UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ELIAS RICKEN DE MEDEIROS

NovaStudio: Gerador de código usando a Arquitetura Dirigida pelos Modelos (MDA)

Trabalho de Graduação.

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, novembro de 2009.

AGRADECIMENTOS

Primeiramente agradeço a Deus por todas as oportunidades e graças recebidas.

Agradeço à minha família, pelo carinho, atenção e apoio recebidos ao longo de todos esses anos de estudo.

Agradeço à Universidade Federal do Rio Grande do Sul, em especial ao Instituto de Informática pelo ensino de altíssima qualidade, além de permitir que eu realizasse sonhos os quais seriam extremamente difíceis de serem alcançados sem a sua ajuda. Agradeço em especial, a todos que ajudaram de alguma forma para a concretização do projeto SCISTEMA que possibilitou a complementação de meus estudos através de um intercâmbio de dois anos na França.

Agradeço ao Professor Dr. Cláudio Fernando Resin Geyer pela sua orientação e apoio durante a realização deste trabalho, assim como ao longo de toda a minha estadia na França.

Agradeço a toda a equipe NovaStudio, em especial Cédric Tran-Xuan e Emmanuel Rias, pela sua ajuda durante o meu estágio, o qual acabou resultando neste trabalho de graduação. Agradeço igualmente a equipe Adèle, em especial ao professor Dr. Philippe Lalanda, pelo seu apoio durante a realização do estágio.

Gostaria de agradecer a todos os meus amigos, os quais contribuíram para uma aula mais descontraída durante os meus anos de estudo na UFRGS.

Por fim, agradeço a todos que tenham contribuído de alguma forma para a minha formação.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AMDD	Agile Model-Driven Development
AOP	Programação orientada a aspectos – aspect oriented programming
CIM	Computation Independent Model
DOP	Domain-Oriented Programming
DSL	Linguagnes específicas de domínio – Domain-Specific Language
DSML	Linguagens de modelagem específicas de domínio - Domain-Specific Modeling Language
GPL	Linguagem generalista - General Purpose Language
MDA	Arquitetura Dirigida pelos Modelos - Model Driven Architecture
MDE	Model Driven Engineering – Engenharia Dirigida pelos Modelos
MIC	Model-Integrated Computing
MOF	Meta-Object Facility
MVC	Model-View-Controller
OMG	Object Management Group
PIM	Modelos Independentes de Plataforma - Platform Independant Models
POO	Programação Orientada a Objetos
PSM	Modelos dependentes de plataforma - Platform Specific Models
SF	Software Factories
UML	Unified Modelin Language
URI	Uniform Resource Identifier
XMI	XML Metadata Interchange

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A automação do processo de produção de software tem tornado-se cada vez mais uma necessidade devido à crescente complexidade dos sistemas de informação. Neste contexto, o produto NovaStudio do grupo Bull apóia-se na Arquitetura Dirigida pelos modelos (MDA – *Model Driven Architecture*), uma das principais variantes da metodologia Engenharia Dirigida pelos Modelos (MDE – *Model Driven Engineering*), para gerar automaticamente uma boa parte do código necessário às aplicações J2EE.

Existe um desejo de estender as funcionalidades do NovaStudio para acrescentar também o suporte à geração de código PHP e .NET. No entanto, a atual tecnologia utilizada no motor de geração não é totalmente adaptada à metodologia MDE, causando alguns inconvenientes. Neste contexto, um estudo realizado pela equipe NovaStudio apontou a solução de código aberto Acceleo como candidata a substituta do motor de geração. Desta forma este trabalho permite a validação da utilização do Acceleo como novo motor de geração para que ele possa ser utilizado, em seguida, para acrescentar o suporte à geração de código PHP e .NET.

Palavras-Chave: Engenharia Dirigida pelos Modelos, Arquitetura Dirigida pelos Modelos.

NovaStudio: Code generator by using MDA approach

ABSTRACT

The automation of software process production has become a necessity due to the increasing systems information's complexity. In this context, the Bull company's product NovaStudio is based on the approach Model Driven Architecture (MDA), one of the major Model Driven Engineering (MDE) initiatives, to automatically generate code for J2EE applications.

Bull wants to extend the NovaStudio's functionalities to also provide support for PHP and .NET code generation. However, the technology currently used in the code generation engine is not completely adapted to the MDA methodology. In this context, a survey conducted by NovaStudio team points the open source solution Acceleo as a candidate for the replacement of the code generation engine. In this way, this work validates the use of Acceleo as the new code generation engine so that it can be used to add support for PHP and .NET code generation.

Keywords: Model Driven Engineering, Model Driven Architecture

1 INTRODUÇÃO

Os sistemas de informação modernos têm tornado-se cada vez maiores e mais complexos devido ao grande número de informação e detalhes técnicos que devem ser considerados. As informações são expostas das formas mais variadas possíveis: elas podem apresentar-se de forma estruturada, semi-estruturada ou não estruturada; elas podem vir de uma página ou de um serviço web, de outra aplicação na rede local, de um dispositivo móvel, etc. Além disso, a aplicação deve levar em consideração uma vasta gama de detalhes técnicos: a diversidade de plataformas, a distribuição na rede, os aspectos associados à segurança, à disponibilidade e a outros aspectos não-funcionais, o caráter dinâmico dos dispositivos móveis que podem aparecer ou desaparecer a qualquer instante, etc. Neste contexto, mesmo utilizando as técnicas e os métodos tradicionais da Engenharia de Software, a complexidade destas aplicações pode colocar em perigo o sucesso do projeto. Desta forma, é necessário um novo paradigma que seja capaz de reduzir o problema da complexidade para atingir aplicações de melhor qualidade e dentro do prazo adequado.

A *Engenharia Dirigida pelos Modelos* (MDE – *Model Driven Engineering*) se apresenta como uma possível solução para contornar o problema da complexidade dos sistemas de informação modernos. Esta metodologia permite que desenvolvedor trabalhe num nível de abstração mais elevado, permitindo uma redução da complexidade de desenvolvimento e gerando ganhos de produtividade significativos. Os modelos obtidos num nível de abstração mais elevado são transformados em modelos equivalentes num nível de abstração mais baixo, até chegar a um artefato final (código aplicativo, por exemplo). Uma grande parte dessas transformações é realizada automaticamente assegurando a consistência entre a implementação e a sua especificação.

Neste contexto, o NovaStudio é uma ferramenta desenvolvida pelo grupo Bull S.A.S.[1] que utiliza a *Arquitetura Dirigida pelos Modelos* (MDA – *Model Driven Engineering*), uma das principais variantes da MDE, para gerar automaticamente uma boa parte do código necessário para aplicações J2EE.

1.1 Motivação e objetivos

O atual motor de geração do NovaStudio é baseado em *templates Velocity* do projeto *Apache Velocity*[2]. No entanto, o *Velocity* não foi concebido com o intuito de atender à metodologia MDE, o que acaba trazendo alguns inconvenientes, como será discutido mais adiante. Assim, como existe um desejo de estender as funcionalidades da ferramenta para também fornecer suporte à geração de código PHP e .NET, seria interessante fazer uso de uma tecnologia mais adaptada ao contexto da MDA e que contornasse os problemas encontrados com o *Velocity*.

---

[1] www.bull.com

[2] http://velocity.apache.org/

Um estudo realizado pela equipe do NovaStudio identificou a solução de código aberto *Acceleo*[3], como candidata para substituir os *templates Velocity* no motor de geração. O Acceleo é hospedado pelo consórcio *OW2*[4] e permite a geração de código em diferentes linguagens (PHP, Java, .NET) utilizando a abordagem MDA. Ele é nativamente integrado ao *Eclipse* e ao *Eclipse Modeling Framework* (EMF) e compreende um conjunto de ferramentas e editores permitindo a sua fácil adaptação a todo tipo de projeto ou tecnologia.

Desta forma, este trabalho de graduação possui como objetivo principal a verificação da viabilidade da utilização do Acceleo como novo motor de geração de código do NovaStudio. Este objetivo compreende três tarefas principais:

- *Identificação das diferenças potenciais entre Acceleo e o atual motor de geração do NovaStudio*: trata-se de um estudo inicial sobre as funcionalidades do Acceleo, assim como, sobre os formatos de arquivos aceitos em entrada. Esta etapa é essencial para garantir que nenhuma funcionalidade já fornecida pelo NovaStudio seja perdida e que os formatos de arquivos aceitos sejam os mesmos. Caso algum problema seja identificado, será necessário estudar a melhor maneira de contorná-lo.
- *Desenvolvimento de templates Acceleo para a geração de código*: consiste no desenvolvimento de *templates* Acceleo para a geração de código aplicativo conforme a arquitetura lógica definida pelo NovaStudio. O objetivo é utilizar a tecnologia Acceleo para gerar um código J2EE idêntico aquele atualmente gerado pelo NovaStudio.
- *Analise dos resultados obtidos*: a última etapa consiste em analisar os resultados obtidos nas duas etapas anteriores. Esta análise permitirá decidir se o Acceleo poderá ou não ser utilizado para a geração de código PHP e .NET. De fato, uma vez validado o novo motor de geração de código, a idéia é desenvolver novos *templates* Acceleo que permitam a geração de código para estas duas tecnologias.

O restante deste trabalho esta dividido da seguinte maneira:

- Capítulo 2: apresenta o estado da arte para a MDE, introduzindo os seus principais conceitos, benefícios e desafios. Além disso, ele apresenta resumidamente as principais variantes da MDE, com destaque para a MDA da OMG e a *Software Factories* da Microsoft.
- Capítulo 3: apresenta uma visão geral da ferramenta NovaStudio, incluindo suas principais funcionalidades, princípio de uso e arquitetura do código gerado. Ele apresenta, igualmente, uma visão geral do processo de geração de código usando *templates Velocity* e usando *templates Acceleo.*
- Capítulo 4: propõe uma modelagem para o novo motor de geração, já com a previsão da geração de código PHP e .NET. Juntamente com o capítulo 5, ele contém as contribuições deste trabalho.
- Capítulo 5: apresenta os detalhes referentes à implementação do novo motor de geração de código, assim como, as principais regras utilizadas para a criação dos *templates* de geração de código.

