordagem MDA. O modelo mostrado na figura 3.2, escrito em UML, é o modelo PIM do sistema exemplo Acadêmico. A ferramenta utilizada para criar o modelo foi o *plug-in* EclipseUML (Apêndice C). O EclipseUML é um *plug-in* do Eclipse que permite a criação de modelos UML. Essa ferramenta já suporta um dos padrões relacionados ao MDA apresentados na seção 2.5.3.2 deste trabalho: os *UML Profiles*.

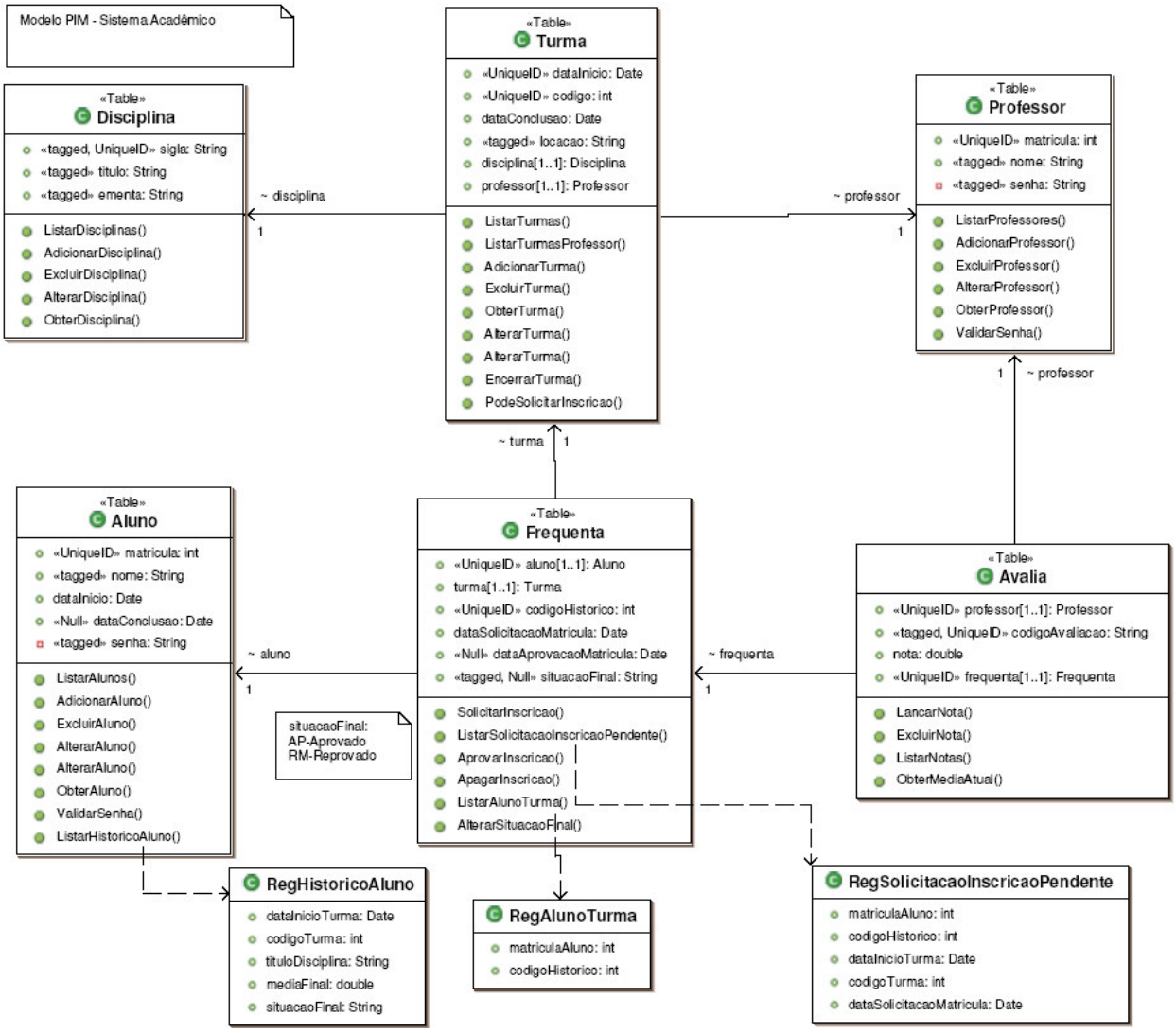


Figura 3.2 Modelo PIM do Sistema Acadêmico

Embora o modelo apresentado pareça meramente um diagrama de classes UML empregado em diversos projetos de *software*, a convergência com a prática MDA se encontra no uso indireto da linguagem MOF, embasando os elementos sintáticos do modelo PIM. É o emprego do MOF que fornecerá subsídios para a execução das transformações necessárias para geração de código estrutural e executável do sistema.

O modelo mostrado anteriormente não exibe a interface (parâmetros de entrada e de retorno) dos métodos definidos nas classes. A seguir mostraremos com detalhes cada uma das classes do modelo completo.

- Classe Disciplina

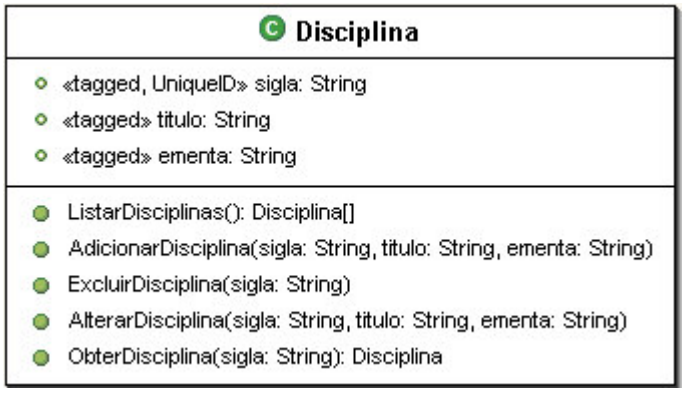


Figura 3.3 Classe Disciplina, PIM Acadêmico

Marcadores utilizados:

- A classe possui o estereótipo *<<Table>>*;
- O atributo *sigla* possui o estereótipo *<<UniqueID>>*;
- O atributo *sigla* possui o tagged value Size com o valor 10;
- O atributo *titulo* possui o tagged value Size com o valor 40;
- O atributo *ementa* possui o *tagged value Size* com o valor 8000;

- O atributo *ementa* possui o estereótipo *<<Null>>*.

- Classe Professor

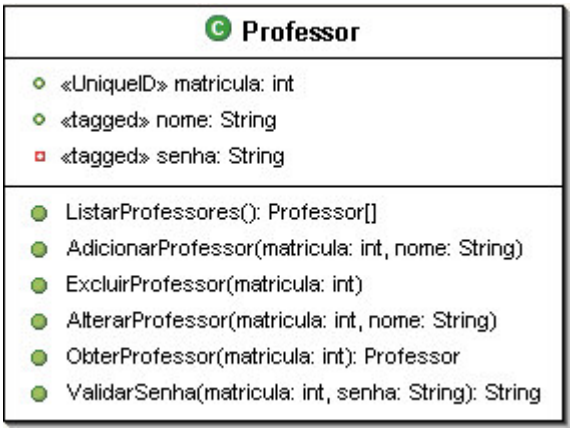


Figura 3.4 Classe Professor, PIM Acadêmico

Marcadores utilizados:

- A classe possui o estereótipo *<<Table>>*;
- O atributo nome possui o *tagged value Size* com o valor 80;
- O atributo senha possui o *tagged value Size* com o valor 12.

- Classe Turma

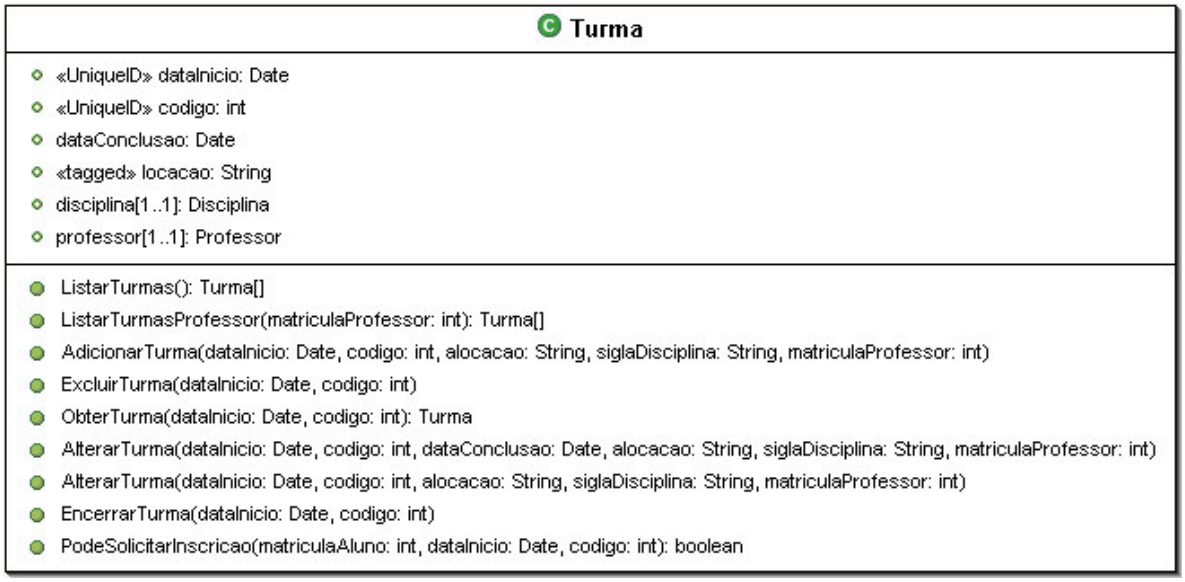


Figura 3.5 Classe Turma, PIM Acadêmico

Marcadores utilizados:

- A classe possui o estereótipo *<<Table>>*;
- O atributo *dataInicio* possui o estereótipo *<<UniqueID>>*;
- O atributo *codigo* possui o estereótipo *<<UniqueID>>*;
- O atributo *locacao* possui o tagged value Size com o valor 8000;
- O atributo *locacao* possui o estereótipo *<<Null>>*.

- Classe Aluno

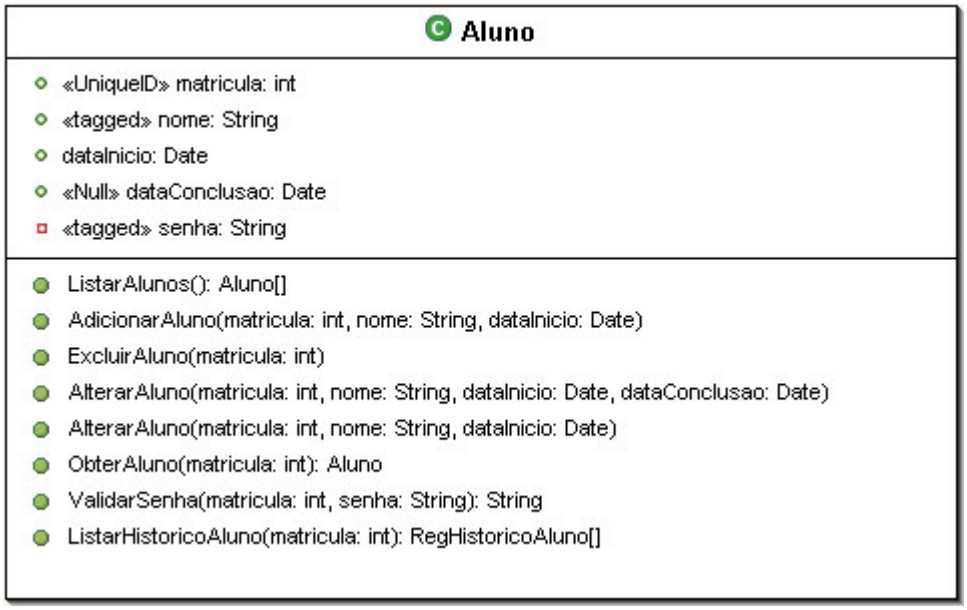


Figura 3.6 Classe Aluno, PIM Acadêmico

Figura 3.7 Estrutura RegHistoricoAluno, PIM Acadêmico

Marcadores utilizados:

- A classe possui o estereótipo *<<Table>>*;
- O atributo *matricula* possui o estereótipo *<<UniqueID>>*;
- O atributo *nome* possui o *tagged value Size* com valor 80;
- O atributo *dataConclusao* possui o estereótipo *<<Null>>*;
- O atributo *senha* possui o *tagged value Size* com valor 12.

- Classe Frequenta

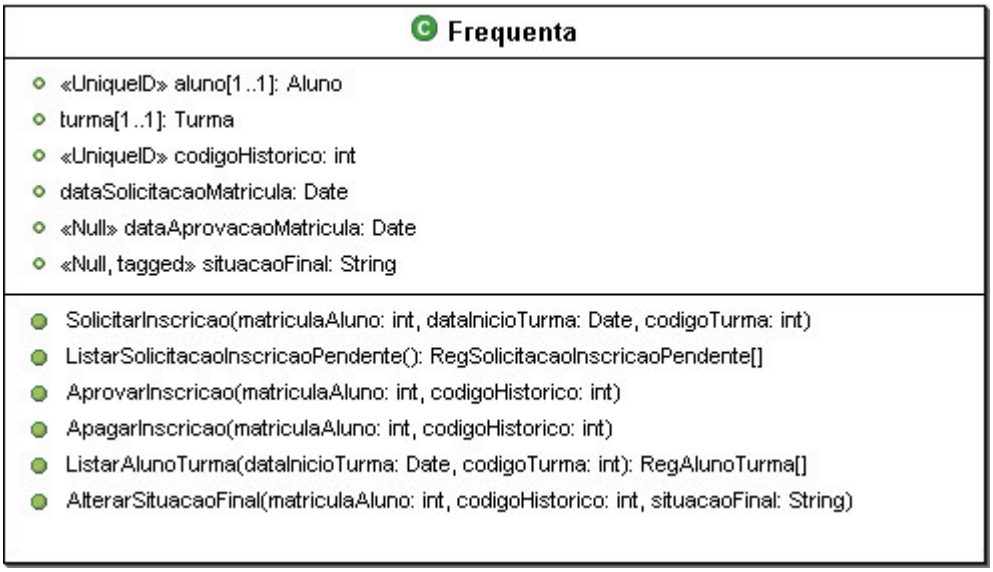


Figura 3.8 Classe Frequenta, PIM Acadêmico

Figura 3.9 Estrutura RegSolicitacaoInscricaoPendente, PIM Acadêmico

Figura 3.10 Estrutura RegAlunoTurma, PIM Acadêmico

Marcadores utilizados:

- A classe possui o estereótipo *<<Table>>*;

- O atributo *aluno* possui o estereótipo *<<UniqueID>>*;
- O atributo *codigoHistorico* possui o estereótipo *<<UniqueID>>*;
- O atributo *dataAprovacaoMatricula* possui o estereótipo *<<Null>>*;
- O atributo *situacaoFinal* possui o *tagged value Size* com o valor 2;
- O atributo *situacaoFinal* possui o estereótipo *<<Null>>*.

- Classe Avalia

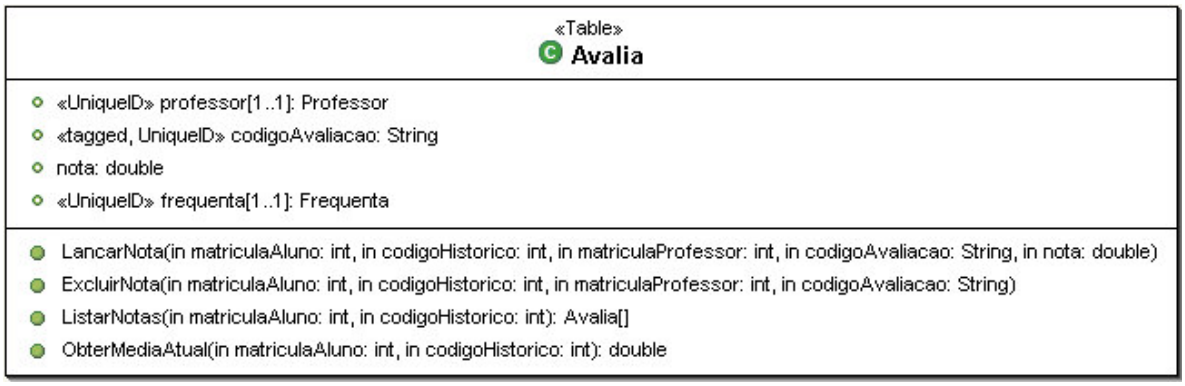


Figura 3.11 Classe Avalia, PIM Acadêmico

Marcadores utilizados:

- A classe possui o estereótipo <<Table>>;
- O atributo *professor* possui o estereótipo <<UniqueID>>;
- O atributo *codigoAvaliacao* possui o estereótipo *<<UniqueID>>*;
- O atributo *codigoAvaliacao* possui o *tagged value Size* com o valor 10;
- O atributo *frequenta* possui o estereótipo *<<UniqueID>>*;

3.4. Arquitetura de Implementação da Aplicação

O sistema Acadêmico será implementado em uma arquitetura comum nos sistemas atuais, em três camadas: camada de apresentação, camada de negócios e camada de persistência. Essa separação em três camadas promove uma melhor separação de interesses, facilitando a manutenção e reutilização dos artefatos de *software*. A figura 3.12 ilustra essa arquitetura.

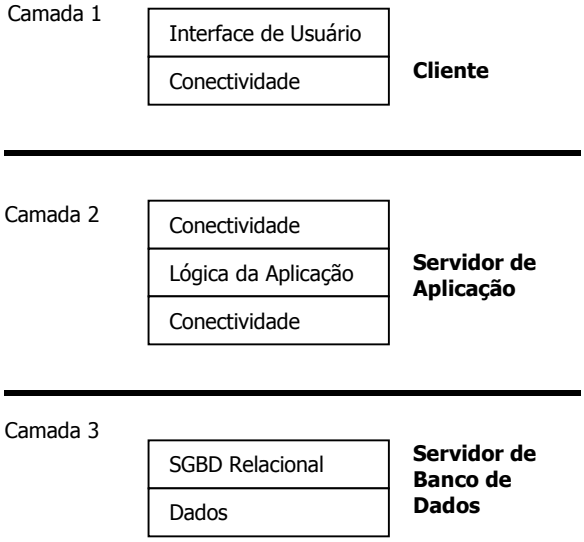


Figura 3.12 Arquitetura em Três Camadas

Aplicando as práticas do MDA ao modelo PIM anteriormente descrito, deveremos realizar, no mínimo, três transformações, uma para cada camada da aplicação: código de apresentação, código de lógica de negócio e código para definir a estrutura do banco de dados. A figura 3.13 mostra as transformações necessárias para gerar os artefatos de cada uma das camadas para executar o sistema exemplo Acadêmico.