[3] http://www.acceleo.org/pages/home/en

[4] www.ow2.org

- Capítulo 6: contém a conclusão e considerações finais.

2 ENGENHARIA DIRIGIDA PELOS MODELOS

Este capítulo tem por objetivo apresentar o estado da arte da Engenharia Dirigida pelos Modelos (MDE – *Model Driven Engineering*). Ele aborda os principais conceitos, benefícios e desafios associados a esta metodologia.

2.1 Introdução

Hoje os sistemas informáticos estão se tornando cada vez mais complexos: eles devem operar distribuidamente, executar em diferentes plataformas, se comunicar com outras aplicações e dispositivos disponíveis em uma rede (seja ela fixa ou móvel), estar sempre disponíveis, etc. Apesar deste aumento de complexidade, uma boa parte dos sistemas ainda é desenvolvida de maneira *ad hoc*, sem nenhuma técnica de engenharia de software mais sofisticada. Mesmo quando um processo de software é utilizado, os modelos possuem um papel de simples documentação e são transformados em implementação de maneira manual, sendo geralmente abandonados após as primeiras fases do ciclo de desenvolvimento. Este processo de transformação manual do modelo de concepção em código aplicativo introduz uma complexidade acidental e como resultado obtemos um produto incoerente com a sua especificação e, freqüentemente, com maiores tempo e custo de desenvolvimento que o inicialmente previsto.

A Engenharia Dirigida pelos Modelos (MDE – *Model Drivien Engineering*) (FAVRE; ESTUBLIER; BLAY-FORNARINO, 2006) é uma metodologia de desenvolvimento de software cujo objetivo é aumentar o nível de abstração do processo de desenvolvimento de software e automatizar as tarefas manuais. Para atingir este objetivo as noções de modelo e transformação de modelo possuem um papel fundamental. Por exemplo, Bézivin (2004) dá ao conceito de modelo em MDE a mesma importância que o conceito de objeto tem na programação orientada a objetos (POO).

A metodologia MDE propõe que os modelos tenham o papel principal ao longo de todo o processo de desenvolvimento (BEZIVIN J; GERBE, 2001). Essa mudança de perspectiva permitiria uma melhora significativa da qualidade do software, reduziria a sua complexidade e aumentaria a reutilização dos componentes de software e a produtividade dos desenvolvedores (WEIGERT; WEIL, 2006; HAILPERN; TAR, 2006; SELIC, 2006). A idéia por trás da MDE é a utilização de modelos para aumentar o nível de abstração utilizado pelos desenvolvedores. Trabalhando em um nível mais abstrato, os desenvolvedores serão mais produtivos. Além disso, os modelos iniciais são transformados automaticamente num nível de abstração mais baixo, até chegar ao código aplicativo (ou qualquer outro artefato final). Esta automação de tarefas reduz a complexidade acidental e o tempo de concepção e também assegura a coerência entre a implementação e a sua especificação.

Segundo Schimidt (2006), a MDE apresenta-se como uma metodologia promissora para contornar a complexidade das plataformas, assim como, a incapacidade das linguagens

de terceira geração de reduzir esta complexidade e de exprimir os conceitos de um domínio de maneira eficaz.

A MDE combina:

- **Linguagens de modelagem generalistas** que utilizam um vocabulário único para exprimir modelos de vários domínios (UML[5], por exemplo).
- **Máquinas de transformações e geradores** que produzem vários tipos de artefatos, tais quais código aplicativo, descritores XML, ou modelos alternativos. Este processo automático de transformação de modelos em artefatos assegura a coerência entre a especificação e a implementação. Diz-se, assim, que este processo é "correto por construção" (SCHIMIDT, 2006).
- **Linguagens de modelagem específicas de domínio** (DSML – *Domain-Specific Modeling Language*) que formalizam a estrutura, o comportamento e os requisitos de uma aplicação num domínio particular. As DSMLs são descritas através de meta-modelos que especificam as relações entre os conceitos do domínio em questão e as restrições associadas a estes conceitos (SCHIMIDT, 2006).

2.2 Terminologia e definições

Esta seção tem o objetivo de apresentar algumas terminologias e definições relacionadas à MDE encontradas na literatura.

Primeiramente, será apresentada a definição dada pelo Object Management Group (OMG) [6] à sua arquitetura chamada *Arquitetura Dirigida pelos Modelos* (MDA - *Model Driven Architecture*). A MDA pode ser considera como a primeira proposição efetiva do que hoje é chamado de *Engenharia Dirigida pelos Modelos*. O Object Management Group (2009) define a MDA da maneira seguinte:

> Baseado nos padrões estabelecidos da OMG, a MDA separa a lógica de negócios e a lógica de aplicação da plataforma à qual ela está associada. Modelos independentes de plataforma de uma aplicação ou funcionalidade e comportamento de um sistema integrado, construídos usando UML e outros padrões da OMG associados, podem ser transformados através da MDA em praticamente qualquer plataforma, aberta ou proprietária, incluindo *Web Services*, .NET, CORBA, J2EE e outros. (Tradução nossa).

Esta definição ressalta a importância da separação entre a lógica de negócios e os detalhes técnicos da plataforma de implementação. A idéia é a criação de modelos mais abstratos num primeiro momento para transformá-los, em seguida, em modelos de uma plataforma específica. Nota-se igualmente, a importância que a UML possui nesta arquitetura. A UML recebe, entretanto, algumas críticas relacionadas à sua não-adaptação aos princípios da metodologia MDE. Por esta razão, alguns pesquisadores desta área propuseram a generalização da MDA obtendo o que é chamado de *Engenharia Dirigida pelos Modelos*. Bézivin et al. (2004), introduzem a noção de Engenharia Dirigida pelos Modelos da seguinte maneira:

> A idéia inicial da OMG consistia em se apoiar no padrão UML para descrever separadamente as partes do sistema independentes de plataformas específicas (PIM

[5] UML - Unified Modeling Language (http://www.uml.org/).

[6] http://www.omg.org/

> ou Platform Independant Models) e as partes associadas às plataformas (PSM ou Platform Specific Models). Nos anos que seguiram, este projeto tornou-se mais ambicioso e evoluiu de maneira interna e de maneira externa.
>
> [...]
>
> De maneira externa, em seguida, a abordagem MDA torna-se uma variante particular a Engenharia Dirigida pelos Modelos. A MDE pode ser vista como uma família de abordagens que se desenvolvem ao mesmo tempo nos laboratórios de pesquisa e nas indústrias implicadas em grandes projetos de desenvolvimento de software. (Tradução nossa).

O termo em inglês *Model-driven development* (MDD) é freqüentemente utilizado como sinônimo de MDE. Assim, encontramos as duas definições seguintes. De acordo com Hailpern e Tarr (2006):

> Desenvolvimento dirigido pelos modelos (MDD) é uma abordagem da engenharia de software consistindo da aplicação de modelos e tecnologias de modelos para atingir um nível de abstração no qual os desenvolvedores criam e desenvolvem software, com o objetivo tanto de simplificar (tornando mais fácil) como de formalizar (padronizando, de maneira que a automação seja possível) as várias atividades e tarefas que compreendem o ciclo de vida de um software. (Tradução nossa).

De acordo com Selic (2006):

> Desenvolvimento dirigido pelos modelos é uma abordagem para o desenvolvimento de software na qual os modelos tornam-se os artefatos essenciais do processo de desenvolvimento, em vez de apresentarem apenas um papel não essencial de suporte. (Tradução nossa).

A definição de Hailpern e Tarr ressalta o aumento do nível de abstração proporcionado pelos modelos e as simplificação e formalização conseqüentes na realização de atividades relacionadas ao desenvolvimento de software. Já a definição de Selic mostra a mudança de paradigma necessária para a aplicação da metodologia: os modelos, que antes tinha apenas um papel secundário de simples documentação, devem agora ser reconhecidos como os artefatos mais importantes, os artefatos que guiam o processo de desenvolvimento.

Em resumo, uma iniciativa da indústria (a MDA) baseada em um conjunto de padrões da OMG, inclusive a UML, chegou aos laboratórios de pesquisa das universidades e de grandes empresas. Essa abordagem foi, então, generalizada para abstrair os padrões propostos pela OMG e, sob o nome de *Engenharia Dirigida pelos Modelos*, ela designa a idéia geral da utilização de modelos para o desenvolvimento de softwares.

2.3 Principais conceitos

Esta seção apresenta os principais conceitos associados à MDE.

2.3.1 Modelo

Um modelo é uma abstração de um conceito de um domínio qualquer. O seu objetivo é permitir um estudo simplificado de um contexto real. O modelo deve ser capaz de responder questões no lugar do sistema modelado (BEZIVIN; GERBE, 2001). À noção de modelo sempre é associada uma relação *RepresentaçãoDe* que determina o domínio concreto representado pelo modelo em questão (FAVRE; ESTUBLIER; BLAY-FORNARINO, 2006).

France e Rumpe (2007) distinguem duas classes principais de modelos:

- *Modelos de desenvolvimento*: são os modelos associados a uma abstração no nível do código. Por exemplo, os modelos de requisitos, de arquitetura, ou de implementação e de desenvolvimento.
- *Modelos de execução*: são os modelos associados a uma abstração de aspectos vindos do ambiente de execução.

Este trabalho tratará apenas a primeira classe de modelos.

2.3.2 Meta-Modelo

O objetivo do meta-modelo é descrever a semântica de um modelo, pode-se dizer que um meta-modelo é um modelo de um conjunto de modelos. A ligação entre o modelo e o meta-modelo é estabelecida através da conformidade do primeiro com relação ao segundo. Um modelo é conforme a um meta-modelo se ele pertence ao conjunto modelado por este meta-modelo (FAVRE; ESTUBLIER; BLAY-FORNARINO, 2006). Além disso, deve-se notar que podemos distinguir várias noções de conformidade, como por exemplo, a conformidade sintática e a conformidade semântica (BÉZIVIN ET AL., 2004).

2.3.3 Hierarquia de meta-modelos

Assim como a noção de meta-modelo, podemos imaginar um modelo que descreve a semântica de um meta-modelo, trata-se de um meta-meta-modelo. Para evitar uma propagação infinita do termo *meta*, os meta-meta-modelos têm a tendência de se auto descreverem. Temos desta maneira, três níveis de hierarquia de (meta) modelos:

- M3: meta-meta-modelo. Exemplos: MOF, linguagem XML.
- M2: meta-modelo. Exemplos: o meta-modelo da UML, um schema XML.
- M1: modelo. Exemplos: um modelo UML específico, um documento XML específico.

Podemos ainda acrescentar um quarto nível (M0) que representa o sistema real representado pelo modelo M1 (Figura 2.1).

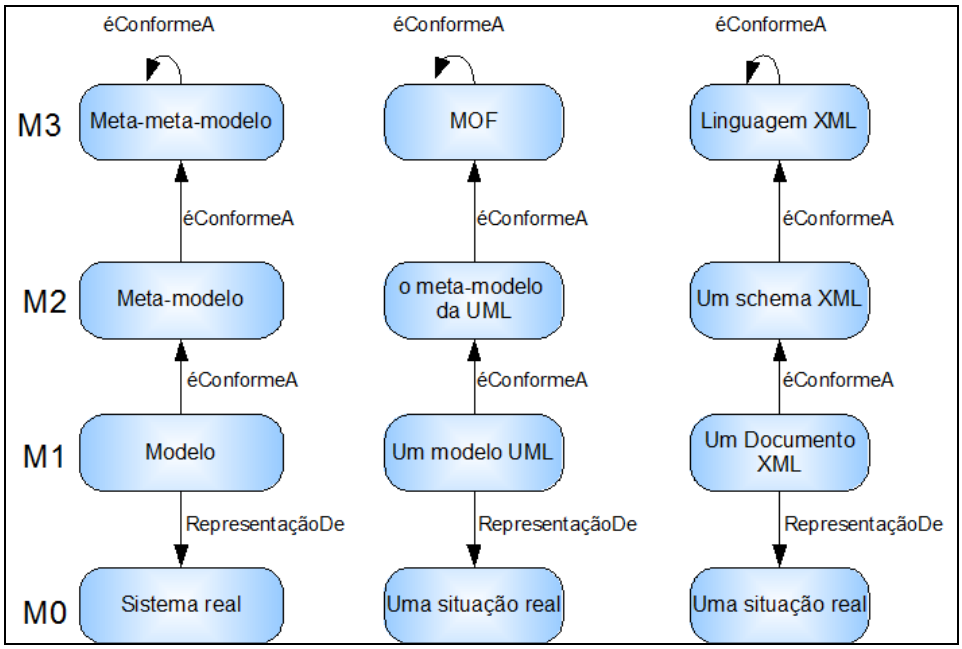

Figura 2.1: Hierarquia de (meta) modelos.

2.3.4 Transformação de modelos

Juntamente com os conceitos de modelo e de meta-modelo, o conceito de *transformação de modelo* faz parte do núcleo da MDE. Uma transformação de modelos estabelece uma relação entre dois modelos que representam diferentes níveis de abstração do sistema real modelado. Cada modelo sendo conforme a um dado meta-modelo, uma transformação de modelos recebe em entrada um modelo conforme ao meta-modelo origem e produz como saída o modelo equivalente para o meta-modelo destino (Figura 2.2). Como exemplo de transformação de modelo, podemos citar a transformação que recebe em entrada um modelo UML e produz na saída o documento XML correspondente.

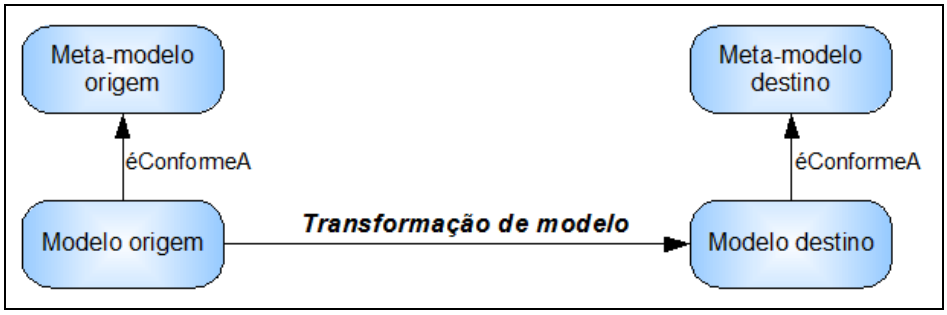

Figura 2.2: Transformação de modelo

2.4 Os benefícios da MDE

Olhando para o que está no núcleo da MDE, é possível identificar dois temas principais (SELIC, 2006):

- *Aumento do nível de abstração* das especificações aproximando-as do domínio funcional e deixando-as mais livres dos detalhes de implementação.
- *Aumento do nível de automação* fazendo automaticamente a ligação entre o modelo e a implementação (código gerado).

Desta forma, uma organização que adota a metodologia MDE, pode esperar dois benefícios principais:

- **Ganho de produtividade**: aumentando o nível de abstração, a MDE reduz ao mesmo tempo o esforço de desenvolvimento e a complexidade dos artefatos de software, por isso o ganho de produtividade (HAILPERN; TAR, 2006). O esforço de concepção é, obviamente, maior; entretanto, isto é compensado por um esforço de desenvolvimento mínimo graças à automação de determinadas tarefas que antes eram executadas manualmente. O número de inspeções necessárias para assegurar a qualidade do código desenvolvido é reduzido significativamente. A Motorola, por exemplo, obteve uma redução de 30% no tempo do ciclo de testes (WEIGERT; WEIL, 2006).
- **Software de melhor qualidade**: as transformações de modelo asseguram a consistência entre a especificação e o código gerado, uma vez que se trata de um processo automatizado (SCHIMIDT, 2006). Além disso, a MDE pode associar-se facilmente a métodos formais e a técnicas de simulação para assegurar a detecção de erros o mais cedo possível (WEIGERT; WEIL, 2006; SELIC, 2006; SCHIMIDT, 2006). Através de técnicas de simulação a Motorola, por exemplo, conseguiu melhorar a taxa de detecção de erros em

aproximadamente 30%, comparado com os métodos de inspeção tradicionais (WEIGERT; WEIL, 2006).

Uma pergunta freqüentemente feita pelas organizações que pretendem adotar a metodologia MDE concerne à eficácia do código gerado em termos de desempenho e de utilização da memória. A este respeito, Selic (2003) afirma que as atuais tecnologias de geração de código são capazes de produzir um código aplicativo com um desempenho e eficiência na utilização da memória entre 5% e 15% melhores ou piores se comparadas com o código equivalente escrito a mão. Além disso, a maioria dos compiladores modernos utilizam técnicas de otimização de desempenho na execução do código. Desta forma, na maioria dos casos a eficiência do código gerado não é um obstáculo. Na verdade, uma boa parte dos projetos MDE de grande sucesso está associado a sistemas embarcados ou de tempo-real (SELIC, 2006).

2.5 As principais iniciativas da Engenharia Dirigida pelos Modelos

Esta seção apresenta as principais propostas que utilizam a metodologia Engenharia Dirigida pelos Modelos

2.5.1 Arquitetura Dirigida pelos Modelos

A arquitetura Dirigida pelos Modelos (MDA) é a variante da MDE proposta e padronizada pela OMG. Essencialmente, a MDA define um conjunto de especificações que tem como objetivo separar a especificação do sistema da sua implementação. Para fazer isto, a MDA apóia-se tipicamente nos padrões MOF (*Meta-Object Facility*)[7], UML (*Unified Model Language*) e XMI (*XML Metadata Interchange*).

A MDA propõe a modelagem de sistemas segundo três pontos de vista (OMG, 2003):

- *Ponto de vista independente de cálculo*: este ponto de vista se preocupa com o ambiente do sistema e os seus requisitos, ocultando os detalhes de estrutura e de tratamento. Ele deve ser capaz de responder às questões relativas às funcionalidades esperadas do sistema representado. Os modelos desenvolvidos neste ponto de vista são chamados *Modelos Independentes de Cálculo* (CIM - *Computation Independent Model*).
- *Ponto de vista independente de plataforma*: este ponto de vista aborda as características do sistema que não mudam de uma plataforma à outra. Ele é representado pelos *Modelos Independentes de Plataforma* (PIM – *Platform Independent Model*).
- *Ponto de vista dependente de plataforma*: este ponto de vista combina o ponto de vista independente de plataforma com os detalhes associados a uma plataforma específica (J2EE, CORBA, Microsof .NET, etc.). Ele é representado pelos *Modelos Específicos de Plataforma* (PSM – *Platform Specific Model*). Os PSMs são obtidos, na maioria das vezes, por transformações automáticas a partir de PIMs.

[7] http://www.omg.org/mof/

2.5.2 Software Factories

Software Factories (SF) é a abordagem da Microsoft para a metodologia MDE. Greenfield e Short (2004) definem uma *Fabrica de software* (*Software Factory*) como uma linha de produção de programas que configura ferramentas, processos e *frameworks* para automatizar o desenvolvimento e a manutenção de variações de uma mesma família de produtos. Ao contrário da MDA que se apóia no padrão UML, *Software Factory* propõe a utilização de linguagens de domínio (DSL – *Domain Specific Language*) em vez de linguagens generalistas (GPL – *General Purpose Language*).

Software Factories apóia-se em dois conceitos principais (GREENFIELD SHORT, 2004):

- *Schema SF* (*Software factory schema*): ele define e categoriza os artefatos que compõem uma família de produtos de acordo com vários pontos de vista e estabelece as relações existentes entre esses diferentes artefatos. Ele pode ser comparado a uma receita que lista os ingredientes, os utensílios e o modo de preparo de um prato.
- *Template SF* (*Software factory template*): ele representa a implementação de *schema SF*. Em outras palavras, ele fornece todos os artefatos que foram definidos no *schema SF*. Ele pode ser visto como um recipiente contendo todos os ingredientes da receita (*schema SF*).