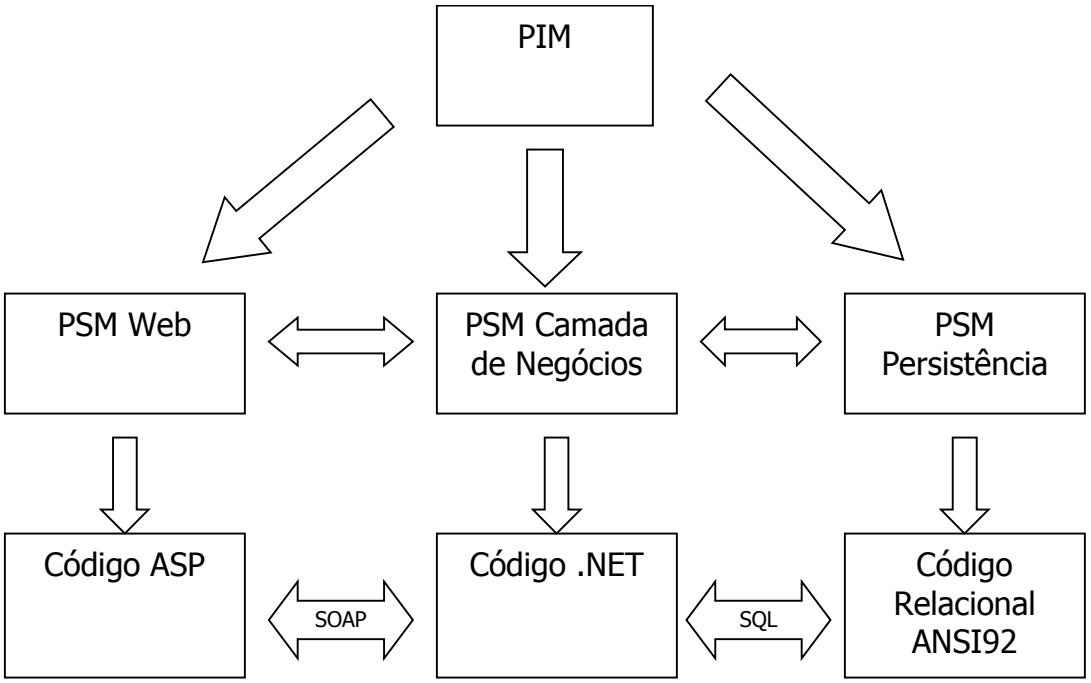


Figura 3.13 Transformações do Sistema Acadêmico

Com o propósito de aplicar integralmente as práticas do MDA no sistema exemplo e limitar o escopo deste trabalho, faremos algumas escolhas de implementação. Embora caiba ressaltar que a grande utilidade do MDA está justamente na adaptabilidade de implementação em várias plataformas, as escolhas ora apresentadas visam possibilitar a geração de um sistema executável usando opções empregadas amplamente em sistemas da atualidade. Além disso, o emprego de padrões amplamente utilizados, como será o caso em estudo, ajuda a aumentar a longevidade do sistema uma vez que várias implementações que adotem o mesmo padrão poderão vir a ser empregadas.

Nas transformações aplicadas serão gerados dois *bridges* – interfaces entre modelos – que obedecem a padrões consagrados: SOAP – para suporte a invocações de serviços remotos - e SQL – para acesso a banco de dados relacionais. Essa decisão de projeto, além de facilitar a geração dos referidos *bridges*, possibilita a eventual substituição da plataforma de uma das camadas sem que seja necessária a regeração do *bridge* envolvido, contanto que a camada recém aplicada obedeça às mesmas restrições dele.

Como exemplo, podemos citar o *software* de suporte a persistência relacional: poderíamos aplicar qualquer produto relacional que obedeça às especificações SQL ANSI´92 [DEC2002], sem necessidade de geração do *bridge* entre as camadas de negócio e de código relacional.

A seguir discutimos as escolhas de implementação para cada uma das camadas:

- Camada Web

A escolha de implementar a camada de interface com usuário usando uma típica interação Web segue uma tendência forte no desenvolvimento atual de aplicações.

No contexto de desenvolvimento de interfaces, as primeiras aplicações usavam interfaces de texto de difícil interação com o usuário. As aplicações gráficas de janelas surgiram no final da década de 80 e trouxeram grandes avanços nas possibilidades de recursos para facilitar o uso pelo usuário final. Em meados da década de 90, assistimos a uma explosão de uso da Internet e suas aplicações acessadas via navegadores. Embora sejam menos flexíveis em recursos que as aplicações gráficas de janelas, as aplicações *Web* tiram proveito da larga utilização por uma base de usuários sem precedentes na história da computação.

A plataforma de implementação escolhida foi a ASP.NET [Asp2004], usando como linguagem de programação C#. O código completo da aplicação pode ser examinado no relatório técnico em http://www.ic.uff.br/~mbelo/dissert/RelTecn.pdf.

- Camada de Negócios

Nessa camada são implementadas as regras de negócio. A linguagem escolhida foi a C#, implementando Web Services [Wsa2004]. A vantagem de encapsular as

funções de negócio em componentes Web Services estão na facilidade para integração dessa camada com a camada de interface.

Web Services [Wsa2004] é uma nova tecnologia para disponibilização de procedimentos remotos tirando proveito de tecnologias consagradas pela Internet. Usando um padrão como esse, tiramos melhor proveito da adaptabilidade que a integração de modelos patrocinada pelo MDA pode oferecer. Embora uma ferramenta que empregue o MDA possa ser capaz de gerar *bridges* que forneçam a interoperabilidade entre duas tecnologias distintas, o uso de padrões consagrados facilita o alcance desse objetivo.

- Camada de persistência

Embora já existam bancos de dados orientados a objetos, a escolha mais comum para persistência de dados nas aplicações atuais ainda está nos bancos de dados relacionais.

Existem diversos *softwares* que implementam banco de dados relacionais. Nossa escolha baseou-se na adoção do padrão SQL ANSI´92 [DEC2002] pela ampla aceitação desse padrão de linguagem entre os vários produtos concorrentes.

Nossa escolha ficou pelo produto Firebird, um banco de dados de uso gratuito e código aberto. O apêndice D deste documento aborda algumas características desse produto.

3.5. Transformações

3.5.1. Introdução

Nosso propósito neste tópico será descrever cada uma das três principais transformações necessárias para a geração dos códigos de implementação do sistema Acadêmico, tanto a parte estrutural quanto a dinâmica. Conforme citado em [Fra2003], a independência preconizada pelo modelo PIM na abordagem MDA é relativizada pelo contexto que definimos. Por exemplo, quando CORBA foi concebido, era um padrão considerado independente de plataforma. Hoje, quando novas opções de *middleware* para chamadas remotas se apresentam, como Java/RMI e *Web Services*, o CORBA pode ser percebido como mais uma opção e, portanto, ele próprio considerado uma plataforma. Para resolver essa questão, devemos definir de quais aspectos nosso sistema deverá ser independente. No caso do sistema acadêmico, a independência se refere a:

- Linguagem dinâmica para interação Web com o usuário na camada de apresentação;
- Plataforma de suporte e implementação aos *Web Services* na camada de negócios;
- Produto de banco de dados relacional que suporte o padrão ANSI na camada de persistência.

Nenhum dos códigos de definição de transformações que serão apresentados a seguir tem a pretensão de estar 100% correto visto a ausência de ferramentas automatizadas que possam verificar a consistência dele. Num ambiente MDA real, as ferramentas de construção desses códigos terão depuradores de sintaxe embutidos, como ocorre com as ferramentas IDE usadas atualmente.

Nos tópicos a seguir cada uma das transformações serão detalhadas.

3.5.2. Transformação PIM UML para Modelo Relacional

Para realizar a transformação PIM para PSM Relacional do sistema Acadêmico, fizemos uso de um *UML Profile* especialmente definido para esse propósito. Nossa intenção foi criar uma variação da UML (através de *profiles*) para uma modelagem em alto nível de abstração.

Temos a percepção de que o modelo de classes UML já supre a maior parte das necessidades para geração de um modelo relacional completo. Tanto a representação abstrata (*Abstract Syntax*) quanto a concreta (*Concrete Syntax*) do modelo de classes UML favorecem a especificação de modelos relacionais.

Entretanto, existem pequenas lacunas para especificação de um sistema relacional completo que devem ser supridas. Como foi discutido no item 2.5.3 deste trabalho, poderíamos resolver tais questões usando duas abordagens: a definição de um metamodelo específico para modelos relacionais – o chamado *Heavyweight Metamodel*, ou a extensão da linguagem UML através de *Profiles*. Esta última opção nos parece muito mais adequada devido à facilidade na utilização e especificação dessa extensão.

No capítulo 4, abordaremos a transformação relacional com maior detalhamento, citando outras abordagens existentes para esse mesmo cenário em MDA.

3.5.3. Transformação PIM UML para Modelo .NET

Nossa intenção, neste caso de transformação, é gerar, a partir do modelo PIM, código .NET [Net2004] que implemente como *Web Services* [Wsa2004] as funcionalidades de negócio definidas para o sistema. Vale ressaltar que poderíamos escolher qualquer outra plataforma semelhante como, por exemplo, Java. Como o próprio modelo UML já se assemelha muito aos modelos utilizados para descrever classes e métodos de um modelo de implementação, sejam eles modelos J2EE ou modelos .NET (PSM), essa

transformação pode ser dita como um caso no qual a geração de um modelo PSM intermediário, para posterior geração do código de implementação, será necessário apenas por motivos de documentação. Muitas ferramentas de transformação tornarão transparente para os desenvolvedores a geração de um modelo PSM intermediário.

Pelos motivos expostos acima, a linguagem de transformação utilizada nesse caso será muito simples, pois o nível de abstração do modelo PIM para o código não será tão diferente.

A seguir as principais regras de transformação são colocadas em forma textual:

1. Para cada classe pública do modelo PIM, gere uma classe no modelo PSM de mesmo nome, classe essa derivada da classe *System.Web.Services.WebService*;

2. Para cada atributo do modelo PIM, gere um atributo público na classe correspondente do modelo PSM;

3. Para cada classe pública no modelo PIM, gere uma estrutura pública com o prefixo 'Reg' seguido do nome da classe de origem. É interessante notar que essa estrutura visa implementar algo semelhante ao *Design Pattern Value Object* do J2EE;

4. Para cada atributo público, gere um atributo público na estrutura gerada;

5. Para cada operação declarada como pública do modelo PIM, gere um método na classe correspondente e publique esse método como um *Web Service*, transformando os tipos de dados dos parâmetros e do tipo de retorno, quando os tipos utilizados forem os do padrão UML, para os tipos de dados do padrão .NET;

6. Para cada operação privada, gere um método privado na classe correspondente transformando os tipos de dados dos parâmetros e do tipo de retorno, quando os tipos utilizados forem os do padrão UML, para os tipos de dados do padrão .NET;

Como exposto no capítulo 2.5, para que seja possível formalizar as regras de transformação num código chamado definição da transformação, precisamos dos respectivos meta-modelos dos modelos de origem e do modelo de destino. Nesse caso específico de transformação, nosso meta-modelo de origem é o meta-modelo UML apresentado na figura 2.6.

Como o meta-modelo, compatível com o padrão MOF, que formaliza a linguagem do modelo de *Web Services* em .NET é muito semelhante ao modelo do diagrama de classes UML, omitiremos sua completa especificação.

Usando os referidos meta-modelos, apresentamos a seguir a definição das transformações necessárias para a geração de código .NET a partir no modelo PIM, usando a linguagem não padronizada apresentada em [KWB2003].

A regra do código 3.1 implementa a definição formal da transformação descrita pela regra textual especificada no item 1.

```
Transformation UMLClassToNetClass(UML,NET)
{
source
 class:      UML::Class;
target
 netClass:   NET::NetClass;
mapping
 class.name         <~>    netClass.name;
 class.attributes() <~>    netClass.attributes();
 class.operations() <~>    netClass.operations();
}
```

Código 3.1 Transformação UMLClassToNetClass

A definição do código 3.2 formaliza a regra 2. Nesse código, existe uma conversão de tipos da UML para tipos .NET. Essa transformação dispara, como conseqüência, a definição de transformação definida no código 3.3, que visa apenas estabelecer que existe a necessidade de transformar um tipo definido em UML para o tipo correspondente em .NET. Os detalhes dessa transformação foram omitidos por motivos de simplificação.

```
Transformation UMLAttributeToNetAttribute(UML,NET)
{
source
 umlAttribute:      UML::Attribute;
target
 netAttribute:      NET::Attribute;
target condition
 umlAttribute.class = netAttribute.class;
mapping
 umlAttribute.name         <~>    netAttribute.name;
 umlAttribute.type         <~>    netAttribute.type;
}
```

Código 3.2 Transformação UMLAttributeToNetAttribute

```
Transformation UMLDataTypeToNetDataType (UML,NET)
{
source
 umlDataType        : UML::DataType;
target
 netDataType        : NET::DataType;
bidirectional;
}
```

Código 3.3 Transformação UMLDataTypeToNetDataType

O código 3.4 define a regra 3. A estrutura gerada (definida no meta-modelo .NET) irá permitir que, em invocações remotas, a estrutura seja o meio de transporte para todos os dados de uma instância da classe. Essa técnica evita chamadas remotas individuais para obter cada um dos valores de propriedade de uma instância remota, diminuindo o tráfego na rede de dados.

```
Transformation UMLClassToNetStructure(UML,NET)
{
source
 class:      UML::Class;
```

```
target
 netStruct:  NET::NetStructure;
mapping
 class.name  <~>    'Reg' + netStruct.name;
}
```

Código 3.4 Transformação UMLClassToNetStructure

O código 3.5 define a regra 4 e complementa a transformação definida no código 3.4, gerando as propriedades para a estrutura. Essa transformação também acionará a transformação definida no código 3.3 para conversão de tipos UML para tipos .NET.

```
Transformation UMLAttributeToNetStructureAttribute(UML,NET)
{
source
 umlAttribute:      UML::Attribute;
target
 netAttribute:      NET::Attribute;
source condition
 umlAttribute.visibility = Public
target condition
 umlAttribute.class = netAttribute.class;
mapping
 umlAttribute.name        <~>    netAttribute.name;
 umlAttribute.type        <~>    netAttribute.type;
}
```

Código 3.5 Transformação UMLAttributeToNetStructureAttribute

O código 3.6 define a regra 5. Nesse código existe uma condição sobre o modelo de origem indicando que a transformação só ocorrerá para operações declaradas como públicas. Cada execução dessa regra também acionará o código 3.3 para conversão de tipos e para a transformação de parâmetros. Para esse último caso, a geração de parâmetros segue o mesmo padrão para transformação de atributos, como o do código 3.5; portanto, o omitimos por questão de simplificação.

```
Transformation UMLOperationToNetOperation(UML,NET)
{
source
 umlOperation       : UML::Operation;
target
 netOperation       : NET::Operation;
source condition
 umlOperation.visibility = VisibilityKind::public
bidirectional;
mapping
```

```
 umlOperation.name         <~> netOperation.name;
 umlOperation.type         <~> netOperation.type;
 umlOperation.parameters() <~> netOperation.parameters();
}
```

Código 3.6 Transformação UMLOperationToNetOperation

Omitimos a regra 6 por ela se assemelhar com o código 3.6, com exceção do fato de ser aplicada apenas a operações privadas, o que mudaria o *VisibilityKind* do código de *public* para *private*.

Como exemplo de aplicação das definições de transformação, será apresentada a transformação da classe *Professor* na figura 3.14 do modelo Acadêmico.

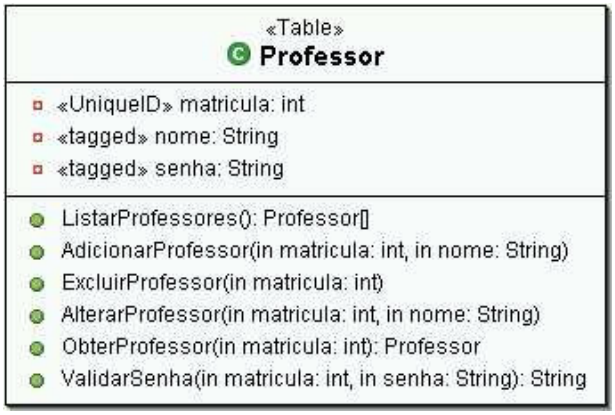


Figura 3.14 Fragmento Professor do Modelo PIM do Acadêmico

Aplicando a transformação *UMLClassToNetClass*, o código 3.7 será gerado.

```
public class Professor : System.Web.Services.WebService
{

}
```

Código 3.7 Resultado da Transformação UMLClassToNetClass

Aplicando a regra *UMLClassToNetStructure*, teremos algo bem parecido com o que aconteceu na transformação de classes, exceto que agora será gerada uma estrutura .NET. O código 3.8 mostra o resultado dessa transformação.

```
public struct RegProfessor
{

}
```

Código 3.8 Resultado da Transformação UMLClassToNetStructure

Complementando a estrutura anterior, a execução da regra *UMLAttributeToNetStructureAttribute* gerará o código 3.9. Podemos reparar que o atributo *senha*, por ter visibilidade privada, não faz parte da estrutura *RegProfessor* gerada.

```
public struct RegProfessor
{
 public int matricula;
 public string nome;
}
```

Código 3.9 Resultado da Transformação UMLAttributeToNetStrucutreAttribute

Finalmente, a aplicação da regra *UMLOperationToNetOperation* irá gerar os métodos que implementam os *WebServices*, gerando o código 3.10. Podemos reparar que a transformação citada irá, por conseqüência, disparar a transformação dos parâmetros da operação, definida pelo código 3.10.

```
[WebMethod] public RegProfessor[] ListarProfessores()
{

}

[WebMethod] public void AdicionarProfessor(int matricula,string nome)
{

}

[WebMethod] public void ExcluirProfessor(int matricula)
{

}
```

```
[WebMethod] public void AlterarProfessor(int matricula,string nome)
{

}

[WebMethod] public RegProfessor ObterProfessor(int matricula)
{

}

[WebMethod] public string ValidarSenha(int matricula,string senha)
{

}
```

Código 3.10 Resultado da Aplicação das Transformações sobre Operações

Fragmentos do código da transformação PIM para código .NET é apresentado no apêndice A (Camada de Negócios) deste documento. O código completo gerado pode ser encontrado em http://www.ic.uff.br/~mbelo/dissert/RelTecn.pdf.

3.5.4. Transformação PIM UML para Modelo Web

O modelo que nos interessa gerar com essa transformação é um modelo PSM que descreva a camada de interface de interação do usuário com a aplicação. Desenho de interfaces constitui uma tarefa que exige, além do emprego de técnicas adequadas, alta dose de criatividade e capricho que são impossíveis de serem alcançadas com uma simples receita de bolo. Além disso, a percepção de qualidade da interface pode variar muito dependendo do gosto ou experiência anterior do usuário.

Atualmente, é forte a tendência ao uso de interfaces *Web* para aplicações voltadas para os mais variados propósitos. Apesar de oferecer menos recursos que as já consagradas interfaces de janelas, as interfaces *Web* são favorecidas pela rápida popularização no uso de aplicativos pela Internet com interfaces acessadas via navegadores.

Apesar da criatividade humana ter um papel fundamental na qualidade dos modelos de interface gerados, as automatizações patrocinadas pelo MDA podem ajudar muito na obtenção de uma boa parte das interfaces que a aplicação irá necessitar.

A abordagem apresentada em [KWB2003], embora exposto pelo autores que intenciona a completa geração de uma interface real, nos parece bastante simplicista frente às reais necessidades esperadas de um modelo de interface. O trabalho citado apresenta a transformação do modelo PIM para o modelo PSM Web – tipo de interface escolhida pelos autores – como semelhante à transformação dos artefatos da camada de negócio. No nosso ponto de vista, o produto resultante desse tipo de transformação seria incipiente demais para uso real.

Pensamos que, para uma efetiva modelagem de interface em alto nível de abstração, necessitaremos de uma linguagem de modelagem apropriada para tal intento. Esta linguagem apresentaria conceitos como página, componentes de interação (caixas

de textos, caixas de seleção, botões, etc.), transições entre páginas, entre outros. As transições, quando condicionais, poderiam ser expressas em OCL.

Como o propósito do presente trabalho não é aprofundar esse tipo de transformação, a geração das interfaces do sistema Acadêmico foi completamente desenhada utilizando uma ferramenta IDE apropriada. As interfaces utilizadas no sistema Acadêmico são apresentadas no Apêndice B.