Uma comparação entre as abordagens MDA e *Software Factories* é fornecida por Demir (2006).

2.5.3 Outras Abordagens

É importante salientar que apesar da MDA e a *Software Factories* serem hoje as duas principais variações da MDE, elas não são as únicas. Existe uma série de outras proposições, como por exemplo, *Agile Model-Driven Development* (AMDD) [8], *Domain-Oriented Programming* (DOP) (THOMAS; BARRY, 2003) e *Model-Integrated Computing* (MIC) [9].

2.6 Modelagem de linguagens de domínio

Para atingir os seus dois principais objetivos (aumentar o nível de abstração e de automação), a MDE precisa exprimir os conceitos associados ao domínio funcional que o sistema representa. De acordo com Kelly (2004), a única maneira de gerar completamente o código aplicativo a partir de modelos é tornar a linguagem de modelagem e os geradores específicos ao domínio representado. Neste contexto existem duas grandes abordagens que serão detalhadas nas seções seguintes: utilização de extensões das *linguagens generalistas* e utilização de *linguagens específicas de domínio* (ou *linguagens dedicadas*).

2.6.1 Linguagens generalistas

As linguagens generalistas (GPLs - *General purpose languages*) são concebidas para representar o maior número possível de domínios. Isso possui a vantagem de fornecer um ambiente único e padronizado para qualquer domínio. Entretanto, elas recebem diversas

[8] http://www.agilemodeling.com/

[9] http://www.isis.vanderbilt.edu/research

críticas (HAILPERN; TAR, 2006; FRANCE ET AL., 2006; AMBLER, 2009), sobretudo no que diz respeito à sua complexidade e à sua difícil adaptação ao domínio representado.

Considere, por exemplo, a UML (que é um exemplo clássico de linguagem pertencente a este grupo) assim como ela é fornecida. Ela permite estabelecer ligações entre tipos de classes quaisquer, mesmo que estas ligações não façam nenhum sentido no domínio funcional representado. Para contornar este problema, a UML utiliza o conceito de *Perfil UML*[10] que permite a adaptação da linguagem a um domínio específico. No entanto, esse mecanismo de extensão apresenta alguns inconvenientes:

- A falta de uma semântica precisa constitui um obstáculo para a utilização de métodos formais (HAILPERN; TAR, 2006; FRANCE; RUMPE, 2007).
- As ferramentas atuais não fornecem suporte apara efetuar manipulações no meta-modelo da UML, como por exemplo, remover as partes que não serão utilizadas para o domínio modelado (FRANCE; RUMPE, 2007; DALGARNO; FOWLER, 2008).
- Todos os tipos de diagramas tem restrições baseadas na semântica da UML (DALGARNO; FOWLER, 2008).

Apesar de todos estes problemas aparentes, a utilização de perfis UML é largamente utilizada na indústria, pois produzir um novo perfil exige pouco esforço. Esta é a abordagem adotada pelo NovaStudio, por exemplo.

2.6.2 Linguagens específicas de domínio

Diferentemente das linguagens generalistas, as linguagens específicas de domínio (DSL - *Domain specific language*) são concebidas especialmente para representar um domínio muito específico, o que permite um importante ganho de produtividade comparado às GPLs (MERNIK; HEERING; SLOANE, 2005). Desta forma, as DSLs apresentam as seguintes vantagens:

- *Ganho de produtividade e redução do custo de manutenção*: por apresentar o mesmo vocabulário utilizado pelos especialistas do domínio que podem, assim, facilmente criar ou modificar um modelo (MERNIK; HEERING; SLOANE, 2005). De acordo com Kelly (2004) a utilização de DSLs permite ganhos de produtividade entre 500% e 1000% contra os modestos 35% obtidos pelas abordagens baseadas sobre UML.
- *Análise e verificação do modelo*: por ser reduzida em tamanho e complexidade, a realização de tais tarefas em DSLs é muito mais simples se comparada às suas realizações em GPLs (MERNIK; HEERING; SLOANE, 2005; DALGARNO; FOWLER, 2008).
- *Reuso*: DSLs são uma boa solução ao problema do reuso, não apenas no plano técnico, mas também no nível de arquitetura e de concepção (FEIKAS, 2006).

As DSLs apresentam, entretanto, alguns inconvenientes que impedem a sua adoção massiva e deixa a decisão do desenvolvimento de uma nova DSL extremamente difícil:

- O desenvolvimento de DSLs é difícil, pois exige um grande conhecimento do domínio e da linguagem de desenvolvimento (MERNIK; HEERING;

[10] http://www.omg.org/technology/documents/profile_catalog.htm

SLOANE, 2005). Além disso, cada DSL necessita suas próprias ferramentas (editor, verificador, gerador, etc.) (FRANCE; RUMPE, 2007), o que, sem a ajuda de uma ferramenta de apoio, pode exigir um esforço de desenvolvimento importante.

- As técnicas para o desenvolvimento de DSLs são mais variadas que as utilizadas para o desenvolvimento de GPLs exigindo, assim, uma análise atenta dos fatores em causa (MERNIK; HEERING; SLOANE, 2005).

2.7 Ferramentas de meta-edição

Como foi apresentado na seção anterior, o desenvolvimento de uma DSL é um processo complexo. Desta forma, o objetivo das ferramentas de meta-edição é tornar esta tarefa mais simples. O princípio geral destas ferramentas é gerar todo ambiente necessário (editor, verificador, gerador, etc.) para a utilização de uma DSL a partir da sua descrição (MERNIK; HEERING; SLOANE, 2005).

Atualmente as três ferramentas de meta-edição mais utilizadas são o conjunto de *plug-ins* do projeto *Eclipse Modeling* [11], o *MetaEdit+* [12] e o *Microsoft DSL Tools* [13]. Comparações entre elas são fornecidas por Kelly (2004) que compara *Eclipse EMF/GMF* e *MetaEdit+* e por Pelechano et al. (2006) que comparam *Microsoft DSL Tools* e *Eclipse Modeling Plug-ins*. Finalmente, defendendo a idéia de que modelos concebidos em meta-editores diferentes devem ser capazes de comunicar-se, Bézivin (2005) estabelece uma correspondência entre *Microsoft DSL Tools* e *Eclipse Modeling Framework* (EMF).

2.8 Desafios

Apesar do progresso obtido nos últimos anos, a MDE ainda é uma metodologia de produção de software relativamente recente e enfrenta vários desafios antes de poder ser utilizada mais amplamente na indústria. France e Rumpe (2007) agrupam esses desafios em três categorias: desafios de *linguagem de modelagem*, desafios de *separação de preocupações* e desafios de *manipulação e gestão de modelos*. As seções seguintes apresentam os principais desafios de acordo com cada categoria.

2.8.1 Desafios de linguagem de modelagem

Trata-se dos desafios ligados à criação e à utilização das abstrações em linguagens de modelagem e à análise rigorosa de seus modelos. Esta categoria consiste de dois desafios principais (FRANCE; RUMPE, 2007):

- A *abstração*: agrupa as questões ligadas ao fornecimento de suporte à criação e à manipulação de abstrações utilizando uma linguagem.
- A *formalidade*: agrupa as questões associadas à semântica do modelo, de maneira que se possa dar suporte às manipulações formais.

[11] http://www.eclipse.org/modeling/.

[12] http://www.metacase.com/

[13] http://code msdn microsoft.com/DSLToolsLab

Para enfrentar esses desafios, duas grandes abordagens são propostas: *linguagens generalistas* e *linguagens específicas de domínio*[14]. O principal desafio para os desenvolvedores de GPLs extensíveis é identificar um pequeno conjunto de conceitos capaz de exprimir uma grande variedade de problemas. Enquanto que para os desenvolvedores de DSLs, o desafio é fornecer um conjunto de ferramentas eficazes para a modelagem e a gerência da comunicação entre os conceitos representados em diferentes DSLs (FRANCE; RUMPE, 2007).

2.8.2 Desafios de separação de preocupações

Trata-se dos desafios ligados à modelagem de sistemas que devem utilizar vários pontos de vista e diferentes linguagens. O principal desafio é a integração das diferentes visões. Uma possível solução para este problema seria a utilização de um método para descrever as relações entre os conceitos definidos em diferentes visões. Uma outra abordagem seria a utilização de programação orientada a aspectos (AOP – *aspect-oriented programming*) (FRANCE; RUMPE, 2007).

Na abordagem MDA, esta separação de preocupações é representada pelos conceitos de PIM e PSM.

2.8.3 Desafios de manipulação e gestão de modelos

Trata-se de desafios ligados à criação, análise e utilização de transformações de modelos, assim como, aqueles associados ao suporte de retro-engenharia e à utilização de modelos durante a execução. O primeiro grande desafio diz respeito à propagação de modificações de um nível de abstração a todos os outros. Por exemplo, como refletir as modificações do código aplicativo no modelo que o gerou. As ferramentas atuais fornecem apenas um suporte muito básico a esta questão (FRANCE; RUMPE, 2007) e algumas críticas dizem que a MDE corre o risco de apenas deslocar a complexidade do desenvolvimento de programas em vez de removê-los (HAILPERN; TAR, 2007).

O segundo desafio esta ligado aos geradores de código aplicativo, na medida em que é necessário prever uma maneira de integrar o código manual e o código herdado ao código gerado (FRANCE; RUMPE, 2007).

Finalmente, o terceiro desafio diz respeito à utilização de modelos para a supervisão e a gestão de diversos aspectos de um sistema em tempo de execução. As pesquisas nesta área são relativamente recentes (FRANCE; RUMPE, 2007).