3.6. Conclusão

Apesar da simplicidade do sistema Acadêmico, este estudo de caso nos oferece uma visão do processo MDA aplicado a um cenário de transformação típico para sistemas informatizados da atualidade.

As transformações, como vimos anteriormente, possuem uma importância vital dentro do processo MDA. Quanto mais automatizada for a transformação, ou seja, quanto mais código executável pronto for gerado a partir do modelo independente PIM, maior será o valor percebido. Outras vantagens, como a portabilidade para novas tecnologias, são menos tangíveis e só poderão ser percebidas no futuro.

4. TRANSFORMAÇÃO RELACIONAL

4.1. Introdução

Muitas aplicações na atualidade utilizam, como método primordial de persistência de dados, produtos de banco de dados relacionais. Embora já existam bancos totalmente orientados a objetos, a realidade no mercado força com que nossos sistemas orientados a objetos tenham que conviver com bancos relacionais. É justamente nessa necessidade de convivência de tecnologias diferentes que o padrão MDA pode mostrar seu valor no tocante à interoperabilidade.

Embora o padrão MDA deva encontrar aprimoramentos na padronização das transformações PIM para modelos PSMs voltados para banco de dados relacionais, já existem grupos de trabalho voltados para definir meta-modelos e códigos de definição de transformações voltadas para esse propósito. Uma dessas iniciativas é o padrão CWM [Cwm2003].

Nesse capítulo detalharemos o motivo da definição da transformação e escolha de um *Profile* especificamente desenvolvido para este trabalho.

4.2. Meta-modelos e outras propostas existentes

A própria OMG definiu um conjunto de meta-modelos voltado para persistência de dados, reconhecendo a importância desses modelos na maioria dos projetos de sistemas. O CWM [Cwm2003] padroniza vários meta-modelos, escritos em MOF, que definem linguagens para criação de modelos de dados. Entretanto, a OMG não se limitou a definir linguagens apenas para modelos relacionais. O CWM contém meta-modelos para os seguintes modelos de dados:

- Relacionais;
- Multidimensionais, aplicados em sistemas de *datawarehouse*;
- Estruturas de registros;
- XML.

Além disso, o CWM define meta-modelos para descrever regras de transformações de dados, como por exemplo regras para sistemas OLAP (*Online Analytical Processing*), *data-mining* e outras regras em geral. Esses meta-modelos não são de nosso interesse nesse trabalho.

Como nossa atenção no sistema exemplo está voltada para a criação de um modelo relacional, o meta-modelo para esse propósito definido pelo CWM será de vital importância. A figura 4.1, retirada de [KWB2003], é uma simplificação do meta-modelo relacional do CWM.

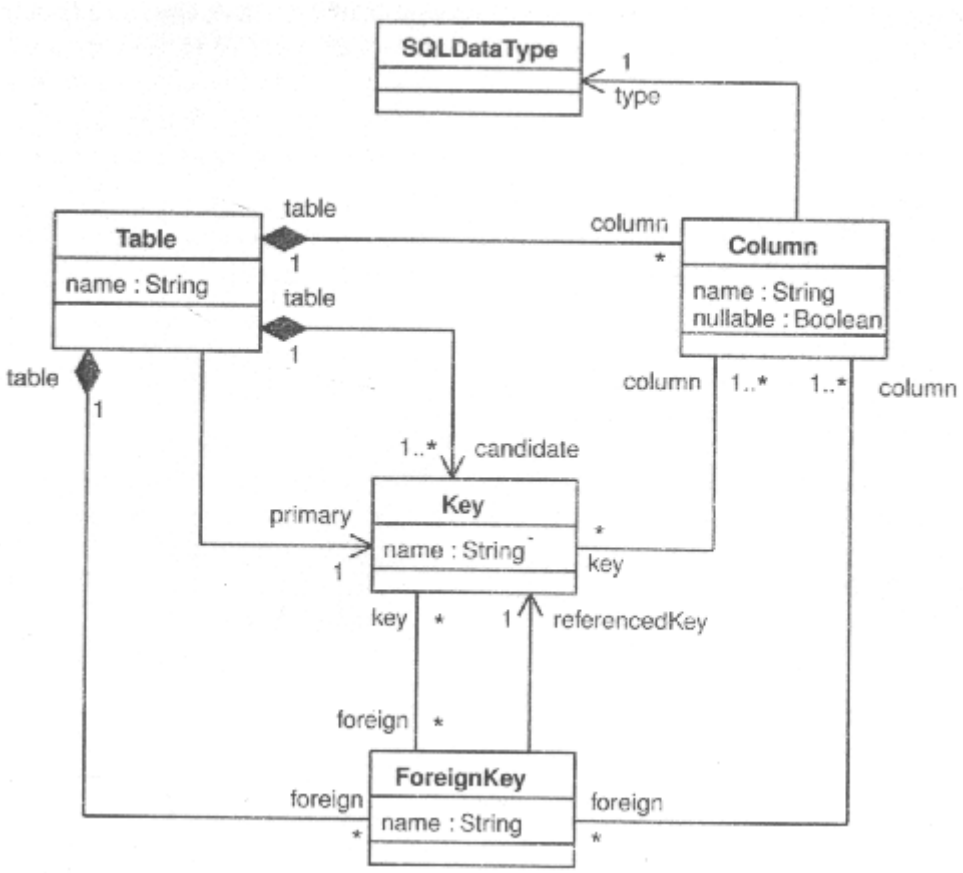


Figura 4.1 Meta-Modelo CWM para Modelos Relacionais

Para fundamentar as decisões tomadas para realização da transformação relacional feitas neste trabalho, analisamos duas alternativas *UML Profiles* que objetivam o mesmo contexto de aplicação. São eles:

The UML and Data Modeling

Essa proposta da Rational, apresentada em [UDM2000], foi definida para utilização na ferramenta da respectiva empresa (Rational Rose Data Modeler) e, talvez por isso, não explicita detalhes para sua adoção de forma aberta. Como esse *profile* atua em nível de PSM na complementação da descrição física de um banco de dados, conceitos como *database*, *schema*, *key*, *index*, *constraint*, *trigger*, etc. fazem parte de sua especificação.

A Rational fundamenta a utilização desse *profile* através do argumento de que muitos sistemas continuarão a serem modelados seguindo a visão orientada a dados, ou seja, a partir dos requisitos levantados, o projetista irá primeiramente elaborar um modelo de dados que contemple toda a necessidade de persistência requerida. Essa visão da Rational para projeto de software sugere uma preocupação antecipada com detalhes de implementação que destoam daquelas intencionadas pelo MDA. Ainda segundo a Rational, a melhor maneira de elaborar tal modelo é usando a linguagem UML apoiada pelo *profile* em questão.

UML Profile for Data Modeling

Embora não exista um padrão de fato no mercado para um *profile*, o *UML Profile for Data Modeling* proposto em [Amb2003] pela *Agile Modeling* é atualmente o que apresenta maior aceitação entre práticas MDA. O *profile* propõe três tipos de modelos de dados, graduados por nível de abstração e cada um deles devendo ser representado por um estereótipo adequado. Os tipos de modelos são:

- Modelo Conceitual de Dados, representado pelo estereótipo *Conceptual Data Model*;
- Modelo Lógico de Dados, representado pelo estereótipo *Logical Data Model*; e
- Modelo Físico de Dados, representado pelo estereótipo *Physical Data Model*.

Esses três tipos de modelos são classicamente usados em modelagem de dados estruturadas. A proposta Agile é incorporar cada um desses modelos no conjunto de artefatos de modelagem da especificação da aplicação.

Além dessa segregação de nível de modelagem, o *profile* ainda define vários tipos de tecnologia para persistência de dados, cada um deles sendo representado por um estereótipo. A seguir apresentamos os principais:

- Arquivo, representado pelo estereótipo *File*;
- Organização de Dados Hierárquica, representado pelo estereótipo *Hierarquical Database*. Esse tipo de estrutura para banco de dados foi largamente usado na década de 70 e atualmente constitui um modelo legado;
- Bancos Orientados a Objetos, representado pelo estereótipo *Object-Oriented Database*. Esse tipo de banco é bastante recente e ainda não obteve aceitação do mercado em aplicações comerciais. Sua utilização tem se restringido a áreas de aplicação muito específicas;
- Banco de Dados Objeto-Relacional, representado pelo estereótipo *Object-Relacional Database*. Esse tipo de banco é híbrido com relação às abordagens relacionais e de objetos e ainda encontra pouca aceitação no mercado. Muitos produtos no mercado se enquadram nessa categoria, embora a utilização prática esteja limitada aos recursos relacionais;
- Banco Relacional, representado pelo estereótipo *Relational Database*.

Complementando a modelagem, o *profile* define uma série de estereótipos e *tagged values* para definir as características de organização do banco. Alguns dos estereótipos podem ser usados em mais de um dos modelos (conceitual, lógico ou físico). A tabela a seguir mostra alguns dos marcadores utilizados neste *profile*:

- *Associative Table*, usado no modelo físico, indica que a tabela existe pela derivação de um relacionamento N para N;
- *Entity*, usado nos modelos lógico e conceitual, indica explicitamente que a classe com ele marcado será uma entidade de persistência de dados;
- *Index*, usando no modelo físico, indica que a classe representa um índice gerado num banco relacional, tipicamente;

- *Stored Procedures*, usado no modelo físico, indica que as operações estáticas daquela classe gerarão procedimentos armazenados no banco;
- *View*, usado no modelo físico, permite criar visões de dados;
- *Identifying*, usado no modelo físico, indica que uma associação de uma instância é dependente da outra instância que ela referencia. Se a instância referenciada for eliminada, o mesmo deve acontecer com a instância que referencia;
- *Non-identifying*, usado no modelo físico, indica um relacionamento fraco, ao contrário do estereótipo *Identifying*;
- *AK*, usado no modelo físico, indica que o atributo/coluna pertence a uma chave alternativa da tabela gerada. Pode ser usado com o *tagged value order*, que indica a ordem que a coluna terá dentro do índice;
- *Auto Generated*, usado no modelo físico, permite indicar que uma ou mais colunas terão valores automaticamente gerados para uma instância da tabela, recurso que existe em vários produtos de banco de dados relacionais;
- FK, usado no modelo físico, indica que o atributo/coluna pertence a uma chave estrangeira da tabela. Pode ser usado com o *tagged value order*, que indica a ordem que a coluna terá dentro do índice que implementa essa chave estrangeira;
- *Not Null*, usado no modelo físico, indica que um atributo/coluna deve obrigatoriamente ter um valor especificado para uma instância do mesmo;
- *Nullable*, usado no modelo físico, indica que um atributo/coluna tem opcionalmente um valor especificado para uma instância do mesmo;

O documento [Amb2003], que propõe o *profile*, contém um estudo de caso mostrado na figura 4.2 que aplica alguns dos estereótipos explicados acima.

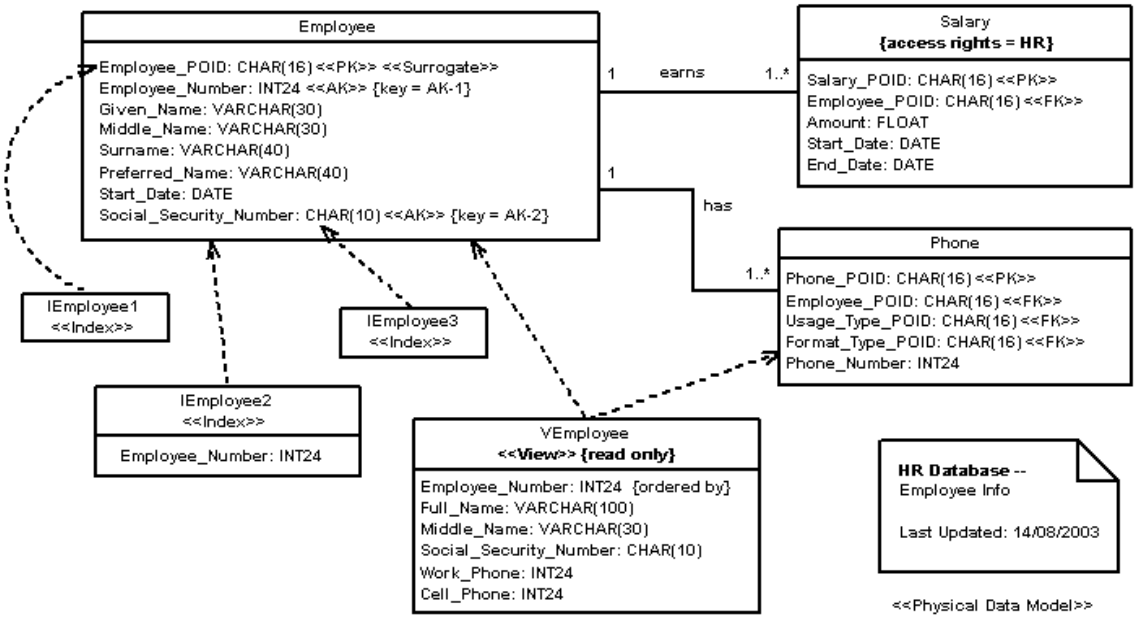


Figura 4.2 Modelo Físico de Dados aplicando o *UML Profile for Data Modelling*

No nosso entendimento, o uso de um modelo específico para modelagem de dados – ou vários, como da proposta da Agile – aumenta a quantidade de artefatos de modelagem e, por conseqüência, a complexidade do sistema sendo modelado. Muitas das representações do *profile* são facilmente derivadas de um modelo de classes UML em alto nível de abstração. No tópico seguinte apresentaremos um modelo de *profile* que, no nosso entendimento, simplifica a representação da abordagem PIM para modelo relacional em estudo neste capítulo.

4.3. UML Profile Proposto

4.3.1. Introdução

Neste trabalho propomos um *profile* que procura simplificar a quantidade de artefatos gerados na modelagem para banco de dados relacionais, baseado na especificação da Agile. A escolha do *UML Profile For DataModeling* da Agile como referência foi motivada por sua maior aceitação e por sua característica aberta, ou seja, não restringe seu uso a qualquer ferramenta específica, ao contrário da proposta da Rational. Esse *profile* foi desenhado para ser utilizado em nível de PIM, uma vez que define a semântica genérica capaz de gerar um modelo PSM relacional baseado no padrão ANSI´92 [DEC2002]. O estudo de caso será feito usando o SGBD Firebird (vide Apêndice D) que suporta o referido padrão.

Nosso ponto de vista favorece a utilização desse *profile* genérico em comparação com outras especificações que atuam em nível de PSM. A própria linguagem UML, utilizada no modelo PIM do Acadêmico e em diversos outros projetos baseados em MDA, fornece a maior parte das construções necessárias para representar em alto nível de abstração as definições de transformação envolvidas.

4.3.2. Definição do *Profile*

Para refinar nosso modelo, definimos um conjunto pequeno de estereótipos e *tagged values* mostrados na figura 4.3.

Tipo	Nome	Metaclasse MOF	Definição
Estereótipo	<<Table>>	Class	Gera um tabela no banco de dados
Estereótipo	<<UniqueID>>	Attribute	Torna-se a chave primária da tabela
Tagged Value	Size	Attribute	Indica o tamanho de uma *String*
Estereótipo	<<Null>>	Attribute	Indica que um atributo é do tipo nulo, ou seja, pode não apresentar valor para uma instância

Figura 4.3 Estereótipos e *Tagged Values* do Profile Relacional

Como em MDA representamos tudo por modelo, não seria diferente com relação ao *Profile* Relacional. O estereótipos e *tagged values* definidos acima serão na prática representados em um modelo de *profile*, que estendem o meta-modelo do diagrama de classes UML. A figura 4.3 mostra o modelo que define esse *profile*.

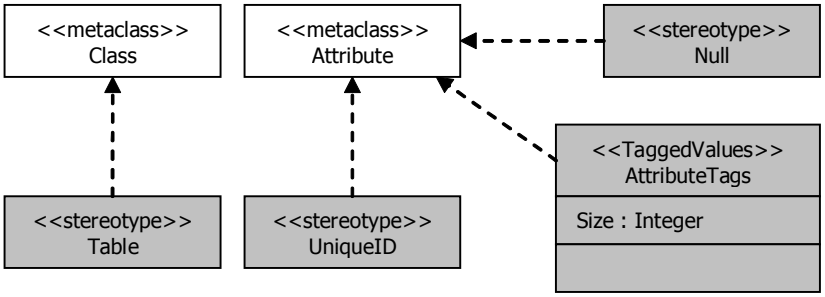


Figura 4.4 Modelo do UML Profile Relacional

O modelo acima indica para cada estereótipo ou *tagged value* do *profile* o elemento do meta-modelo ao qual ele está associado. Do modelo acima podemos verificar, por exemplo, que o estereótipo *Table* se aplica apenas a classes do modelo diagrama de classes da UML.

Como já foi abordado no tópico 2.5, um modelo de *UML Profile* pode necessitar do estabelecimento de regras para sua aplicação além das que são possíveis representar

através da notação gráfica. A seguir, definimos códigos em OCL que estabelecem as regras para a utilização do *profile*:

- Invariante 1: toda classe que possui o estereótipo *Table* deve apresentar ao menos uma coluna com o estereótipo *UniqueID*. O código 4.1 especifica essa regra.

```
context Class inv:
 self.feature->exists(isStereotyped('UniqueID'))
```

Código 4.1 Invariante 1 para o *UML Profile Relacional*

- Invariante 2: de forma análoga, um atributo só pode conter o estereótipo *UniqueID* se estiver numa classe que possua o estereótipo *Table*. O código 4.2 especifica essa regra.

```
context UniqueID inv:
 self.owner.isStereotyped('Table');
```

Código 4.2 Invariante 2 para o *UML Profile Relacional*

- Invariante 3: um atributo com o *tagged value Size* deve ser do tipo *String*. O código 4.3 especifica essa regra.

```
context Size inv:
 self.owner.type.oclIsTypeOf(UML::DataType::String)
```

Código 4.3 Invariante 3 para o *UML Profile Relacional*

- Invariante 4: um atributo com o *tagged value UniqueID* não pode ter o estereótipo *Null*. O código 4.3 especifica essa regra.

```
context Attribute inv:
 self.isStereotyped('UniqueID') and not self.isStereotyped('Null');
```

Código 4.4 Invariante 4 para o *UML Profile Relacional*

4.3.3. Transformações usando UML Profile Relacional

No tópico 2.5 foram discutidos os aspectos necessários para que a transformação MDA seja realizada. A seguir são destacados os pontos fundamentais para a realização da transformação do modelo PIM do acadêmico para um modelo PSM Relacional:

- Meta-modelo do modelo de origem PIM, nesse caso o meta-modelo simplificado do diagrama de classes UML da figura 4.5, retirado de [KWB2003]. O termo *UML* apresentado nos códigos a seguir serão referências para este meta-modelo;

- Meta-modelo do modelo de destino PSM relacional, mostrado na figura 4.1. O termo *SQL* apresentado nos códigos a seguir serão referências para este meta-modelo;

- Uma linguagem de definição de transformação que converta os elementos do modelo de origem para elementos no modelo de destino;

- Uma ferramenta de transformação. Neste trabalho faremos a transformação manualmente, ao contrário do que aconteceria num processo MDA real.

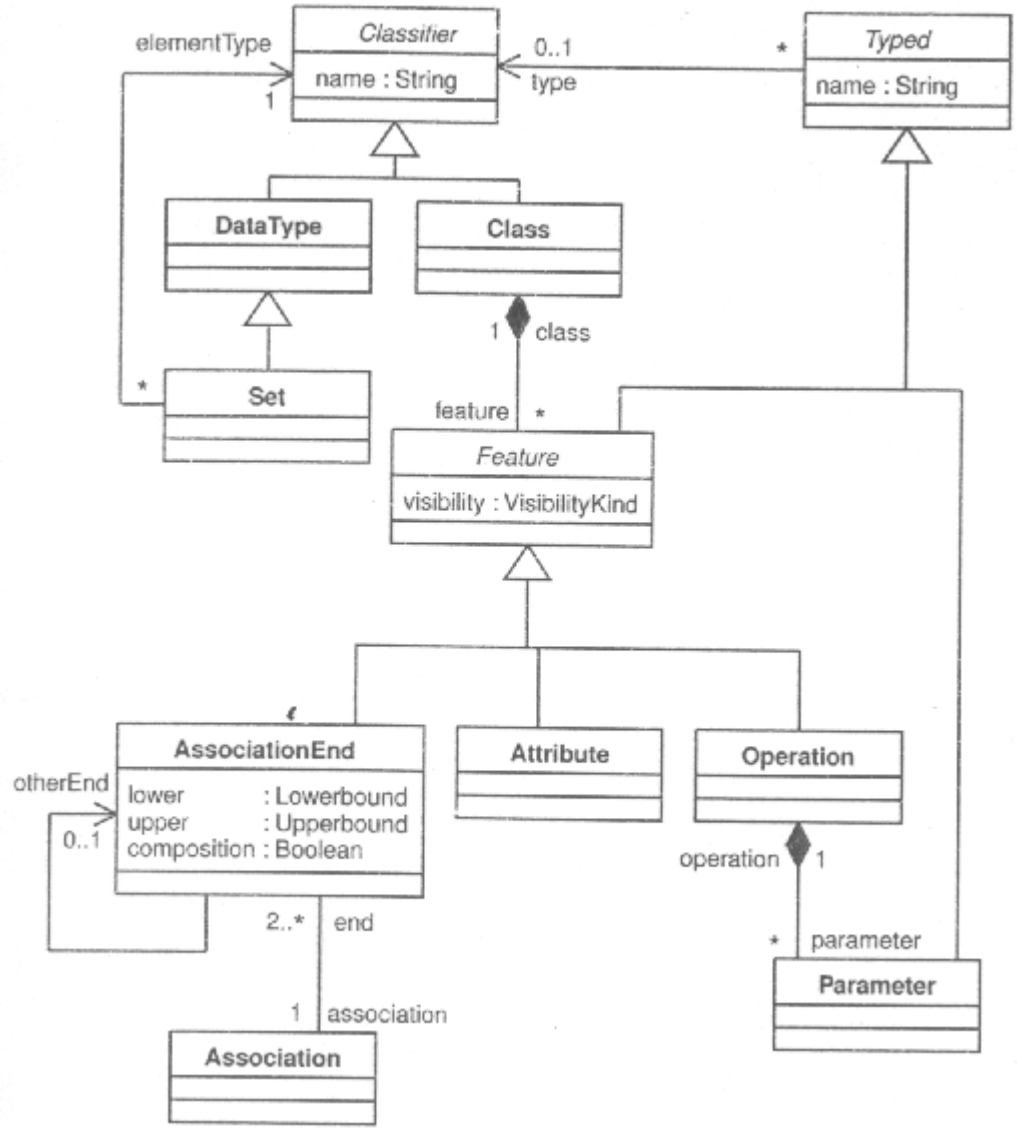


Figura 4.5 Meta-modelo Simplificado do Diagrama de Classes UML

A seguir mostraremos, em forma textual, seguido do respectivo código de transformação, os passos para geração do modelo relacional:

1. Para cada classe pública do modelo PIM que contenha o estereótipo *Table*, gere uma tabela no modelo relacional. O código 4.1 formaliza essa transformação;

```
Transformation ClassToTable(UML,SQL) {
source
 class : UML::Class;
target
 table : SQL::Table;
source condition
 class.isStereotyped('Table');
mapping
 class.name <~> table.name;
 class.attributes() <~> table.column;
 class.associationEnds() <~> table.foreign;
}
```

Código 4.5 Transformação ClassToTable

2. Para cada atributo da classe UML opcional, gere uma coluna do tipo opcional na tabela resultante. O código 4.5 formaliza essa transformação;

```
Transformation AttributeToNullColumn(UML,SQL)
{
source
 attr        :      UML::Attribute;
target
 column      :      SQL::Column;
source condition
 attr.isStereotyped('Null');
target condition
 column.nullable=true;
unidirectional;
mapping
 attr.name <~> column.name;
 attr.type <~> column.type;
}
```

Código 4.6 Transformação AttributeToNullColumn

3. Para cada atributo da classe UML não opcional, gere uma coluna não opcional na tabela resultante. O código 4.6 formaliza essa transformação;

```
Transformation AttributeToNotNullColumn(UML,SQL)
{
source
 attr        :      UML::Attribute;
target
 column      :      SQL:: Column;
source condition
 not attr.isStereotyped('Null');
target condition
 column.nullable=false;
unidirectional;
mapping
 attr.name <~> column.name;
 attr.type <~> column.type;
}
```

Código 4.7 Transformação AttributeToNotNullColumn

4. Cada tipo de dado primitivo da UML deve ser convertido para o respectivo tipo primitivo em SQL. As três transformações do código 4.7 realizam a conversão dos três tipos de dados primitivos em UML usados no sistema acadêmico para os respectivos tipos em SQL. O detalhe que deve ser ressaltado é que na transformação do tipo UML *String*, utiliza-se o *tagged value Size* para determinar a quantidade de caracteres a ser usado na implementação em SQL;

```
Transformation UMLDataTypeToSQLDataTypeInteger(UML,SQL)
{
source
 umlDataType : UML::DataType;
target
 sqlDataType : SQL::SQLDataType;
source condition
 umlDataType = UML::DataType::int;
target condition
 sqlDataType = SQL::SQLDataType::INTEGER;
unidirectional;
}

Transformation UMLDataTypeToSQLDataTypeDate(UML,SQL)
{
source
 umlDataType : UML::DataType;
target
 sqlDataType : SQL::SQLDataType;
source condition
 umlDataType = UML::DataType::Date;
target condition
 sqlDataType = SQL::SQLDataType::DATE;
unidirectional;
}
```

```
Transformation UMLDataTypeToSQLDataTypeString(UML,SQL)
{
source
 umlDataType : UML::DataType;
target
 sqlDataType : SQL::SQLDataType;
source condition
 umlDataType = UML::DataType::String;
target condition
 sqlDataType = SQL::SQLDataType::VARCHAR and
 sqlDataType.Size = umlDataType.TaggedValue('Size').Value;
unidirectional;
}
```

Código 4.8 Transformações de Tipos UML para SQL

5. Para cada classe com o estereótipo *Table*, crie uma chave primária na tabela resultante relacionando todos os atributos com estereótipo *UniqueID* como parte dessa chave. O código 4.8 formaliza essa transformação;

```
Transformation ClassUniqueIdToKey(UML,SQL)
{
source
 class : UML::Class;
 attr : UML::Attribute;
target
 key   : SQL::Key;
 column : SQL::Column;
 table : SQL::Table;
source condition
 class.isStereotyped('Table') and
 attr.isStereotyped('UniqueID') and
 attr.class = class;
target condition
 key.table.name = class.name and
 table = key.table and
 column.table = table and
 table.primary=key and
 key.column->includes(column);
unidirectional;
mapping
 class.name + "_PK" <~> key.name;
}
```

Código 4.9 Transformação ClassUniqueIdToKey

6. Para cada fim de associação, cujo outro fim de associação seja uma classe, crie uma chave estrangeira. Caso o fim de associação referencie uma classe de associação ao invés de uma classe, o tratamento da transformação deveria ser diferente. Como no modelo do acadêmico não foi utilizado classe de asso-

ciação, omitiremos esse tipo de transformação. O código 4.9 formaliza essa transformação;

```
Transformation AssociationEndToForeignKey(UML,SQL)
{
source
 assocEnd : UML::AssociationEnd;
target
 foreign : SQL::ForeignKey;
source condition
 assocEnd.upper = 1 and
 assocEnd.association.oclIsTypeOf(UML::Association)
unidirectional;
mapping
 assocEnd.name <~> foreign.name;
 assocEnd.type <~> foreign.referencedKey;
}
```

Código 4.10 Transformação AssociationEndToForeignKey

7. A transformação anterior apenas indica que uma *foreign key* deve ser gerada, mas as colunas que a formarão precisam ainda ser criadas na tabela que faz a referência. O código 4.10 abaixo complementa a transformação anterior gerando as colunas estrangeiras;

```
Transformation GenerateForeignColumns(SQL,SQL)
{
source
 foreign : SQL::ForeignKey;
 refColumn : SQL::Column;
 key : SQL::Key;
target
 column : SQL::Column;
source condition
 key.table = foreign.table and
 foreign.referencedKey.column->includes(refColumn)
unidirectional;
mapping
 column.name <~> refColumn.name;
 column.table <~> foreign.table;
 column.type <~> refColumn.type;
 column.foreign->first() <~> foreign;
}
```

Código 4.11 Transformação GenerateForeignColumns

A figura 4.6 mostra o modelo resultante da aplicação das transformações para geração do modelo relacional usando os códigos de definição mostrados com a aplicação do *profile* UML Relacional.

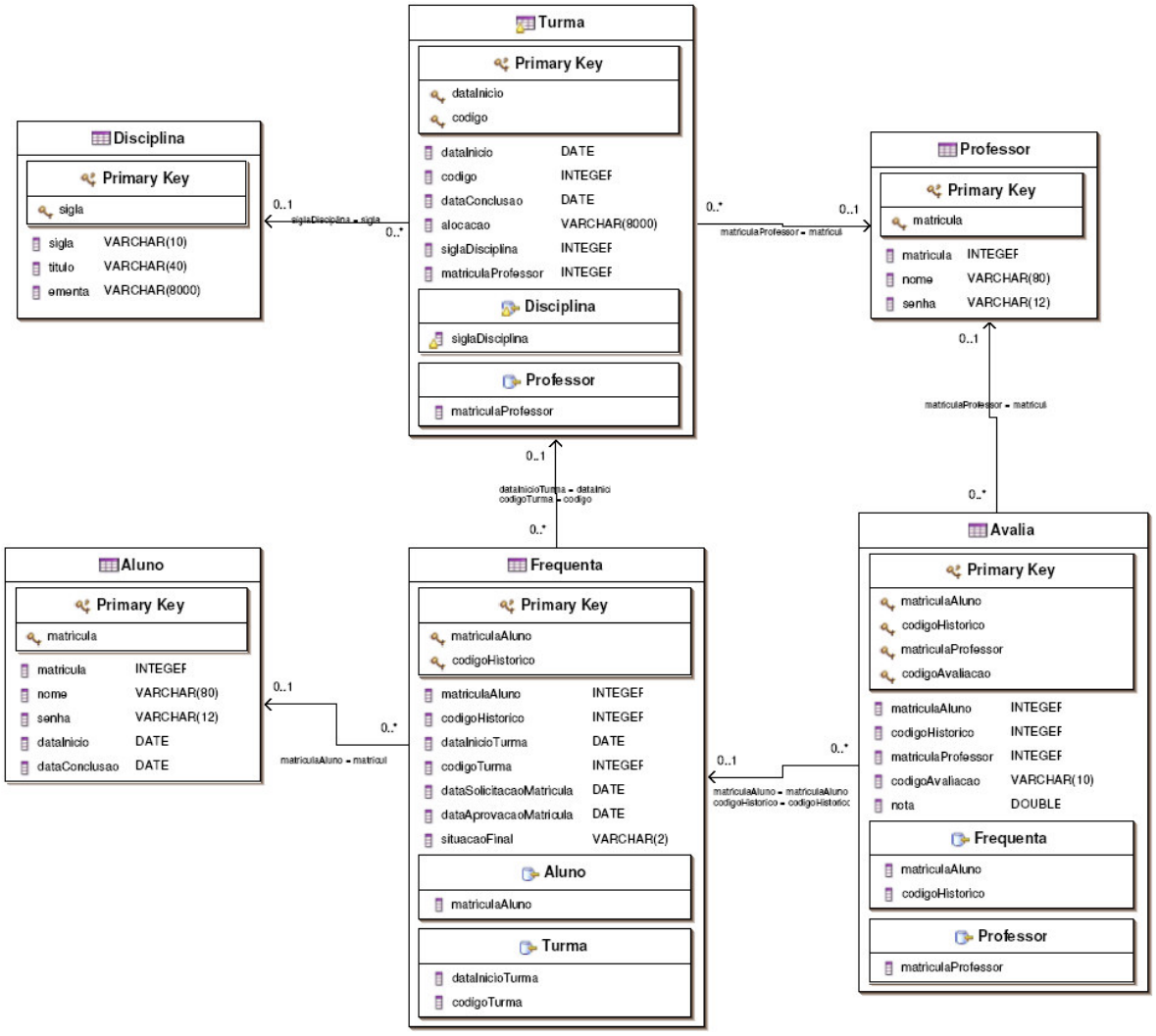


Figura 4.6 Modelo PSM Relacional do Sistema Acadêmico

4.3.4. Conclusão

O resultado obtido com a aplicação manual das transformações usando o *Profile* ora proposto mostrou-se eficaz frente às necessidades demandadas pelo sistema exemplo do Acadêmico. Embora seja um estudo de caso simples, várias situações envolvidas nesse tipo de transformação puderam ser experimentadas. O código completo da transformação PIM para código ANSI´92 Relacional é apresentado no apêndice A (Camada de Persistência) deste documento.

5. ANÁLISE DOS RESULTADOS

5.1. Introdução

Da experiência na aplicação dos conceitos MDA na construção do sistema Acadêmico, desenvolvido com o intuito de explorar os principais aspectos e desafios que a abordagem aponta, podemos tirar várias conclusões valiosas. Numa análise superficial, podemos declarar que os resultados obtidos com essa experiência serviram para vislumbrar as vantagens que a abordagem pode oferecer aos futuros projetos de desenvolvimento de *software*, sustentando os argumentos expostos por diversos autores estudados neste trabalho. Os itens a seguir, divididos nos aspectos mais relevantes, buscam expor nossa análise dos resultados obtidos com essa experiência.

5.2. Processo de Desenvolvimento

Uma vez que o padrão MDA não dita qualquer orientação quanto a aplicação de uma metodologia para o processo de desenvolvimento, conduzimos o levantamento de requisitos para a elaboração do sistema da mesma forma como faríamos usando um processo de desenvolvimento tradicional. Da mesma forma, a proposta desse trabalho não enfocou o processo de desenvolvimento empregado, porém é certo que esse aspecto continuará tendo grande importância para o sucesso dos projetos de *software*. O trabalho em [FS2004] aborda questões sobre processos de desenvolvimento e apresenta

uma proposta de processo chamada *Model Driven Engineering*, que define uma metodologia adequada para um projeto de sistema aplicando o MDA.

Citamos, na introdução deste trabalho, a desconfiança ocorrida quando surge uma nova plataforma para desenvolvimento de sistema usando um maior nível de abstração. Os utilizadores da nova ferramenta ou da nova prática sentem-se, usualmente, desconfortáveis com o aumento na distância na linguagem utilizada para expressar o sistema e a linguagem que será de fato entendida pelo computador. Esses praticantes que migram para a nova plataforma sentem-se inseguros quanto a eficiência que as ferramentas de tradução terão em gerar o sistema pretendido.

Durante o desenvolvimento do modelo PIM do Acadêmico, sofremos dessa mesma desconfiança. Embora o MDA oriente o processo de desenvolvimento para a elaboração de um modelo totalmente independente de plataforma, as dúvidas quanto ao ambiente em que o sistema será implementado estiveram presentes e dificultaram a aplicação do processo em seu sentido ideal, pois acabaram antecipando problemas que deveriam ser abstraídos até o momento da geração do sistema executável. Temos a confiança de que, como pregam os evangelistas, o amadurecimento da proposta irá gradualmente diminuindo essa desconfiança.

5.3. Transformações

Uma vez tendo desenvolvido o modelo PIM, nossa atenção se voltou para a geração dos modelos PSM. A definição da arquitetura utilizada teve a intenção de adotar um cenário comum das aplicações atuais. Sendo assim, tivemos três transformações: uma para o modelo de apresentação, uma para o modelo de negócios e outra para o modelo de persistência de dados (que neste trabalho teve os bancos relacionais como escolha). Nos tópicos a seguir expomos as percepções obtidas em cada uma dessas três transformações.

5.3.1. Modelo de Negócios

A transformação do modelo PIM para o modelo PSM de negócios foi, na nossa visão, a que apresentou a maior facilidade. A própria modelagem do sistema usando um modelo PIM UML, embora independente de plataforma, favorece a geração dos artefatos dessa camada para plataformas específicas. Com propósito de colocar em prática a implementação do trabalho, escolhemos a linguagem C# como alvo para nosso modelo de implementação. Apesar disso, devemos citar que a própria ferramenta usada para criação do modelo PIM (Eclipse com o *plug-in* EclipseUML, citados no Apêndice C) gera grande parte do código Java de que precisaríamos para implementar nessa outra linguagem. O trabalho em [DGLRS2003] concorda com essa facilidade de geração dos artefatos da camada de negócios a partir do PIM e indica uma extensão da linguagem que especializa os recursos dessa transformação, a chamada *UML Profile for Enterprise Distributed Objects Computing* (EDOC). Além disso, vale ressaltar que a escolha de implementar essa camada como *Web Services* [Wsa2004] facilitou a geração dos artefatos de integração com a camada de apresentação, que na linguagem MDA são chamados os *bridges*, embora não tenha sido algo originalmente intencionado nesse trabalho.

5.3.2. Modelo de Apresentação

Ao contrário do proposto em [Amb2003], ficou claro que uma linguagem baseada no diagrama de classes UML não é suficiente para representar todos os aspectos necessários para a geração de um modelo de interface. Acreditamos que novas linguagens surgirão nesse contexto e que tornarão possível a especificação completa, independente de plataforma como mandam os preceitos do MDA, dos elementos de interação com o usuário. Como conseqüência, limitamo-nos a indicar essa deficiência através da geração dos códigos de implementação para uma interface Web, neste trabalho implementada

em Asp.Net [Asp2004]. Nas pesquisas por referências MDA no contexto de interface constatamos que esse aspecto precisa ser trabalhado para emprego da abordagem também nesse cenário. Entretanto, não consideramos essa deficiência em nada desabonadora para a utilização atual da proposta MDA, visto que a camada de apresentação já possui excelentes ferramentas de implementação, mesmo não sendo baseadas em MDA.

5.3.3. Persistência dos Dados

Na transformação para o modelo relacional, conseguimos experimentar vários passos propostos na abordagem MDA, que objetivam complementar o modelo PIM de tal forma que um modelo de implementação possa ser gerado. No estudo dessa transformação, aplicamos um *UML Profile* definido neste próprio trabalho e, com isto, tivemos a oportunidade de descrever em detalhes como o processo de transformação ocorreria. Essa experiência deixou evidente a importância dos métodos de complementação semântica e da linguagem de definição da transformação dentro do MDA.

Outro aspecto que pudemos observar através da aplicação dessa transformação usando o *profile* proposto é vantagem em usar marcações em nível do PIM, ao contrário de usá-las em nível PSM, como advogam [KWB2003, MSUW2004] alguns praticantes do MDA. Ao associar complementações semânticas ao modelo de maior nível de abstração, obtemos a vantagem de gerar artefatos para qualquer produto compatível com o padrão SQL ANSI´92, conforme especificado da declaração de independência intecionada pelo modelo PIM do Acadêmico.

6. CONCLUSÃO

Embora a importância do MDA possa ser atestada pelo atual estado da arte, muitas especificações ainda precisam ser finalizadas e algumas até mesmo propostas para atingir os objetivos esperados por seus defensores. No momento em que esse trabalho está sendo finalizado, várias propostas de revisão estão em andamento na OMG sobre aspectos importantes, dentre as quais a linguagem QVT, que definirá um padrão de linguagem a ser utilizada para codificar a definição de transformações em MDA.