2.9 Considerações finais

Diante da complexidade crescente dos sistemas atuais, a MDE se apresenta como uma abordagem promissora para produzir aplicações de boa qualidade, num tempo eficaz. Várias aplicações práticas da MDE junto a grandes empresas mostram a sua viabilidade prática e a importância da continuidade das pesquisas na área.

Aproximadamente dez anos após a sua aparição, a MDE ainda está na sua infância, o que significa que existem ainda vários temas de pesquisa que podem trazer, nos próximos anos, um melhora significativa na maneira como a MDE é realizada.

---

[14] Veja seção 2.6.

3 NOVASTUDIO

O produto NovaStudio[15] do grupo Bull S.A.S. fornece um ambiente de desenvolvimento baseado em *plug-ins* Eclipse fazendo interface com uma ferramenta de modelagem UML através de arquivos XMI. O objetivo do NovaStudio é aumentar a produtividade das equipes de desenvolvimento através da abordagem MDA, gerando automaticamente uma boa parte do código necessário a aplicações J2EE.

As seções que seguem dão uma visão geral da ferramenta.

3.1 Funcionalidades

O NovaStudio fornece as seguintes funcionalidades:

- Geração de código de uma aplicação J2EE a partir de diagramas de classes UML utilizando a arquitetura MDA. Podendo suportar também a abordagem *bottom-up* que permite a realização de retro-engenharia de uma base de dados até os diagramas de classes nas ferramentas de modelagem. Esses diagramas podem, então, ser utilizados para gerar o código aplicativo utilizando a arquitetura MDA.
- Suporte a diferentes tipos de *frameworks*: WEB ou J2EE, JDK 1.4 ou JDK 1.5, Hibernate, EJB3 ou JDO.
- Suporte SOA: o NovaStudio oferece uma visão permitindo a gerência do registro de serviços de negócios de uma empresa, e uma visão permitindo configurar um acesso para diferentes tipos de Sistemas de Informação Executivos de empresas e fornecedores.

Além disso, o NovaStudio apresenta a vantagem de ser completamente integrado ao Eclipse, que é um ambiente de desenvolvimento muito utilizado pelas por empresas e laboratórios de pesquisa. Os *frameworks* definidos são utilizados pelo setor de serviços da Bull e são construídos pelo retorno de experiências de arquitetos de projeto. Finalmente, a implementação ou a escolha de um *framework aplicativo* não é estruturante, ou seja, a partir de uma mesma modelagem, pode-se escolher a qualquer momento outra implementação.

3.2 Concepção e Desenvolvimento

No processo de desenvolvimento de uma aplicação utilizando o NovaStudio distinguem-se dois papéis principais: o arquiteto e o desenvolvedor. O arquiteto utiliza uma ferramenta de modelagem UML para definir os objetos de negócio e os serviços a serem

[15] http://www.bull.com/fr/middleware/novastudio.php

implementados. O desenvolvedor, por sua vez, utiliza o NovaStudio para lançar o processo de geração do código a partir do modelo definido pelo arquiteto. Além disso, ele também é responsável por acrescentar o código específico da camada de negócios.

3.2.1 Etapas do desenvolvimento de uma aplicação

Esta seção apresenta passo a passo o processo de concepção de uma aplicação utilizando o NovaStudio.

a) **Modelagem:** o arquiteto cria o modelo com a ajuda de uma ferramenta de modelagem UML e o exporta em formato XMI (Figura 3.1).

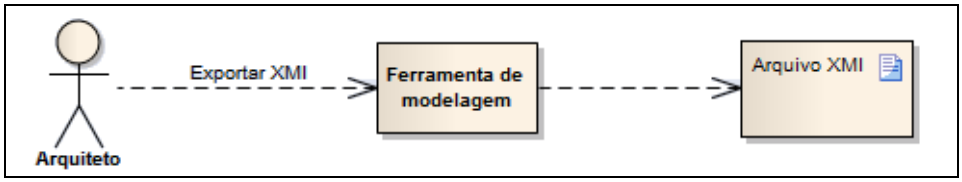

Figura 3.1: Etapa 1 - Modelagem

b) **Conversão em modelo específico:** o desenvolvedor utiliza o arquivo XMI exportado pelo arquiteto para gerar um novo arquivo XMI baseado no meta-modelo específico do NovaStudio (arquivo de extensão "*.generation"). Para isto, ele utiliza o menu "*Populate Generation Model*" do NovaStudio (Figura 3.2).

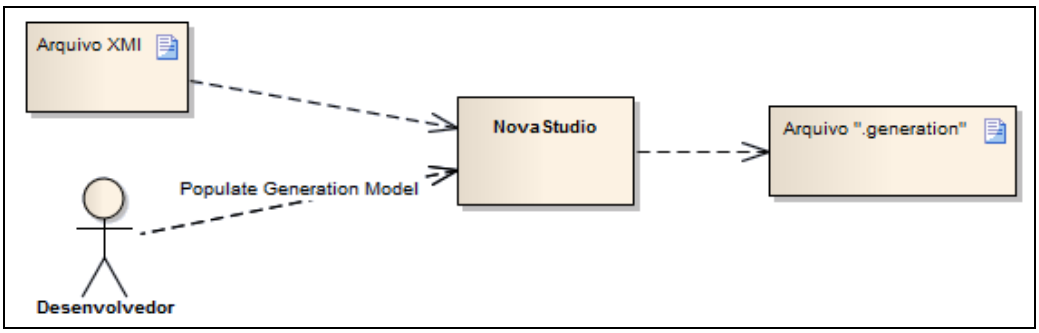

Figura 3.2: Etapa2 – Conversão em modelo específico

c) **Geração do código aplicativo:** a próxima etapa consiste na geração do código aplicativo a partir do arquivo "*.generation". Par isto, o desenvolvedor utiliza o menu "*Generate Aplication Code*" (Figura 3.3).

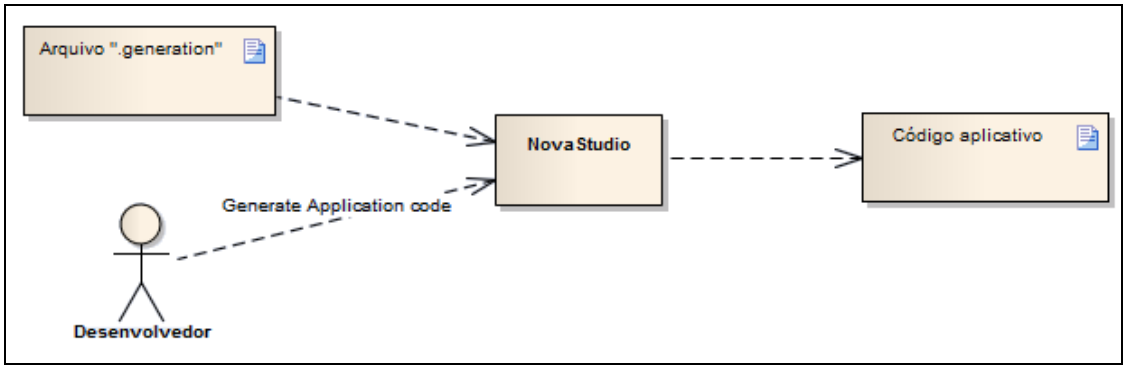

Figura 3.3: Etapa 3 - Geração do código aplicativo

d) **Acréscimo da lógica de negócios:** a última etapa consiste em acrescentar a lógica necessária à camada de negócios. Esta tarefa é realizada pelo desenvolvedor diretamente no editor do *Eclipse*.

3.3 Estereótipos e propriedades

Para a geração de código, o NovaStudio apóia-se sobre os conceitos de propriedades (*tagged values*) e estereótipos da UML. Uma *propriedade* é constituída de uma chave e um valor, e é utilizada para acrescentar uma informação complementar a um elemento do modelo. Podemos utilizar, por exemplo, a propriedade "*(AttributeId, True)*" para indicar que um atributo é o identificador único de uma entidade. Já um *estereótipo* é uma extensão de um elemento existente ao qual são acrescentadas informações associadas a um domínio específico. Isso é feito tipicamente através do uso de propriedades. No NovaStudio, podemos utilizar, por exemplo, uma classe com o estereótipo "*Entity*" para indicar que se trata de uma entidade, sendo que este estereótipo já contém todas as propriedades necessárias para esta entidade.

3.4 Arquitetura lógica do código J2EE

A arquitetura lógica J2EE suportada pelo NovaStudio é baseada na separação em camadas (figura abaixo).

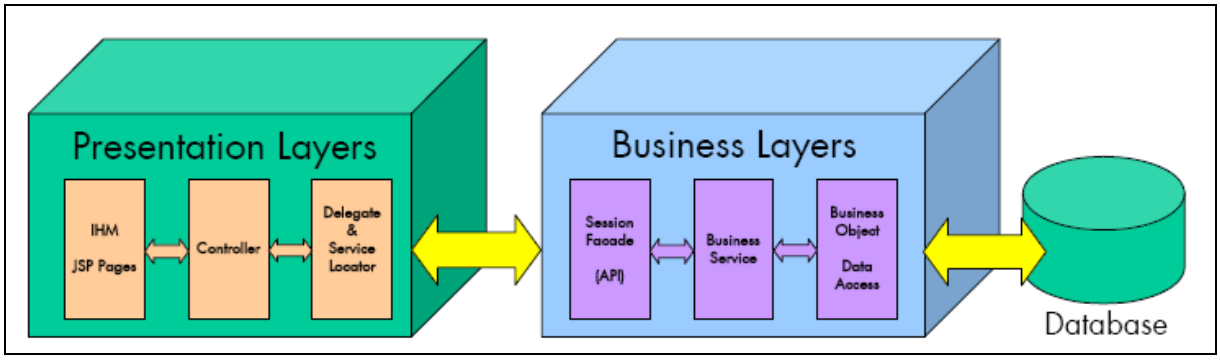

Figura 3.4 : Arquitetura lógica J2EE

A arquitetura separa aplicação em três componentes principais: a camada de apresentação, a camada de negócios e a base de dados. Esta separação é lógica, mas também pode ser física: em função da plataforma utilizada, as camadas podem executar em um único servidor ou em servidores separados.