A geração dos aspectos estáticos do sistema certamente promove grandes ganhos de produtividade, mas não são suficientes para justificar a adoção do MDA. Ainda não estão claros muitos pontos relativos à derivação dos aspectos dinâmicos do sistema a partir do modelo independente de plataforma. Embora parte do código possa ser obtido através das regras OCL, fica evidente que a definição de linguagens como a *Action Semantics* (AS) serão fundamentais para completar essa questão. A linguagem AS permitirá descrever, em alto nível de abstração, uma lógica de programação que resolva um determinado problema que nosso sistema se propõe a tratar.

O surgimento de ferramentas CASE que dizem suportar o padrão MDA cresce a cada dia, embora vários autores apontem que, na verdade, apenas parte das especificações é atendida. Como já foi citado, já existem ferramentas capazes de realizar a transformação de modelos PIM para códigos de implementação, mas MDA propõe a padroni-

zação dessa capacidade, o que cria um círculo virtuoso, ampliando a disponibilidade de ferramentas para atender esses objetivos.

Podemos vislumbrar que a utilização de um modelo PIM único para a especificação de todos os aspectos de um sistema estará longe das reais necessidades de MDA para a maioria dos projetos. Espera-se que grande parte dos projetos envolverá a utilização de vários modelos PIM, cada um adequado para modelar um aspecto específico do sistema, que depois serão mesclados automaticamente pelas ferramentas MDA. Essa abordagem não foi explorada neste trabalho, embora os problemas apontados na transformação da interface tenham indiretamente mostrado a necessidade no uso de um modelo PIM específico para o propósito em questão. Recursos como *Merging Mappings* [MSUW2004] deverão ser praticados em projetos MDA de maior complexidade. Essa mesclagem possibilitará a construção de sistemas separando em vários modelos aspectos específicos, como por exemplo, o modelo de especificação de persistência podendo ser mesclado com um modelo que implemente aspectos de segurança, gerando um modelo PIM único. Aspectos não-funcionais, como os de qualidade de serviço explorados em [YBB2003], também podem tirar proveito dessa capacidade.

O problema de sincronização, abordado neste trabalho no item 2.5.4, foi explorado em um nível superficial tendo em vista as graves conseqüências que ele acarreta na consistência de modelos entre vários níveis de abstração. O estudo de caso do sistema Acadêmico considerou um processo ideal para sincronização de modelos classificado em [Fra2003] como *Forward-Engineering Only*, onde todas as alterações na especificação do sistema são feitas exclusivamente nos modelos PIM. Embora seja provável que as ferramentas baseadas em MDA apliquem exclusivamente a abordagem unidirecional num primeiro momento, a aceitação e transição para o paradigma MDA será facilitada com suporte à mudanças também nos modelos de implementação (PSM ou código). Para isso, os problemas para sincronização de modelos precisarão ser resolvidos. Vários autores já apontam soluções para o problema de sincronização de modelos em MDA, como, por exemplo, o trabalho em [AP2004].

A proposta deste trabalho para a questão da transformação para banco de dados relacionais se restringiu a tratar das situações apresentadas pelo sistema exemplo – o sistema Acadêmico. Quando essa mesma transformação for empregada em outro sistema, mais complexo, outros aspectos surgirão, como, por exemplo, a necessidade de gerar procedimentos armazenados e visões de dados, características comuns em bancos relacionais e não demandas pelo sistema exposto neste trabalho.

Uma oportunidade de trabalho futuro identificada a partir do nosso estudo de caso surge da ausência de uma linguagem de especificação de alto nível para a modelagem de interfaces, conforme visto no item 3.5.4. A criação de um meta-modelo que defina uma linguagem para criação de modelos PIM capazes de especificar interfaces de usuário representaria uma ampliação das possibilidades do MDA. Em nenhum momento, durante as pesquisas realizadas para este trabalho, nos deparamos com qualquer proposta nesse sentido.

APÊNDICE A – CÓDIGO DO ACADÊMICO

Camada de Persistência: SQL-ANSI´92 Relacional

```
CREATE TABLE Disciplina
(
 sigla CHARACTER VARYING(5) NOT NULL,
 titulo CHARACTER VARYING(40) NOT NULL,
 ementa CHARACTER VARYING(8000) NOT NULL,
 CONSTRAINT Disciplina_PK PRIMARY KEY (sigla)
)

CREATE TABLE Professor
(
 matricula INTEGER NOT NULL,
 nome CHARACTER VARYING(80) NOT NULL,
 senha CHARACTER VARYING(12) DEFAULT 'senha' NOT NULL,
 CONSTRAINT Professor_PK PRIMARY KEY (matricula)
)

CREATE TABLE Aluno
(
 matricula INTEGER NOT NULL,
 nome CHARACTER VARYING(80) NOT NULL,
 senha CHARACTER VARYING(12) DEFAULT 'senha' NOT NULL,
 dataInicio DATE NOT NULL,
 dataConclusao DATE,
 CONSTRAINT Aluno_PK PRIMARY KEY (matricula)
)

CREATE TABLE Turma
(
 dataInicio DATE NOT NULL,
 codigo INTEGER NOT NULL,
 dataConclusao DATE,
 alocacao CHARACTER VARYING(8000),
 siglaDisciplina CHARACTER VARYING(5) NOT NULL,
 matriculaProfessor INTEGER NOT NULL,
 CONSTRAINT Turma_PK PRIMARY KEY (dataInicio,codigo),
 CONSTRAINT Turma_Disciplina_FK FOREIGN KEY (siglaDisciplina) REFERENCES Disciplina,
 CONSTRAINT Turma_Professor_FK FOREIGN KEY (matriculaProfessor) REFERENCES Professor
)

CREATE TABLE Frequenta
(
 matriculaAluno INTEGER NOT NULL,
 codigoHistorico INTEGER NOT NULL,
 dataInicioTurma DATE NOT NULL,
 codigoTurma INTEGER NOT NULL,
 dataSolicitacaoMatricula DATE NOT NULL,
 dataAprovacaoMatricula DATE,
 situacaoFinal CHARACTER VARYING(2),
 CONSTRAINT Frequenta_PK PRIMARY KEY (matriculaAluno,codigoHistorico),
 CONSTRAINT Frequenta_Aluno_FK FOREIGN KEY (matriculaAluno) REFERENCES Aluno,
 CONSTRAINT Frequenta_Turma_FK FOREIGN KEY (dataInicioTurma,codigoTurma) REFERENCES
Turma
)
```

```
CREATE TABLE Avalia
(
 matriculaAluno INTEGER NOT NULL,
 codigoHistorico INTEGER NOT NULL,
 matriculaProfessor INTEGER NOT NULL,
 codigoAvaliacao CHARACTER VARYING(10) NOT NULL,
 nota DOUBLE NOT NULL,
 CONSTRAINT Avalia_PK PRIMARY KEY
(matriculaAluno,codigoHistorico,matriculaProfessor,codigoAvaliacao),
 CONSTRAINT Avalia_Professor_FK FOREIGN KEY (matriculaProfessor) REFERENCES Professor,
 CONSTRAINT Avalia_Frequenta_FK FOREIGN KEY (matriculaAluno,codigoHistorico) REFERENCES
Frequenta
)
```

Camada de Negócios: C#

O código completo desse trecho do Apêndice pode ser encontrado em http://www.ic.uff.br/~mbelo/RelTecn.pdf.

Fragmento do Arquivo: Disciplina.asmx.cs

```
namespace AcademicoMT
{

 public class Disciplina : System.Web.Services.WebService
 {

  public struct RegDisciplina
  {
   public string sigla;
   public string titulo;
   public string ementa;
  }

  [WebMethod]public RegDisciplina[] ListarDisciplinas()
  {
   (...)
  }

  [WebMethod]public void AdicionarDisciplina(string sigla,string titulo,string ementa)
  {
   (...)
  }

  [WebMethod]public void ExcluirDisciplina(string sigla)
  {
   (...)
  }

  [WebMethod]public void AlterarDisciplina(string sigla,string titulo,string ementa)
  {
   (...)
  }

  [WebMethod]public RegDisciplina ObterDisciplina(string sigla)
  {
   (...)
  }

 }

}
```

Fragmento do Arquivo: Professor.asmx.cs

```
namespace AcademicoMT
{

 public class Professor : System.Web.Services.WebService
 {

  public struct RegProfessor
  {
   public int matricula;
   public string nome;
  }

  [WebMethod]public RegProfessor[] ListarProfessores()
  {
   (...)
  }

  [WebMethod]public void AdicionarProfessor(int matricula,string nome)
  {
   (...)
  }

  [WebMethod]public void ExcluirProfessor(int matricula)
  {
   (...)
  }

  [WebMethod]public void AlterarProfessor(int matricula,string nome)
  {
   (...)
  }

  [WebMethod]public RegProfessor ObterProfessor(int matricula)
  {
   (...)
  }

  [WebMethod]public string ValidarSenha(int matricula,string senha)
  {
   (...)
  }

 }

}
```

Fragmento do Arquivo: Aluno.asmx.cs

```
namespace AcademicoMT
{
 public class Aluno : System.Web.Services.WebService
 {

  public struct RegAluno
  {
   public int matricula;
   public string nome;
   public DateTime dataInicio;
   public Object dataConclusao;
  }

  [WebMethod]public RegAluno[] ListarAlunos()
  {
   (...)
  }

  [WebMethod]public void AdicionarAluno(int matricula,string nome,DateTime dataInicio)
  {
   (...)
  }

  [WebMethod]public void ExcluirAluno(int matricula)
  {
   (...)
  }

  [WebMethod]public RegAluno ObterAluno(int matricula)
  {
   (...)
  }

  [WebMethod(MessageName="AlterarAluno1")]
  public void AlterarAluno(int matricula,string nome,DateTime dataInicio,DateTime
dataConclusao)
  {
   (...)
  }

  [WebMethod(MessageName="AlterarAluno2")]
  public void AlterarAluno(int matricula,string nome,DateTime dataInicio)
  {
   (...)
  }

  [WebMethod]public string ValidarSenha(int matricula,string senha)
  {
   (...)
  }

  public struct RegHistoricoAluno
  {
   public DateTime dataInicioTurma;
   public int codigoTurma;
   public string tituloDisciplina;
   public double mediaFinal;
   public string situacaoFinal;
  }
```

```
  [WebMethod]public RegHistoricoAluno[] ListarHistoricoAluno(int matricula)
  {
   (...)
  }

 }

}
```

Fragmento do Arquivo: Turma.asmx.cs

```
namespace AcademicoMT
{
 public class Turma : System.Web.Services.WebService
 {

  public struct RegTurma
  {
   public DateTime dataInicio;
   public int codigo;
   public Object dataConclusao;
   public string alocacao;
   public string siglaDisciplina;
   public int matriculaProfessor;
  }

  [WebMethod]public RegTurma[] ListarTurmas()
  {
   (...)
  }

  [WebMethod]public RegTurma[] ListarTurmasProfessor(int matriculaProfessor)
  {
   (...)
  }

  [WebMethod]public void AdicionarTurma(DateTime dataInicio,int codigo,string
alocacao,string siglaDisciplina,int matriculaProfessor)
  {
   (...)
  }

  [WebMethod]public void ExcluirTurma(DateTime dataInicio,int codigo)
  {
   (...)
  }

  [WebMethod]public RegTurma ObterTurma(DateTime dataInicio,int codigo)
  {
   (...)
  }

  [WebMethod(MessageName="AlterarTurma1")]    public void AlterarTurma(DateTime
dataInicio,int codigo,DateTime dataConclusao,string alocacao,string siglaDisciplina,int
matriculaProfessor)
  {
   (...)
  }

  [WebMethod(MessageName="AlterarTurma2")]    public void AlterarTurma(DateTime
dataInicio,int codigo,string alocacao,string siglaDisciplina,int matriculaProfessor)
  {
   (...)
  }

  [WebMethod]public void EncerrarTurma(DateTime dataInicio,int codigo)
  {
   (...)
  }
```

```
  [WebMethod] public bool PodeSolicitarInscricao(int matriculaAluno,DateTime
dataInicio,int codigo)
  {
   (...)
  }


 } // End of class Turma
}
```

Fragmento do Arquivo: Freqüenta.asmx.cs

```
namespace AcademicoMT
{
 public class Frequenta : System.Web.Services.WebService
 {

  [WebMethod] public void SolicitarInscricao(int matriculaAluno,DateTime
dataInicioTurma,int codigoTurma)
  {
   (...)
  }

  public struct RegSolicitacaoInscricaoPendente
  {
   public int matriculaAluno;
   public int codigoHistorico;
   public DateTime dataInicioTurma;
   public int codigoTurma;
   public DateTime dataSolicitacaoMatricula;
  }

  [WebMethod]public RegSolicitacaoInscricaoPendente[]
ListarSolicitacaoInscricaoPendente()
  {
   (...)
  }

  [WebMethod]public void AprovarInscricao(int matriculaAluno,int codigoHistorico)
  {
   (...)
  }

  [WebMethod]public void ApagarInscricao(int matriculaAluno,int codigoHistorico)
  {
   (...)
  }

  public struct RegAlunoTurma
  {
   public int matriculaAluno;
   public int codigoHistorico;
  }

  [WebMethod] public RegAlunoTurma[] ListarAlunoTurma(DateTime dataInicioTurma,int
codigoTurma)
  {
   (...)
  }
```

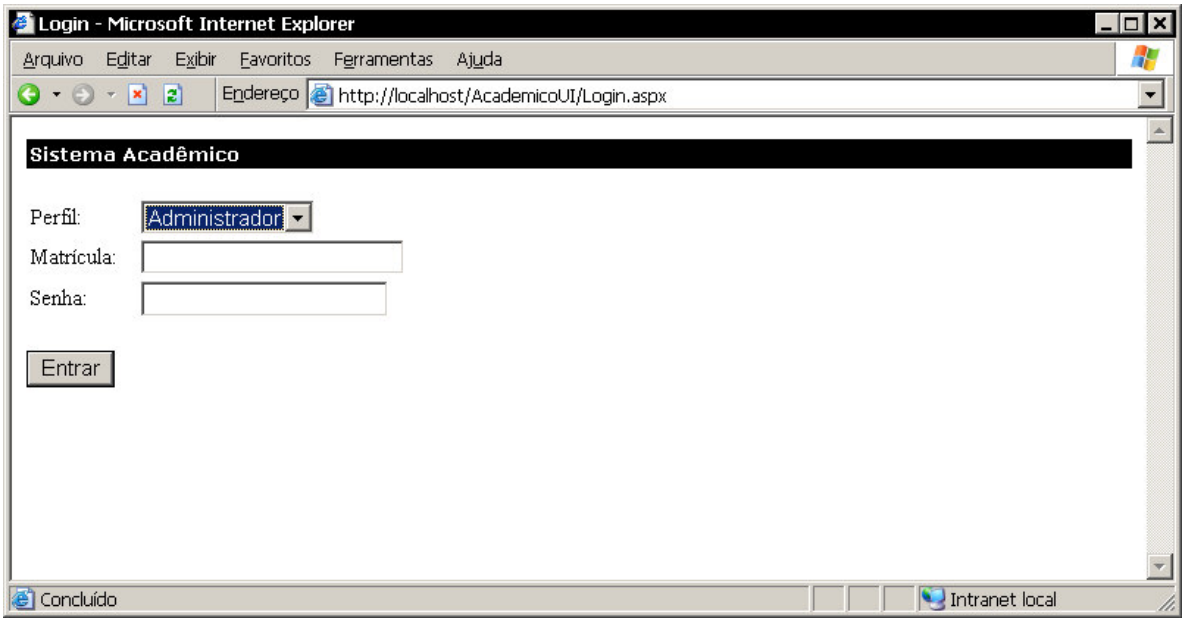
```
  [WebMethod]public void AlterarSituacaoFinal(int matriculaAluno,int
codigoHistorico,string situacaoFinal)
  {
   (...)
  }

 }

}
```

Fragmento do Arquivo: Avalia.asmx.cs

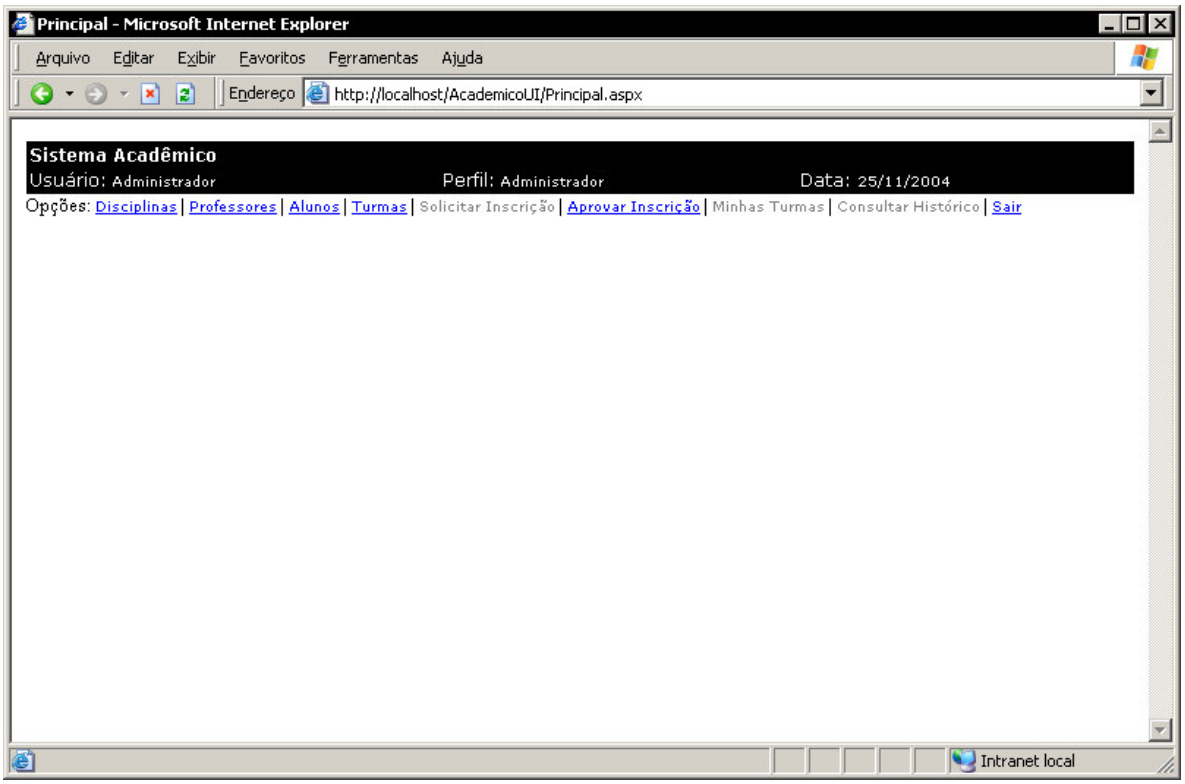
```
namespace AcademicoMT
{
 public class Avalia : System.Web.Services.WebService
 {

  [WebMethod] public void LancarNota(int matriculaAluno,int codigoHistorico,int
matriculaProfessor,string codigoAvaliacao,double nota)
  {
   (...)
  }

  [WebMethod] public void ExcluirNota(int matriculaAluno,int codigoHistorico,int
matriculaProfessor,string codigoAvaliacao)
  {
   (...)
  }

  public struct RegAvalia
  {
   public string matriculaAluno;
   public string codigoHistorico;
   public string matriculaProfessor;
   public string codigoAvaliacao;
   public string nota;
  }

  [WebMethod]public RegAvalia[] ListarNotas(int matriculaAluno,int codigoHistorico)
  {
   (...)
  }

  [WebMethod]public double ObterMediaAtual(int matriculaAluno,int codigoHistorico)
  {
   (...)
  }

 }

}
```

APÊNDICE B – INTERFACES DO ACADÊMICO

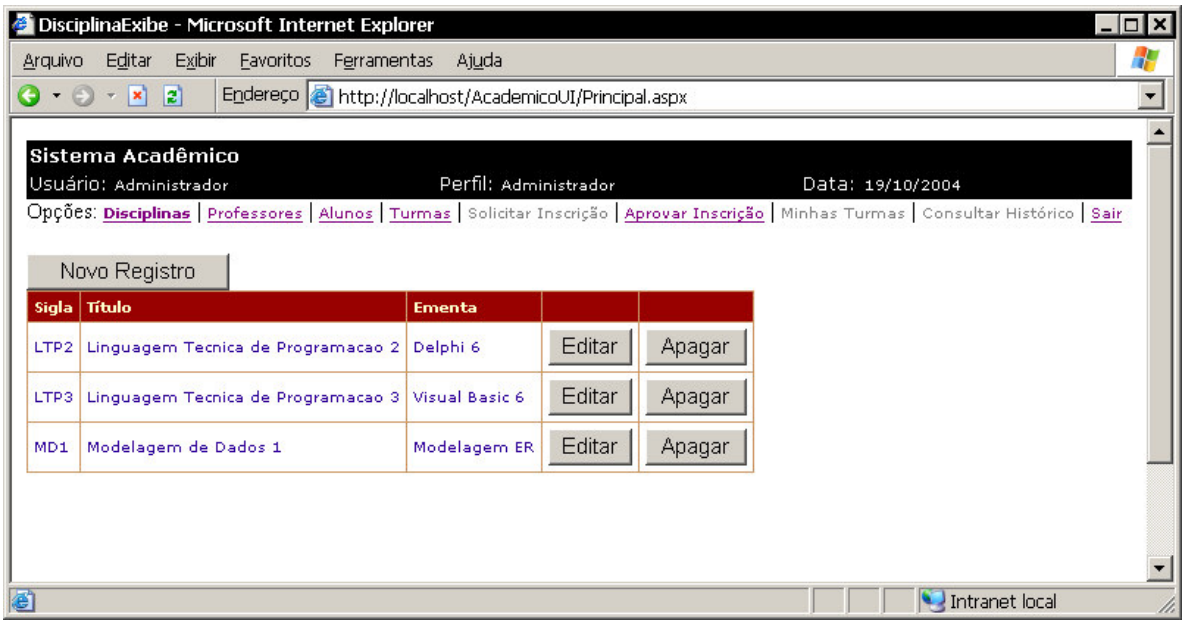
Tela Login.aspx



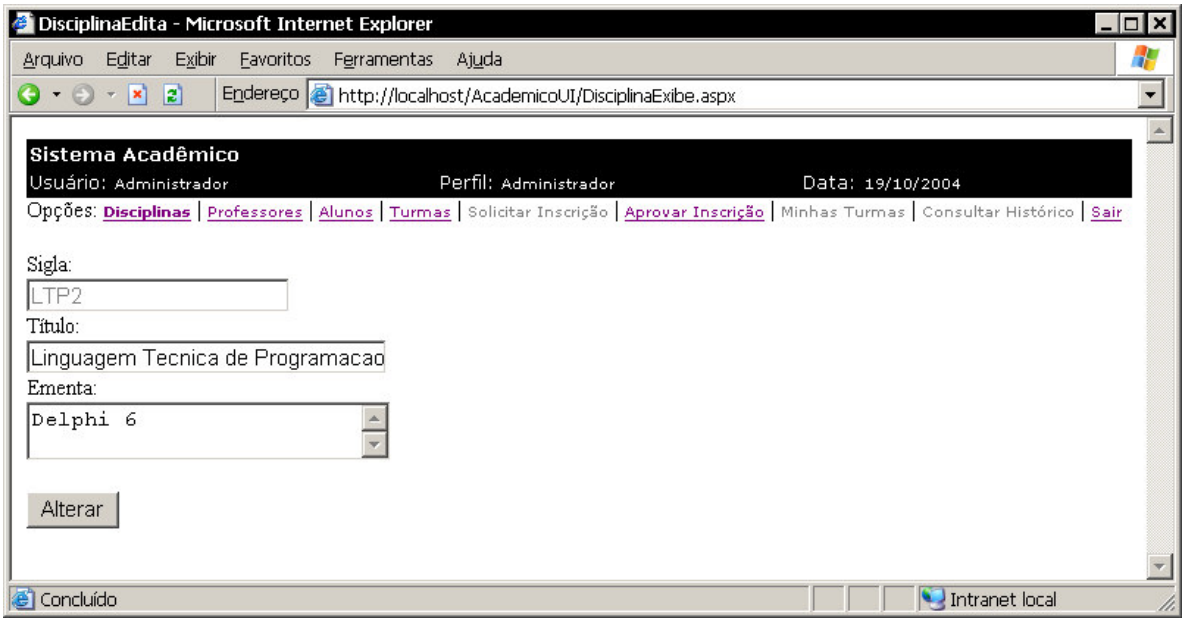
Tela Principal.aspx



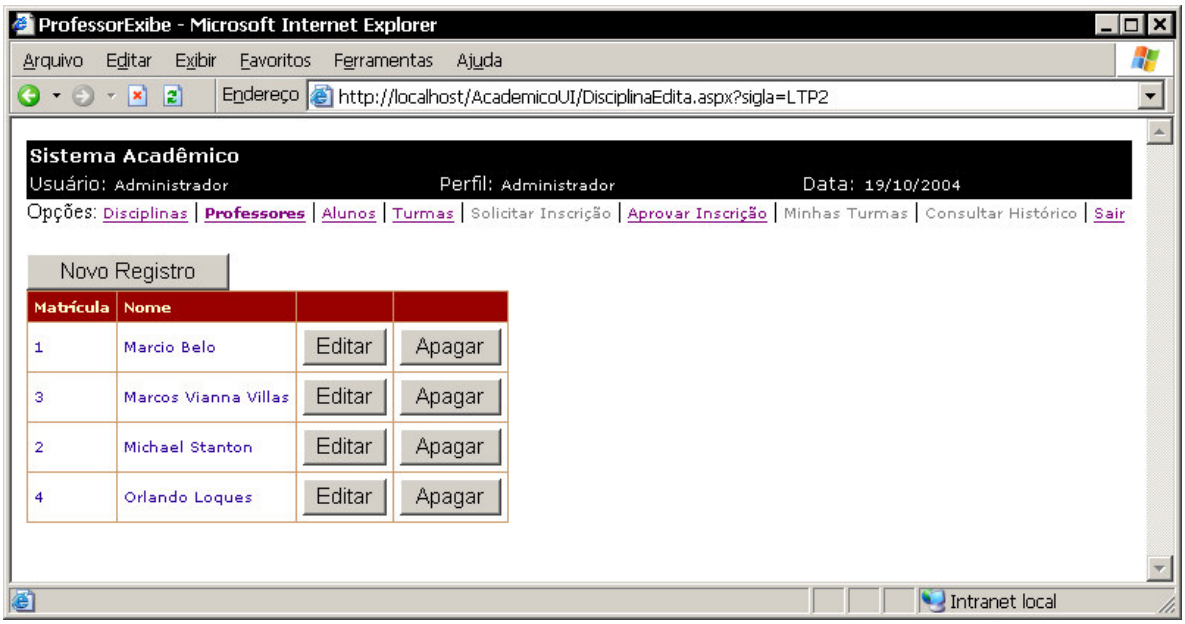
Tela DisciplinaExibe.aspx



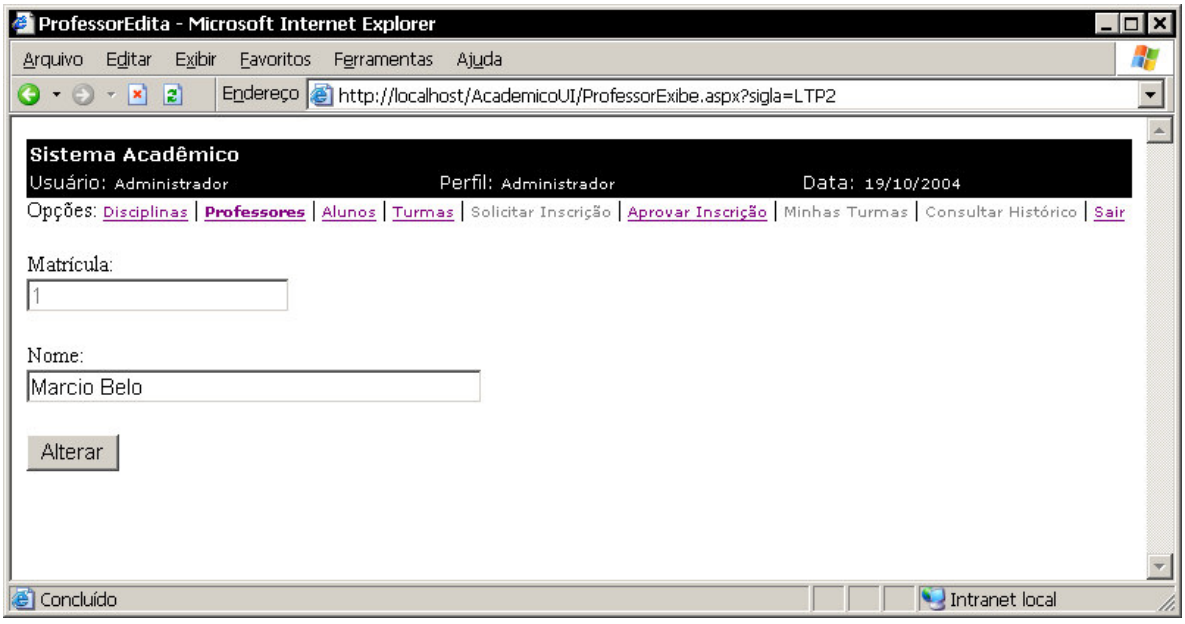
Tela DisciplinaEdita.aspx



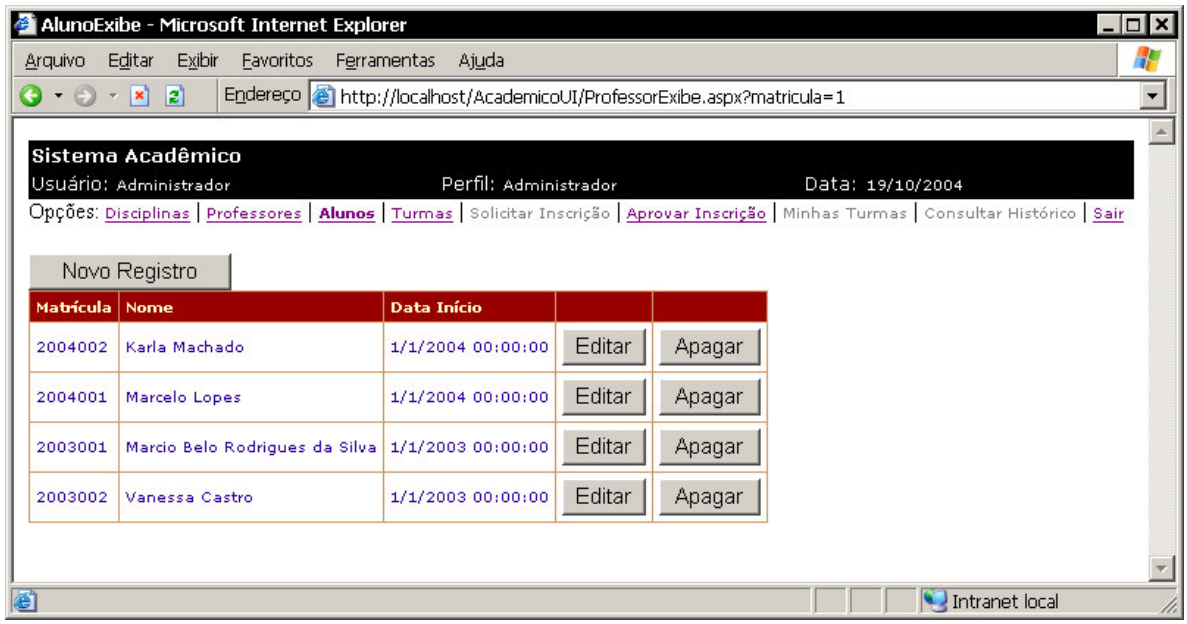
Tela ProfessorExibe.aspx



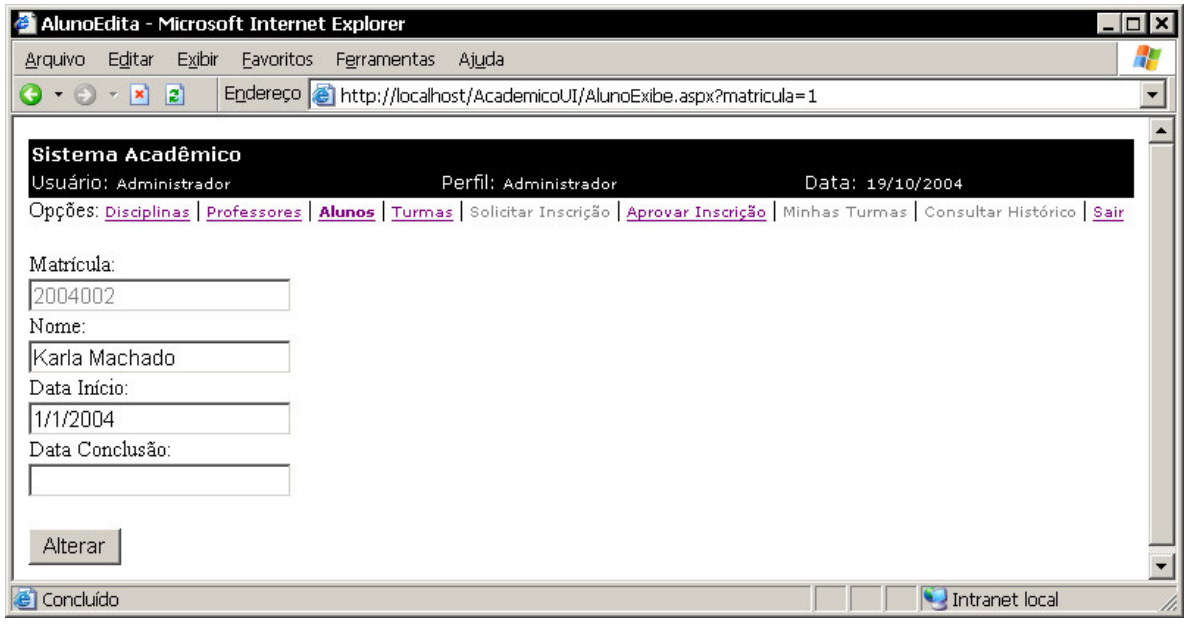
Tela ProfessorEdita.aspx



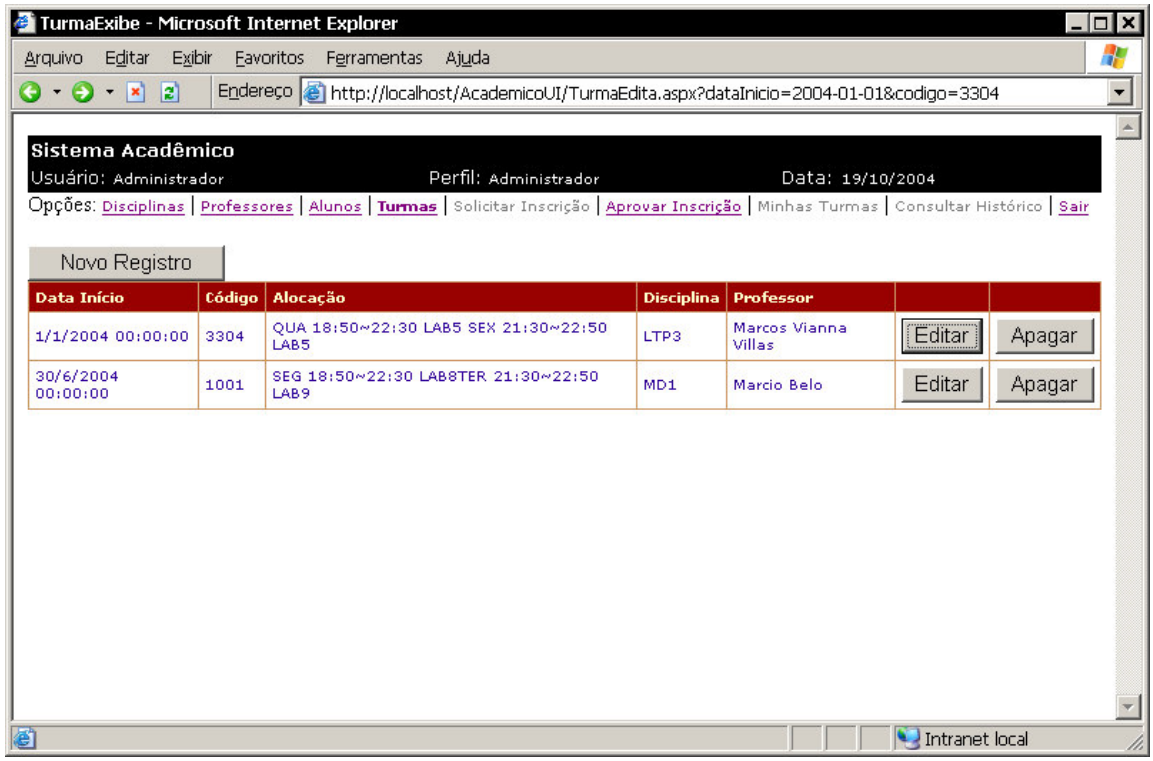
Tela AlunoExibe.aspx



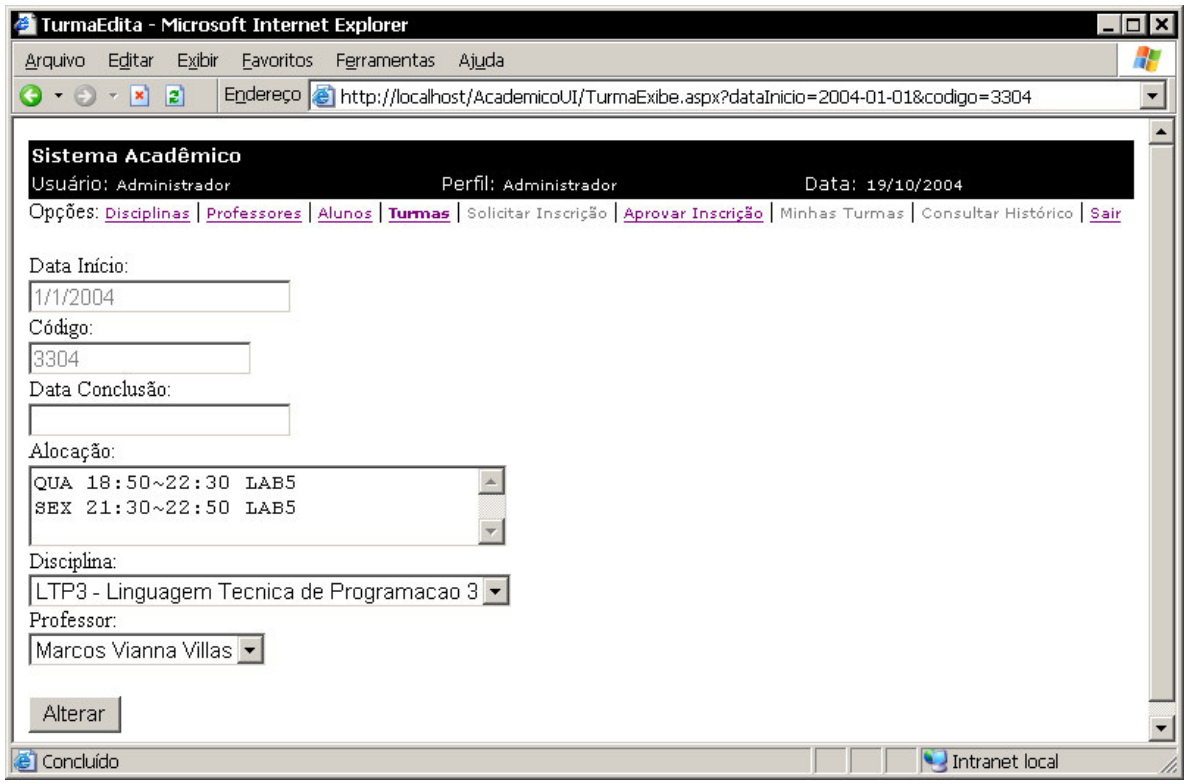
Tela AlunoEdita.aspx



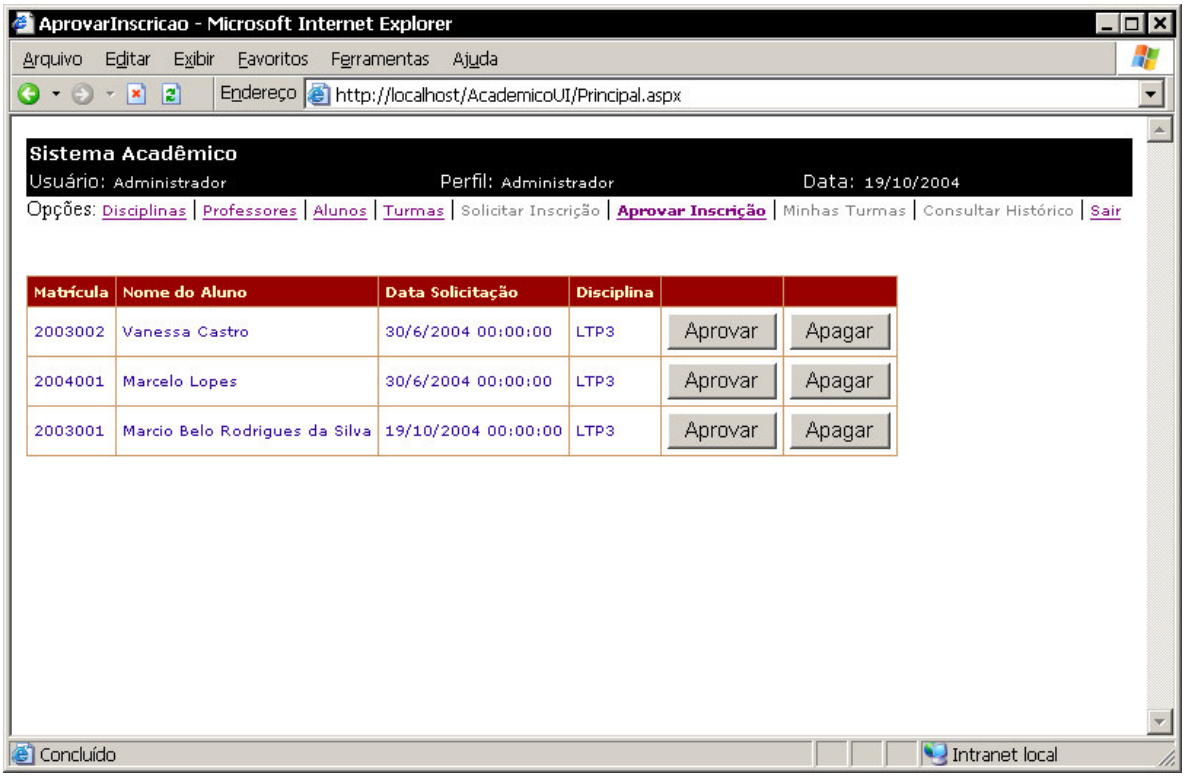
Tela TurmaExibe.aspx



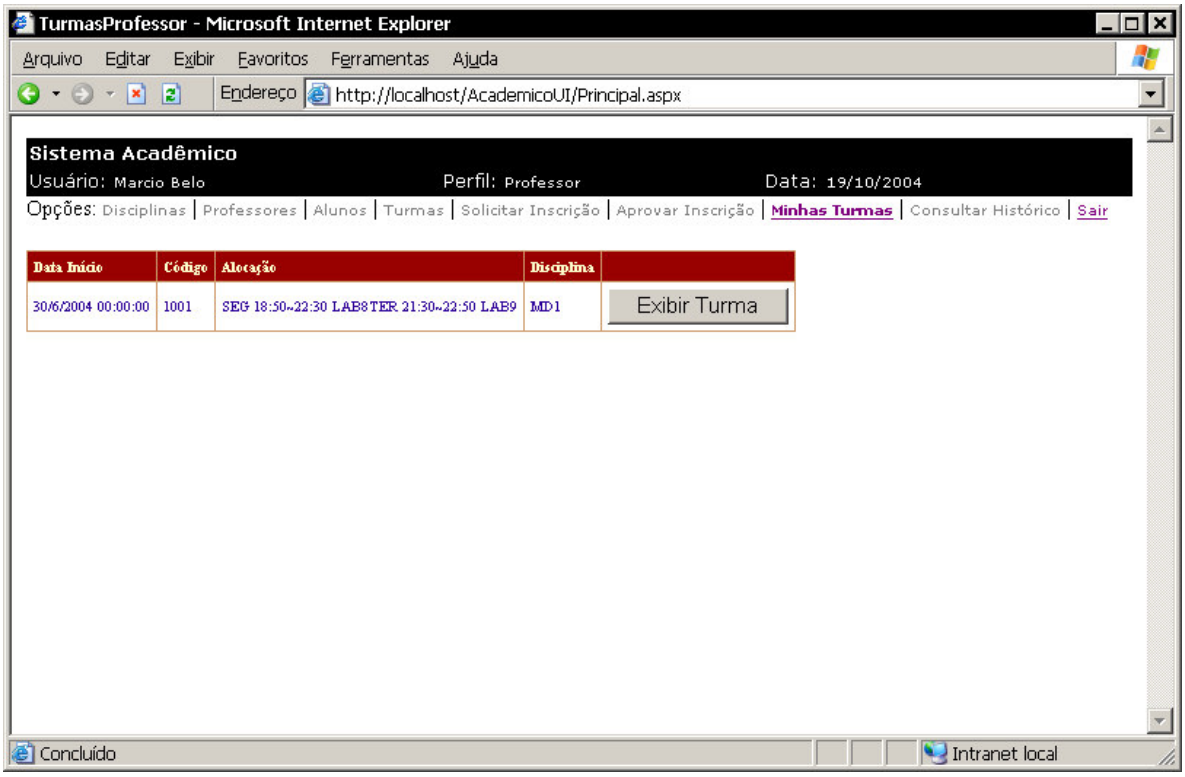
Tela TurmaEdita.aspx



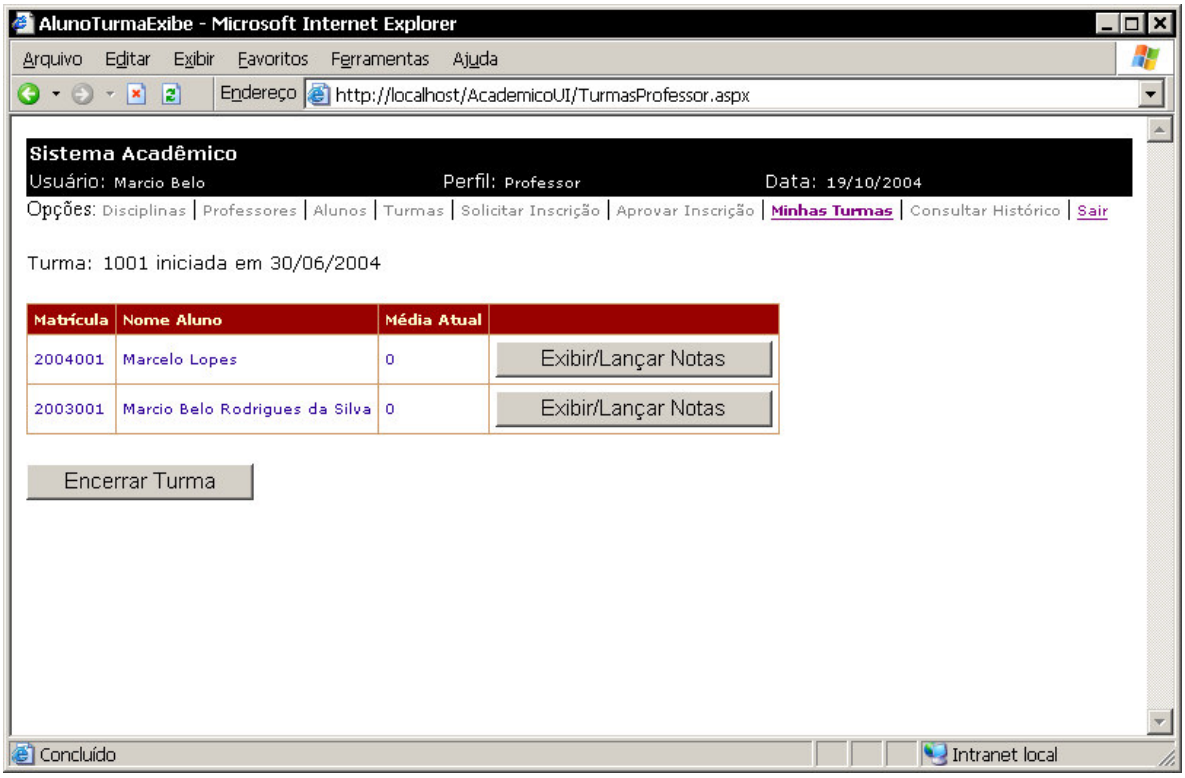
Tela AprovarInscricao.aspx



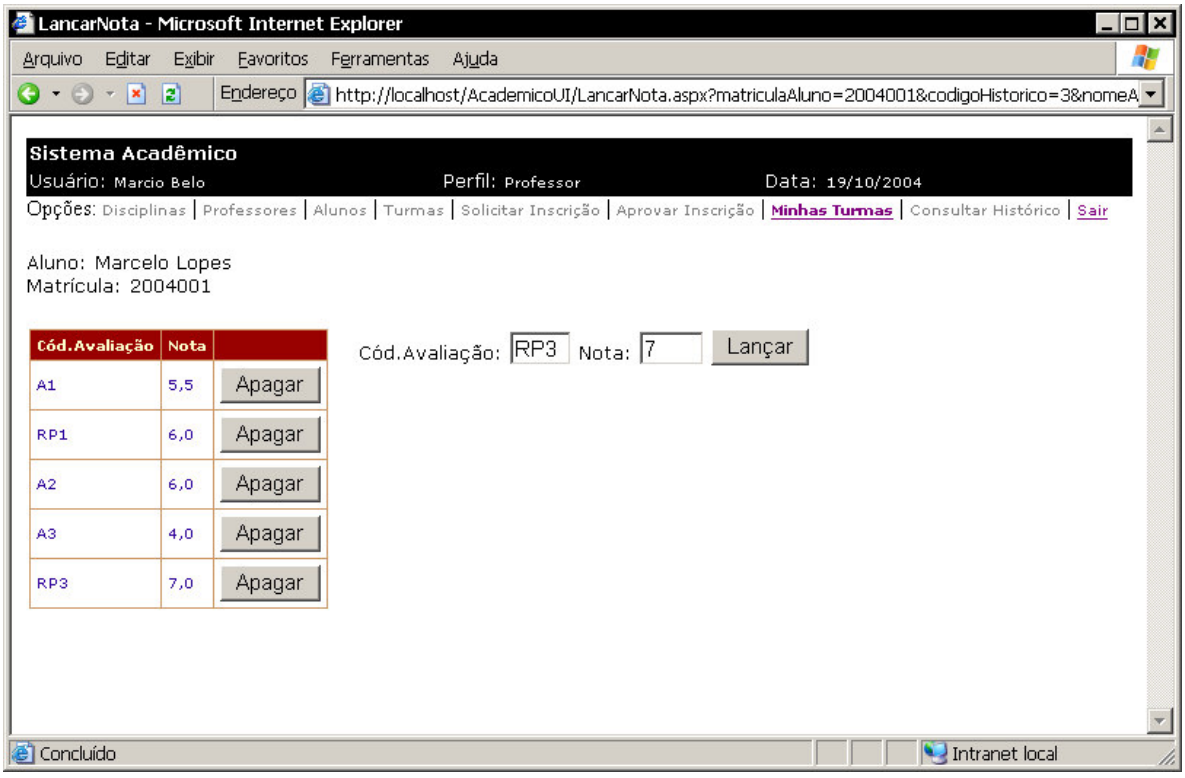
Tela TurmasProfessor.aspx



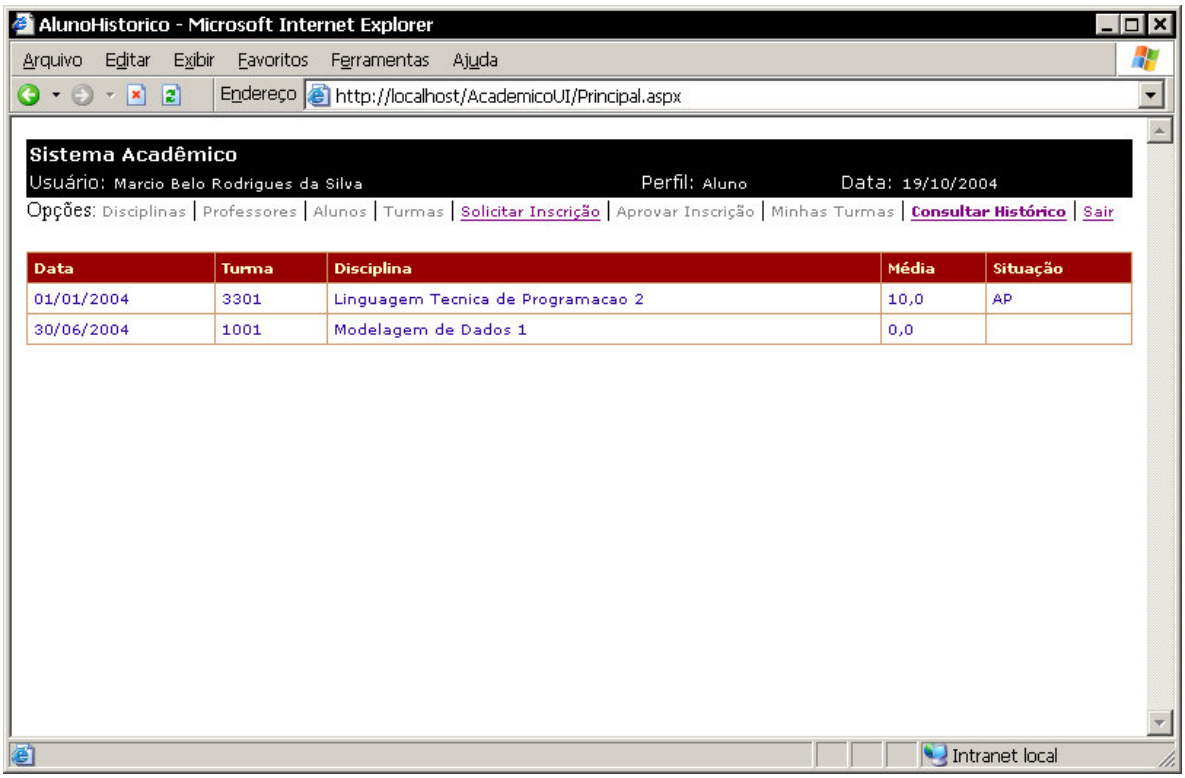
Tela AlunoTurmaExibe.aspx



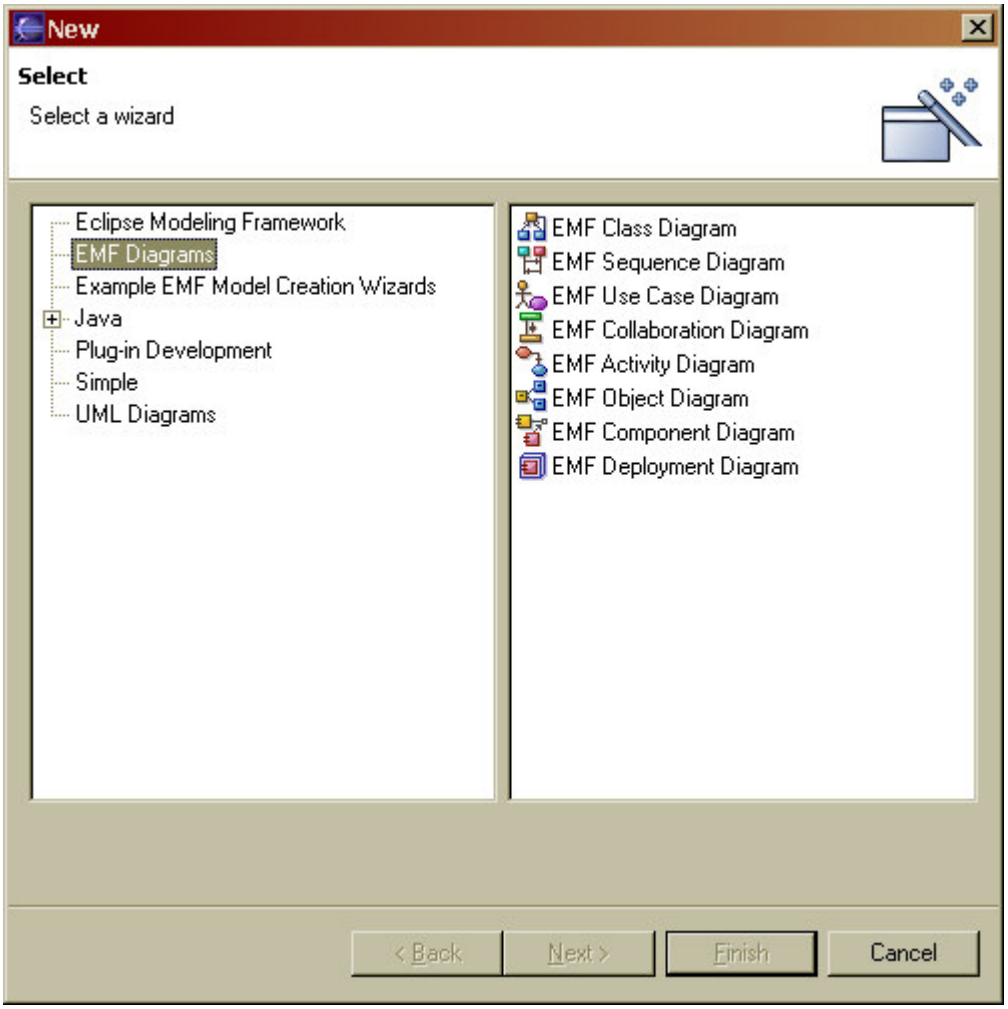
Tela LancarNota.aspx



Tela AlunoHistorico.aspx

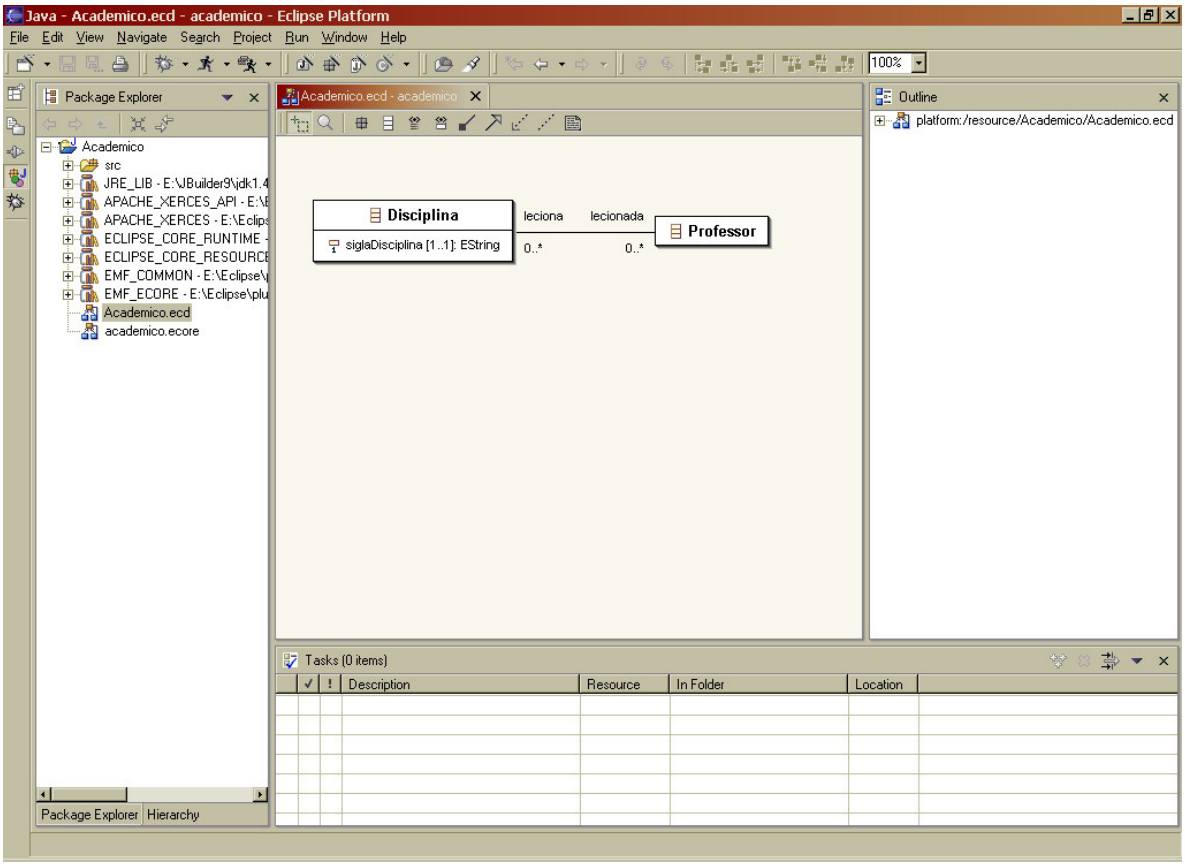


APÊNDICE C - ECLIPSE E O PLUG-IN ECLIPSE UML



A tela acima mostra que o plug-in Eclipse UML disponibiliza todos os modelos da UML.

A seguir é mostrado um pequeno exemplo de um diagrama de classes no eclipseUML:



O conteúdo do modelo é persistido em XMI. Para exemplificar, o pequeno modelo mostrado na tela acima é descrito pelo seguinte XMI:

Arquivo: Academico.ecd

```
<?xml version="1.0" encoding="UTF-8"?>
<editmodel:ClassDiagramEditModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:editmodel="editmodel.xmi"
size="730,377" id="platform:/resource/Academico/academico.ecore#/" metadata="emf-1.0"
initialized="true" showWireOptions="1">
  <children xsi:type="editmodel:ClassEditModel" location="32,55" size="177,57"
id="platform:/resource/Academico/academico.ecore#//Disciplina"
runTimeClassModel="siglaDisciplina">
    <children xsi:type="editmodel:CompartmentEditModel">
      <children xsi:type="editmodel:AttributeEditModel"
id="platform:/resource/Academico/academico.ecore#//Disciplina/siglaDisciplina">
        <model href="academico.ecore#//Disciplina/siglaDisciplina"/>
      </children>
    </children>
    <children xsi:type="editmodel:CompartmentEditModel"/>
    <children xsi:type="editmodel:CompartmentEditModel"/>
    <sourceConnections xsi:type="editmodel:AssociationEditModel"
connectionRouterKind="Manual" source="//@children.0" target="//@children.1"
```

```
targetEnd="//@children.0/@sourceConnections.0/@children.1"
sourceEnd="//@children.0/@sourceConnections.0/@children.0">
      <children xsi:type="editmodel:AssociationEndEditModel" location="176,3"
id="platform:/resource/Academico/academico.ecore#//Professor/leciona"
attachSource="true" navigable="true"
multiplicityLabel="//@children.0/@sourceConnections.0/@children.0/@children.1"
roleLabel="//@children.0/@sourceConnections.0/@children.0/@children.0">
        <children xsi:type="editmodel:LabelEditModel" size="35,14" fontInfo="Arial-8-0"
anchorKind="FirstPart"/>
        <children xsi:type="editmodel:LabelEditModel" size="17,14" fontInfo="Arial-8-0"
anchorKind="FirstPart"/>
        <model href="academico.ecore#//Professor/leciona"/>
      </children>
      <children xsi:type="editmodel:AssociationEndEditModel"
id="platform:/resource/Academico/academico.ecore#//Disciplina/lecionada"
navigable="true"
multiplicityLabel="//@children.0/@sourceConnections.0/@children.1/@children.1"
roleLabel="//@children.0/@sourceConnections.0/@children.1/@children.0">
        <children xsi:type="editmodel:LabelEditModel" size="47,14" fontInfo="Arial-8-0"
anchorKind="LastPart"/>
        <children xsi:type="editmodel:LabelEditModel" size="17,14" fontInfo="Arial-8-0"
anchorKind="LastPart"/>
        <model href="academico.ecore#//Disciplina/lecionada"/>
      </children>
      <bendpoints secondRelativeDimension="-136,0" firstRelativeDimension="143,0"/>
    </sourceConnections>
    <model href="academico.ecore#//Disciplina"/>
    <classifierPreferences xsi:type="editmodel:EMFClassDiagramClassifierPreference"