3.4.1 Arquitetura Modelo-Visão-Controle (MVC)

Antes de passar à descrição de cada camada, será apresentado de maneira resumida a abordagem arquitetural Modelo-Visão-Controle (MVC – *Model-View-Controller*) (KRASNER; POPE, 1988), no qual a divisão em camadas do NovaStudio é baseada. A figura abaixo mostra as interações entre os componentes da arquitetura:

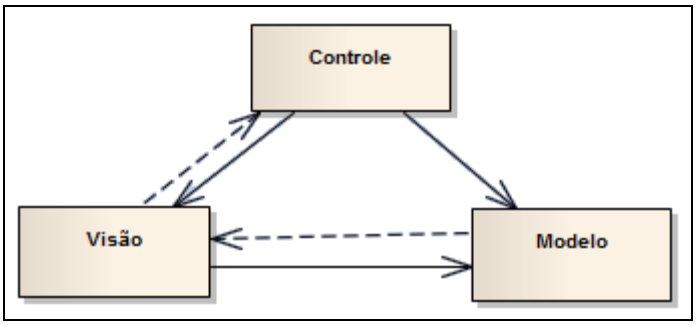

Figura 3.5: Arquitetura Modelo-Visão-Controle (MVC)

- **Modelo (*model*):** é responsável pela representação dos dados e da descrição da interação entre os diferentes objetos que compõem um domínio funcional. Ele fornece igualmente as maneiras de acessar e modificar o seu conteúdo.
- **Visão (*view*):** permite a exibição do modelo da maneira visual apropriada e permite a interação com o usuário. Nas aplicações web, a interface com o usuário é geralmente constituída por páginas HTML e pelo código que recupera os dados dinâmicos da página
- **Controle (*controller*):** centraliza o tratamento dos eventos obtidos a partir da interação com os usuários. Basicamente, ele recebe um evento disparado pelo usuário através da visão, chama o método correspondente do modelo e espera a resposta. Em função da resposta obtida ele sinaliza a visão que ela deve se atualizar ou simplesmente escolhe uma nova visão para mostrar o resultado.

Para entender melhor, considere o caso onde uma empresa deseja consultar os dados de um determinado empregado. Para isto, o usuário disporá de uma interface gráfica (visão) onde poderá informar os detalhes do empregado em questão. Uma vez lançada a pesquisa, um evento ativará o controle que chamará o método correspondente do modelo, passando as informações inseridas pelo usuário. Quando o controle receber uma resposta, ele encarregará a interface gráfica apropriada para a exibição do resultado.

3.4.2 Componentes da camada de apresentação

Nesta subseção serão apresentadas as diferentes subcamadas que compõem a camada de apresentação (*Presentation Layers*).

- **Interface gráfica (IHM)**: corresponde à visão da arquitetura MVC.
- **Controller**: corresponde ao controle da arquitetura MVC.
- **Business Delegate**: este padrão (SUN MICROSYSTEMS, 2009-a) orquestra os componentes distribuídos e o tratamento de exceção, além de substituir a interface do componente de negócios por uma interface mais simples que é utilizada pela interface gráfica com o usuário. Juntamente com o *Service Locator*, que será apresentado a seguir, o *Business Delegate* permite que a interface gráfica utilize os serviços de negócios de maneira independente da implementação e da localização da sessão *façade*.
- **Service Locator:** este padrão (SUN MICROSYSTEMS, 2009-b) centraliza os repositórios de serviços para os componentes distribuídos e fornece um ponto de controle centralizado. Ele também pode servir como uma *cache* para eliminar *lookups* redundantes.

3.4.3 Componentes da camada de negócios

Nesta subseção serão apresentas as diferentes subcamadas que compõem a camada de negócios (*Business Layers*).

- **Session Façade**: ela encapsula a complexidade das interações entre os objetos de negócios (entidades) e fornece uma camada de serviço que expõe apenas as interfaces necessárias aos clientes (SUN MICROSYSTEMS, 2009-c). Existe uma única *session façade* por domínio funcional: ela reagrupa todos os serviços de negócios oferecidos pelas diferentes classes deste domínio. Ela permite, desta forma, que os clientes acessem os serviços de negócio de uma maneira mais simples.
- **Business service**: esta camada contém a implementação dos serviços de negócios definidos na ferramenta de modelagem.
- **Business Objects**: a camada Data Access Objet (DAO) gerencia o acesso à base de dados (SUN MICROSYSTEMS, 2009-d). Os objetos de negócios (*Business Objects*) são as entidades definidas na ferramenta de modelagem e que são salvas no banco de dados.

Uma visão geral da interação entre as diferentes camadas é dada pelo diagrama de seqüência abaixo.

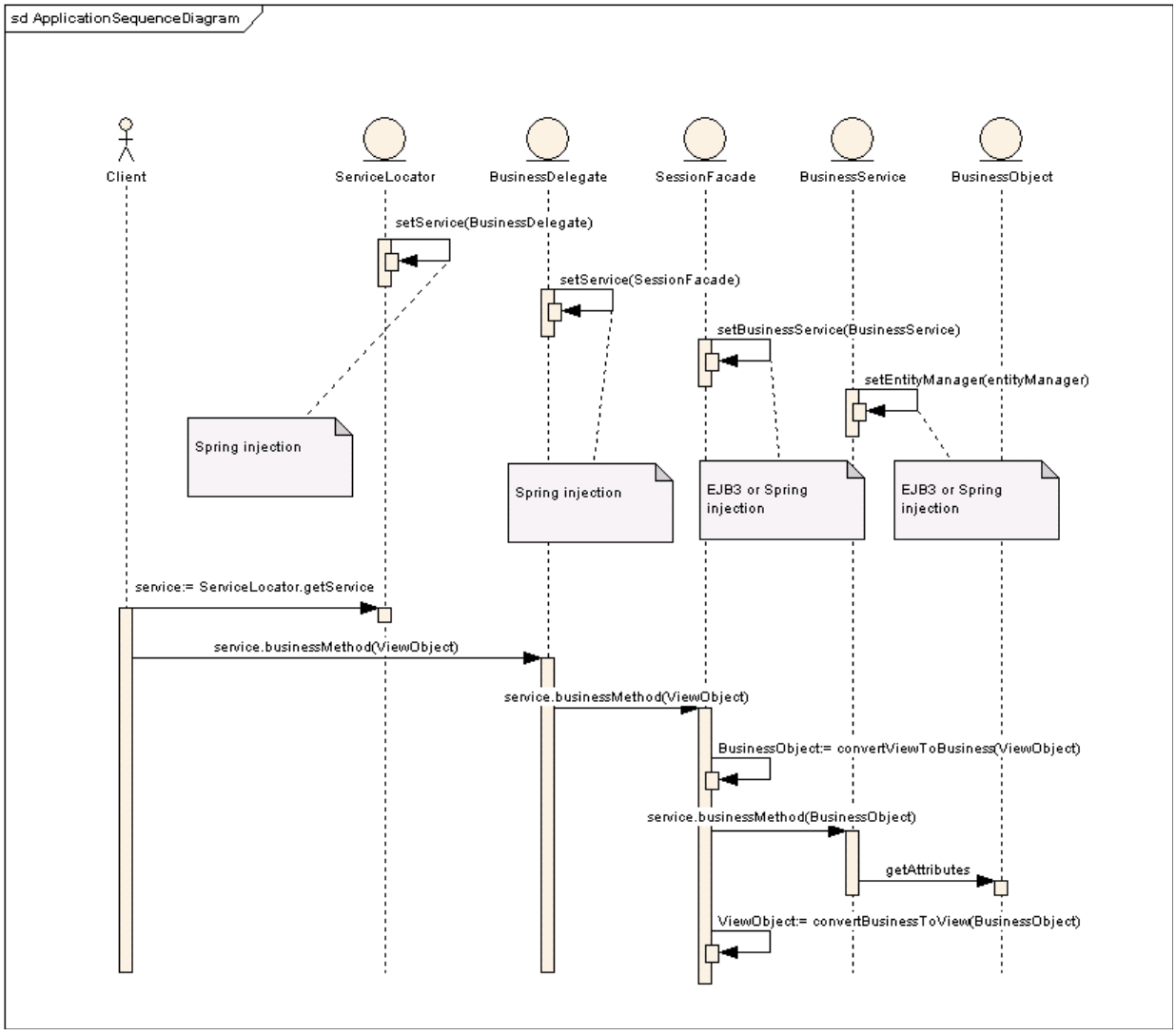

Figura 3.6: Interação entre as diversas camadas.

3.4.4 Domínios funcionais

As classes que compõem as camadas da arquitetura apresentada acima são divididas em um ou mais domínios funcionais. Um *domínio funcional* reagrupa um conjunto de classes que são inter-relacionadas. Em um modelo que representa uma escola, por exemplo, podemos definir três domínios funcionais: um para os professores, outro para os alunos e um terceiro para a lista de cursos disponíveis.

3.5 Relação entre os componentes

O código gerado pelo NovaStudio segue as seguintes regras:

- Existe uma *Session Façade* para cada domínio funcional. Ela contém os serviços de todas as classes de serviços de negócios que pertencem a este domínio funcional.
- Existe um serviço de negócios (*Business Service*) associado a cada objeto de negócios (*Business Object*). É função do desenvolvedor acrescentar a lógica específica a cada serviço de negócios.

- Um serviço de negócios pode acessar os objetos de negócios que estão associados a ele e a nenhum outro. No caso em que um serviço de negócios precise acessar um objeto que não lhe está associado, ele deve utilizar o serviço de negócios do objeto correspondente.

3.6 Implementação da arquitetura lógica

O NovaStudio suporta dois tipos de quadro aplicativos ou *frameworks* de desenvolvimento que podem ser utilizados conjuntamente: o *framework J2EE* e o *framework web*. O *framework J2EE* é utilizado para o desenvolvimento de aplicações que precisam das funcionalidades fornecidas pelo J2EE, tais quais transações e segurança. O *framework web* serve para o desenvolvimento de aplicações que podem ser colocadas em execução por servidores Web como o Apache Tomcat[16].