attributeSorter="Natural" methodSorter="Natural"/>
  </children>
  <children xsi:type="editmodel:ClassEditModel" location="329,69" size="102,29"
targetConnections="//@children.0/@sourceConnections.0"
id="platform:/resource/Academico/academico.ecore#//Professor" runTimeClassModel="">
    <children xsi:type="editmodel:CompartmentEditModel"/>
    <children xsi:type="editmodel:CompartmentEditModel"/>
    <children xsi:type="editmodel:CompartmentEditModel"/>
    <model href="academico.ecore#//Professor"/>
    <classifierPreferences xsi:type="editmodel:EMFClassDiagramClassifierPreference"
attributeSorter="Natural" methodSorter="Natural"/>
  </children>
  <model href="academico.ecore#/"/>
  <classDiagramPreferences xsi:type="editmodel:EMFClassDiagramPreference"
attributeSorter="Natural" methodSorter="Natural"/>
</editmodel:ClassDiagramEditModel>
```

Arquivo: Academico.ecd

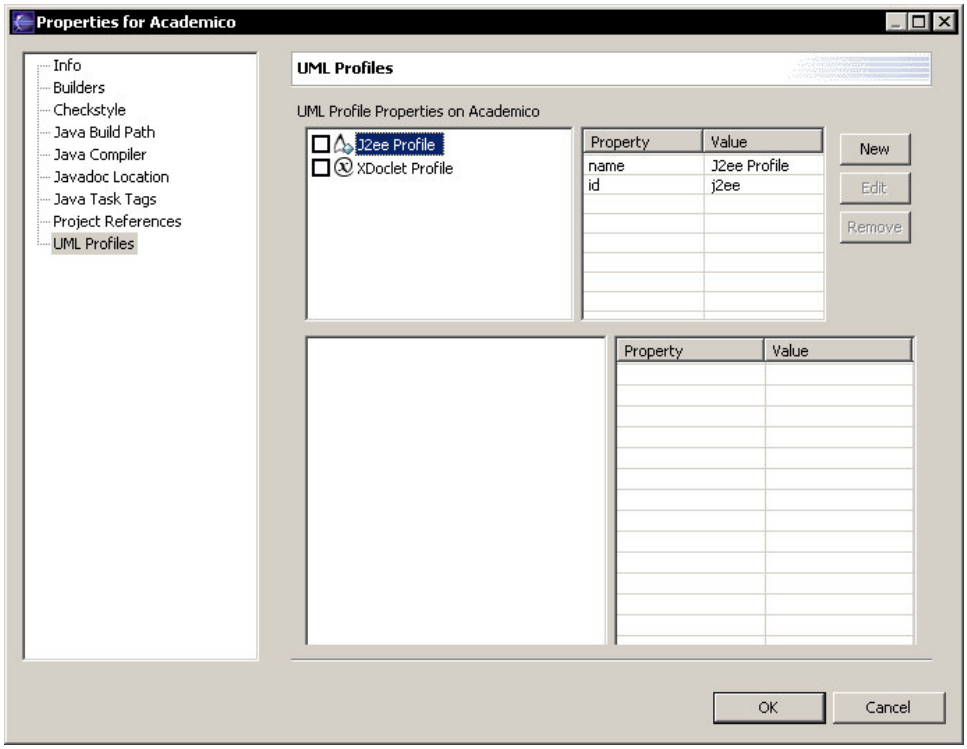
```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="academico"
    nsURI="http://academico" nsPrefix="academico">
  <eClassifiers xsi:type="ecore:EClass" name="Disciplina">
    <eReferences name="lecionada" eType="#//Professor" upperBound="-1"
eOpposite="#//Professor/leciona"/>
    <eAttributes name="siglaDisciplina" eType="ecore:EDataType
http://www.eclipse.org/emf/2002/Ecore#//EString"
        lowerBound="1"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Professor">
```

```
    <eReferences name="leciona" eType="#//Disciplina" upperBound="-1"
eOpposite="#//Disciplina/lecionada"/>
  </eClassifiers>
</ecore:EPackage>
```

O eclipseUML utiliza o framework EMF, definido pela Eclipse.org.

Após várias tentativas, tive sucesso em instalar o EclipseUML Studio (anteriormente chamado de EclipseUML Enterprise Edition). Minha preocupação agora é com o prazo de validade do produto para avaliação, que vence no dia 10 de maio. Solicitei ao proprietário do plugin (OMONDO) uma autorização acadêmica de uso do produto. Na resposta eles me questionaram o intervalo de IPs dos computadores que usariam o plugin. Perguntei ao Prof.Vinod e forneci o intervalo. Estou agora na expectativa de receber a licença para utilização do produto sem preocupações com prazos.

O uso dessa versão Studio é fundamental o projeto, uma vez que, ao contrário de versão Free Edition, ela suporta o uso de UML Profiles. A seguir mostra o diálogo do produto onde é configurado o uso de um determinado UML Profile no projeto:



Repare que a ferramenta já fornece o UML Profile for J2EE, demonstrando que a ferramenta deve receber novos plug-ins com sua evolução.

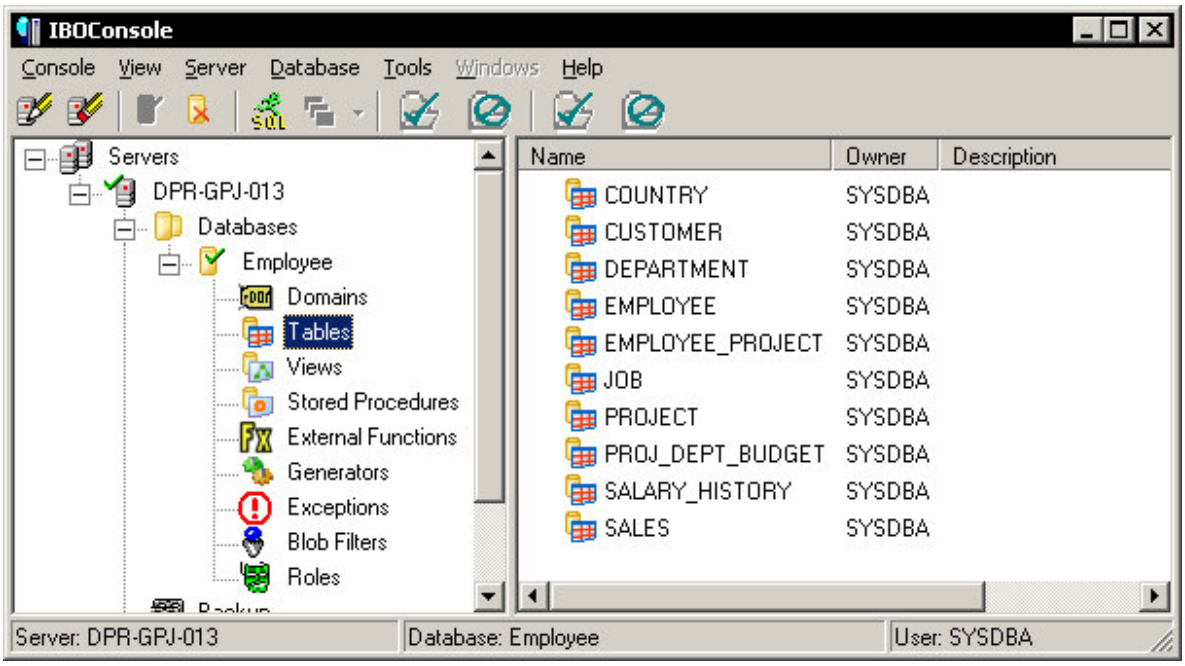
APÊNDICE D – FIREBIRD

É um banco de dados relacional que oferece suporte ao padrão ANSI SQL-92, ou pelo menos para maior parte dele. Ele roda em Linux, Windows e outras plataformas UNIX. Firebird oferece excelente concorrência, alta performance, e uma linguagem poderosa para criação de stored procedures e triggers; e é utilizado desde 1981 (segundo o site http://firebird.sourceforge.net/).

O FireBird possui duas versões de servidor: a Super Server – mais utilizada – e a Classic. A Classic Server suporta computadores multiprocessados. A Super Server roda em um único processo, abrindo threads para cada uma das conexões, portanto, é adequada para maioria das máquinas de um só processador.

É distribuído para uso gratuito, sob a licença GPL. Ao ser instalado, o servidor roda como um serviço no Windows 2000/XP e utiliza como padrão a porta 3050. Além disso, é rodado um serviço chamado Guardian que monitora o funcionamento do servidor e realiza procedimentos de correção em situações de falha.

Para administração do servidor existem várias ferramentas também gratuitas. Um exemplo delas é o IBOConsole. Abaixo é mostrada a tela desse aplicativo conectado a um servidor Firebird.



O FireBird e outras ferramentas relacionadas a ele podem ser encontradas no sítio www.firebase.com.br

BIBLIOGRAFIA

[Amb2003] Scott Ambler. Agile Database Techniques : Effective Strategies for the Agile Software Developer. Wiley. 2003.

[AP2004] Marcus Alanen, Ivan Porres. Change Propagation in Model-Driven Development Tool. WiSME@UML 2004, 3rd WorkShop in Software Model Engineering. Lisbon, Portugal. 2004.
Disponível em http://albini.xactium.com/wisme/papers.html

[Asp2004] Microsoft Corporation. Microsoft ASP.NET, www.asp.net. Acessado em outubro de 2004.

[Cwm2003] Object Management Group. Common Warehouse Metamodel Specification, Version 1.1. 2003.

[DEC2002] Digital Equipament Corporation. Database Language SQL, ISO/IEC 9075:1992. 2002.

[DGLRS2003] Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel. Modelware for Middleware. Em Middleware 2003 Companion. 2003.

[Fra2003] David S. Frankel. Model Driven Architecture - Applying MDA to Enterprise Computing. OMG Press. 2003.

[FS2000] Martin Fowler, Kendal Scott. UML Distilled – A Brief Guide to the Standard Object Modeling Language, Second Edition. Addison Wesley. 2000.

[FS2004] Frédéric Fondement, Raul Silaghi. Defining Model Driven Engineering Processes. WiSME@UML 2004, 3rd WorkShop in Software Model Engineering. Lisbon, Portugal. 2004.
Disponível em http://albini.xactium.com/wisme/papers.html

[JMK2004] Jochen M. Küster. Systematic Validation of Model Transformations. WiSME@UML 2004, 3rd WorkShop in Software Model Engineering. Lisbon, Portugal. 2004.

Portugal. 2004.
Disponível em http://albini.xactium.com/wisme/papers.html

[KW1998] Anneke Kleppe, Jos Warmer. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley. 1998.

[KWB2003] Anneke Kleppe, Jos Warmer, Wim Bast. MDA Explained The Model Driven Architecture: Practice and Promise. Addison-Wesley. 2003.

[LS2004] Miguel Pinto Luz, Alberto Rodrigues da Silva. Executing UML Models. WiSME@UML 2004, 3rd WorkShop in Software Model Engineering. Lisbon, Portugal. 2004.
Disponível em http://albini.xactium.com/wisme/papers.html

[Mash2004] Atof Mashkoor. Investigating Model Driven Architecture, Master Thesis. Department of Computing Science, Umea University, Umea, Sweden. 2004.

[MB2003] Stephen J. Mellor, Marc J. Balcer. Executable UML - A Foundation for Model-Driven Architecture. Addison-Wesley. 2003.

[Mda2004] Object Management Group. OMG Model Driven Architecture. http://www.omg.com/mda. Acessado em maio de 2004.

[Mof2002] Object Management Group. Meta Object Facility (MOF) Specification. Version 1.4. 2002.

[MSUW2004] Stephen J.Mellor, Kendall Scott, Axel Uhl, Dirk Weise. MDA Distilled, Principles of Model-Driven Architecture. Addison-Wesley. 2004.

[Net2004] Microsoft Corporation. Microsoft .NET Framework, www.microsoft.com/net. Acessado em outubro de 2004.

[Ocl2003] Object Management Group. UML 2.0 OCL Specification. pct/03-10-14. 2003.

[PJ2004] Ilia Petrov, Stefan Jablonski. Towards a Language For Querying OMG MOF-based Repository Systems. WiSME@UML 2004, 3rd WorkShop in Software Model Engineering. Lisbon, Portugal. 2004. Disponível em http://albini.xactium.com/wisme/papers.html

[Syb2005] Sybase. PowerDesigner. http://www.sybase.com/products/informationmanagement/powerdesigner, Acessado em janeiro de 2005.

[UDM2000] Rational. The UML and Data Modeling Whitepaper, TP-180 3/00. Disponível em www.rational.com. 2000.

[Uml2004] Object Management Group. UML Resource Page. www.uml.org. Acessado em setembro de 2004.

[WJ2004] Weerasak Witthawaskul, Ralph Johnson. An Object Oriented Model Transformer Framework based on Stereotypes. WiSME@UML 2004, 3rd WorkShop in Software Model Engineering. Lisbon, Portugal. 2004. Disponível em http://albini.xactium.com/wisme/papers.html

[Wsa2004] W3C. Web Services Activity, http://www.w3c.org/2002/ws/. Acessado em outubro de 2004.

[Xmi2003] Object Management Group. XML Metadata Interchange Specification v2.0. 2003.

[YBB2003] Chunmin Yang, Barrett R. Bryant, Carol C. Burt. Formal Methods For Quality Of Service Analysis In Component-Based Distributed Computing. Society for Design and Process Science. Integrated Design and Process Technology. IDPT. 2003. Disponível em http://www.cs.iupui.edu/uniFrame/pubs-openaccess/idpt2003Yang.pdf