3.6.1 Escolhas de implementação

Uma vez escolhido o *framework* de implementação é necessário escolher a implementação técnica. Por exemplo, EJB3 para a persistência no *framework* J2EE e Hibernate 3 para a persistência no *framework Web*. As implementações suportadas pelo NovaStudio estão ilustrados na tabela a seguir.

Tabela 3.1 : Implementações suportadas pelo NovaStudio

	JDK	*Session Facade*	*Business Services*	*Persistent Framework*
Framework J2EE (EJB3)	1.5	EJB3	EJB3	EJB3
Framework J2EE (EJB2) 1	1.4	EJB2	EJB2	Hibernate 3
Framework J2EE (EJB2) 2	1.4	EJB2	EJB2	JDO (Speedo)
WEB framework (1.4) 1	1.4	Classes simples	Classes simples	Hibernate 3
WEB framework (1.4) 2	1.4	Classes simples	Classes simples	JDO (Speedo)

3.7 Motor de geração de código

Esta seção aborda os aspectos relacionados ao motor de geração do NovaStuido.

[16] http://tomcat.apache.org/

3.7.1 Arquitetura do motor nativo de geração

Esta seção apresenta a arquitetura do motor nativo de geração[17]. A figura a seguir mostra a visão lógica do sistema.

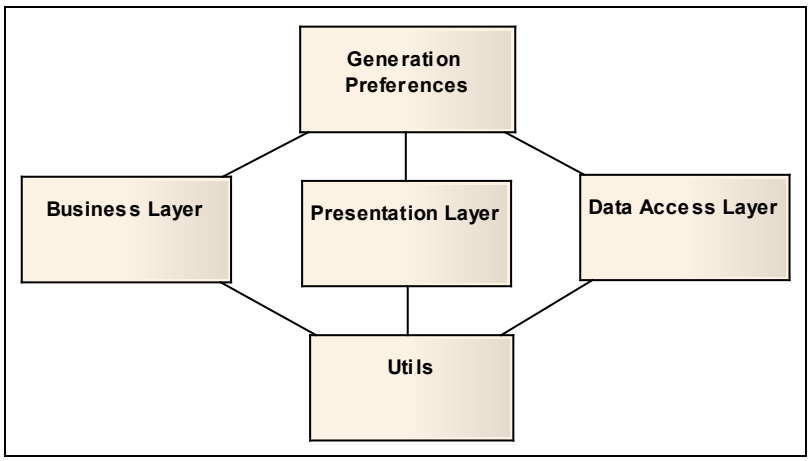

Figura 3.7: Visão lógica do motor nativo de geração.

- **Generation preferences**: este módulo gerencia as preferências utilizadas para gerar o código: tipo de *framework* (*Framework J2EE* ou *Framewok Web*), tipo de tecnologia a ser utilizada (EJB3, EJB2, Hibernate, JDO), etc.
- **Business Layer**: este módulo é responsável pela geração do código relativo à camada *Business Service*[18].
- **Presentation Layer**: este módulo é responsável pela geração do código relativo à camada *Delegate* & *Service Locator*.
- **Data Access Layer**: este módulo é responsável pela geração do código relativo à camada *Business Object* & *Data Acess*.
- **Utils**: este módulo oferece toda a infraestrutura que os outros módulos precisam. Ele se preocupa, por exemplo, com a formatação do código gerado e com o fornecimento das bibliotecas que ele necessita.

3.7.2 Tecnologia atualmente utilizada pelo motor de geração

Como visto na seção 1.1, o motor de geração do NovaStudio é baseado em *templates Velocity*. No entanto, pelo fato de não ter sido criado para atender aos fins específicos da MDA, o *Velocity* apresenta alguns inconvenientes, sendo que o principal deles é que os seus *templates* não levam em consideração o meta-modelo, ficando a cargo do desenvolvedor a realização de manipulações sobre o modelo que não violem o seu meta-modelo. Considere, por exemplo, o *template Velocity* que simplesmente gerará o esqueleto de uma classe Java (Figura 3.8). O resultado da execução deste *template* será uma classe Java cujo nome será obtido pela substituição da variável *$name* por um valor que será definido no interior da classe Java chamadora do *template* (Figura 3.9).

[17] Entende-se por "motor nativo de geração", o motor de geração atualmente utilizado pelo NovaStudio.

[18] Veja seção 3.4.

```
public class $name{

}
```

Figura 3.8: Template Velocity para a geração do esqueleto de uma classe Java.

```
VelocityContext context = new VelocityContext();
        context.put("name", "Foo");
```

Figura 3.9: Código Java para a inicialização das variáveis do *template* Velocity.

Identificam-se claramente dois problemas principais: primeiramente não existe nenhuma garantia de que a variável *$name* estará inicializada no momento em que o *template* for executado. Pode acontecer que o programador erre o nome da variável no momento da inicialização (por exemplo, trocando "name", em inglês, por "nome", em português) ou que ele simplesmente esqueça-se de inicializá-la. Além disso, um segundo problema, e o mais grave no contexto da MDA, é o fato de não existir nenhuma garantia de que a estrutura do *template* está obedecendo as restrições impostas pelo meta-modelo, exigindo do desenvolvedor de *templates* uma programação extremamente criteriosa.

Desta forma, a equipe do NovaStudio realizou um estudo para a identificação de uma nova tecnologia, adaptada ao contexto da MDA, para substituir os *templates Velocity*. Ao final deste estudo identificou-se a solução de código aberto Acceleo como o provável substituto dos *templates* Velocity no motor de geração do NovaStudio, sendo o objetivo deste trabalho a validação da utilização desta tecnologia.

A próxima seção apresenta resumidamente o Acceleo.

3.8 Acceleo

O Acceleo[19] é um gerador de código que utiliza a abordagem MDA para a geração de código em diferentes linguagens (PHP, Java, .NET). Ele é nativamente integrado ao *Eclipse* e ao *Eclipse Modeling Framework* (EMF) e compreende um conjunto de ferramentas e editores permitindo a sua fácil adaptação a todo tipo de projeto ou tecnologia. Ele oferece um conjunto de *templates* de geração de código já prontos, assim como, permite ao usuário desenvolver novos *templates* que atendam às suas necessidades.

As seções a seguir darão uma visão geral do Acceleo.

3.8.1 Conceitos

3.8.1.1 Template Acceleo

Um *template Acceleo* (MUSSET; JULIOT; LACRAMPE, 2009) descreve a estrutura do arquivo a ser gerado. Cada *template* contém um conjunto de *scripts Acceleo*, sendo que no máximo um deles contém o atributo `file = « »` na etiqueta `<%script%>`. Este atributo define o nome do arquivo a ser gerado. Além disso, cada *template* está associado a um meta-modelo que é identificado por seu identificador universal de recursos (URI - *Uniform Resource Identifier*).

[19] http://www.acceleo.org/pages/home/en

A figura a seguir mostra um exemplo simples de um *template Acceleo* baseado no meta-modelo da UML 2.0. Esse *template* gera uma classe Java para cada classe do modelo sobre o qual ele é aplicado. As classes serão geradas no diretório "`src/persitence/dto`" e possuirão a declaração de todos os seus atributos.

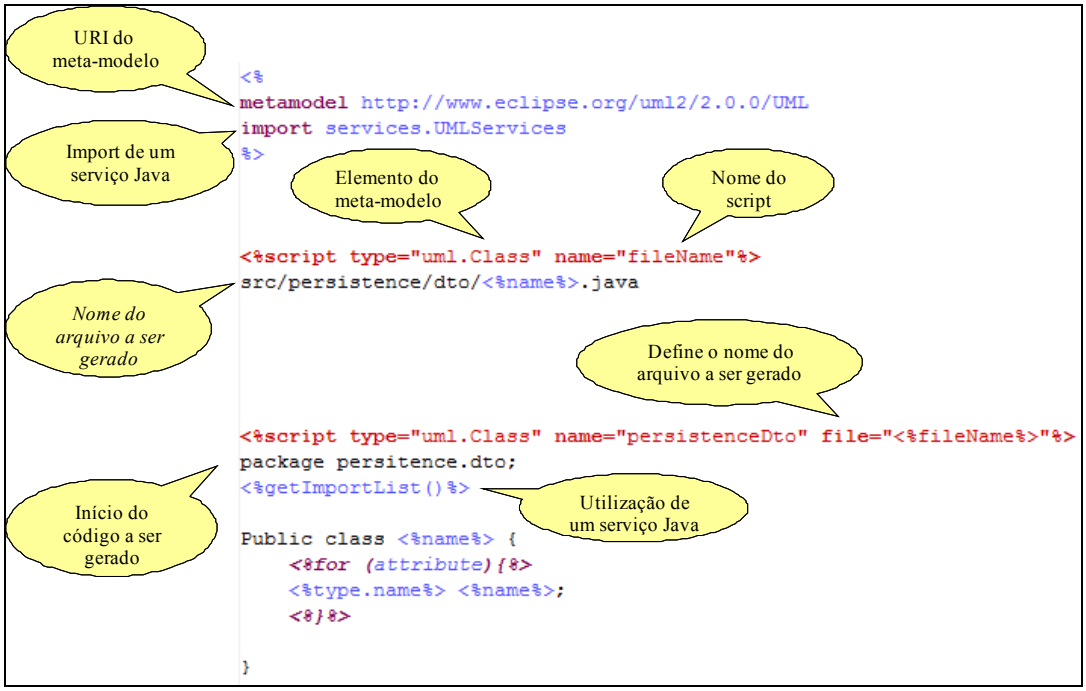

Figura 3.10: Exemplo de *template Acceleo*.

Comparado ao *template Velocity*, o *template Acceleo* possui a vantagem de estar diretamente associado ao meta-modelo. Por exemplo, o nome da classe que no *template Velocity* (Figura 3.8) era dado por uma variável qualquer definida pelo usuário, no *template Acceleo* ele é obtido a partir de um campo definido diretamente no meta-modelo, ou seja, o *template Acceleo* é, por construção, fiel ao meta-modelo. Além disso, não existe a necessidade de uma inicialização do valor deste atributo como no *template Velocity* (Figura 3.9), constituindo um fator de erro a menos. A inicialização será feita automaticamente a partir de modelo de entrada.

Finalmente, como o Acceleo procura estar em conformidade com as normas da OMG[20], o simples fato de utilizar o Acceleo garante que o NovaStudio também estará em conformidade com estas normas.

3.8.1.2 Serviço Acceleo

Um *serviço Acceleo* (MUSSET; JULIOT; LACRAMPE, 2009) é um conjunto de operações escritas em Java, que permitem a realização de operações mais complexas sobre os tipos primitivos e elementos do meta-modelo. Estas operações são importadas e utilizadas nos *templates Acceleo*. Este mecanismo dá uma maior legibilidade ao script e possibilita a realização de operações difíceis de serem escritas com as macros oferecidos pela linguagem de script do Acceleo.

[20] Para maiores detalhes acesse http://www.acceleo.org/pages/le-mda-et-acceleo/en.

3.8.1.3 Código do usuário

Como visto na seção 2.8.3, um dos desafios associados aos geradores de código aplicativo é a integração do código manual ao código gerado. Imagine, por exemplo, que o desenvolvedor já acrescentou o código específico da lógica de negócios e que mudanças no modelo ocasionem um novo lançamento do processo de geração de código. Caso nenhum mecanismo de fusão de código tenha sido previsto o desenvolvedor correrá o risco de perder tudo o que já tinha feito e ser obrigado a recomeçar do zero, ou na melhor das hipóteses, ele terá um trabalhoso processo manual para fusionar os dois códigos. Para resolver este problema o Acceleo fornece duas balizas básicas na sua linguagem de script: uma para indicar o início do código do usuário (<%startUserCode%>) e outra para indicar o final do código do usuário (<%endUserCode%>). Estas balizas serão traduzidas como comentários no código gerado, sendo que tudo que o desenvolvedor escrever entre esses dois comentários será preservado. O Acceleo fará automaticamente a fusão entre o novo código gerado e código existente.

3.8.1.4 Cadeia de lançamento

As cadeias de lançamento (MUSSET; JULIOT; LACRAMPE, 2009) são responsáveis pelo lançamento do processo de geração do código aplicativo. Uma cadeia de lançamento é um arquivo que estabelece a ligação entre os *templates Acceleo* e o modelo sobre o qual eles serão aplicados. Em outras palavras, as cadeias de lançamento associam um *modelo* a cada *template Acceleo*. Além disso, as cadeias de lançamento também definem o diretório no qual o código será gerado.

Vale a pena ressaltar que o lançamento é feito de forma manual através de um menu fornecido para este fim.

3.8.1.5 Meta-modelos clássicos e meta-modelos específicos

O Acceleo fornece um conjunto de URIs de meta-modelos clássicos, como por exemplo, o meta-modelo da UML 2.0 (http://www.eclipse.org/uml2/2.0.0/UML) e o do Ecore (http://www.eclipse.org/emf/2002/Ecore). No entanto, em alguns casos, estes meta-modelos não atendem a todas as necessidades do domínio em questão. A solução adotada pelo Acceleo para contornar este problema, foi fornecer ao usuário a possibilidade de definir o seu próprio meta-modelo e em seguida utilizar a URI deste meta-modelo específico nos *templates* de geração de código.

3.8.2 Processo de geração de código

Em resumo, o processo geração de código usando Acceleo apóia-se sobre os conceitos de *templates* e *serviços* vistos na seção precedente. Basicamente, o Acceleo aplica um conjunto de *templates* (que utilizam *serviços*) sobre um modelo conforme ao meta-modelo utilizado para a criação do *template* e desse processo obtém-se arquivos de código fonte ou arquivos de configuração (Figura 3.11).

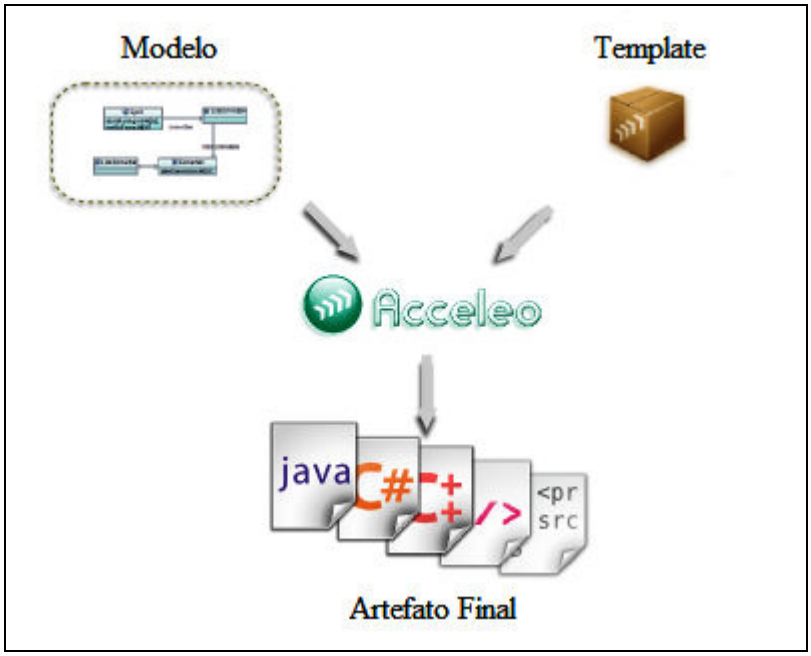

Figura 3.11: Processo de geração de código usando Acceleo

3.9 Considerações finais

O NovaStudio é uma ferramenta do grupo Bull S.A.S baseada no ambiente de desenvolvimento Eclipse, que utiliza a arquitetura MDA para a geração de código J2EE. O código gerado é baseado na divisão em camadas provenientes da abordagem arquitetural modelo-visão-controle e suporta diferentes tecnologias, tais como EJB3, Hibernate3 e JDO.

A tecnologia utilizada no atual motor de geração do NovaStudio apresenta algumas limitações pelo fato de não ser adaptada ao contexto da MDA. Desta forma, um estudo realizado pela equipe do NovaStudio identificou a solução de código aberto Acceleo, que mostrou-se mais adaptada ao contexto da MDA, para substituir os *templates Velocity* no motor de geração.

O próximo capítulo apresenta algumas modificações a serem realizadas na arquitetura do motor de geração do NovaStudio para a integração do novo motor de geração baseado na solução de código aberto Acceleo.

4 MODELAGEM DO NOVO MOTOR DE GERAÇÃO

Este capítulo propõe as alterações que devem ser efetuadas na arquitetura lógica do motor de geração do NovaStudio para permitir a utilização do Acceleo como novo motor de geração.

4.1 Modificações da arquitetura para a utilização do Acceleo

Esta seção mostra as mudanças a serem feitas na arquitetura do motor de geração de código do NovaStudio para a inclusão dos módulos de geração Acceleo tanto para a geração de código J2EE quanto para a geração de código PHP e .NET.

4.1.1 Arquitetura do motor J2EE

Para a validação inicial do Acceleo como novo motor de geração optou-se pelo desenvolvimento de *templates* Acceleo para a geração de código J2EE, pois o processo de geração de código para esta tecnologia já é bem conhecido. Caso consiga-se obter o mesmo código gerado atualmente pelo NovaStudio sem perdas de funcionalidades, isso significa que o Acceleo pode ser utilizado como novo motor de geração. Vale ressaltar que para termos de validação não será utilizada a mesma cobertura de geração de código fornecida pelo motor nativo, mas será gerado apenas o código relativo à camada de negócios (*bussiness layers*) para o *framework J2EE* com a tecnologia EJB3. A camada de apresentação (*presentation layers*) e as outras combinações de *framework* e tecnologias continuarão sendo geradas apenas pelo motor nativo do NovaStudio.

Estima-se que os resultados obtidos com a geração do código relativo à camada de negócios serão suficientes para a validação ou não da utilização do Acceleo. O objetivo é que a longo prazo todo o código gerado com o Velocity passe a ser gerado com o Acceleo. No entanto, a curto prazo, a geração de código PHP e .NET é mais urgente. Assim, primeiramente será gerada apenas uma parte do código Java para termos de validação da nova tecnologia e, uma vez validada, passa-se à geração de código PHP e .NET. A tabela a seguir ilustra a cobertura inicial da geração de código Acceleo.

Tabela 4.1: Cobertura inicial do motor de geração Acceleo.

		Framework J2EE	*Framework Web*
Business Layers	*Business Object et Data Acess*	Acceleo (EJB3)	Motor Nativo
	Business Service	Acceleo (EJB3)	Motor Nativo
	Session Facade	Acceleo (EJB3)	Motor Nativo
Presentation Layers	*Delegate et Service Locator*	Motor Nativo	Motor Nativo
	Controller	Motor Nativo	Motor Nativo
	IHM	Motor Nativo	Motor Nativo

Desta maneira, a Figura 4.1 mostra as alterações feitas na arquitetura do motor nativo de geração para a inclusão dos módulos de geração Acceleo:

- **Generation Preferences**: este módulo quase não sofre modificações, a única diferença é que agora ele também permitirá ao usuário a escolha do motor de geração: nativo ou Acceleo.
- **Acceleo Business Layer e Acceleo Data Access Layer**: estes dois novos módulos substituirão, respectivamente, os módulos *Business Layer* e *Data Access Layer* quando o motor Acceleo de geração for escolhido.

Como num primeiro momento a camada de apresentação continuará sendo gerado pelo motor nativo do NovaStudio, o módulo *Presentation Layer* permanece inalterado. O mesmo acontece com o módulo *Utils*, pois suas funcionalidades não dependem do motor de geração escolhido.

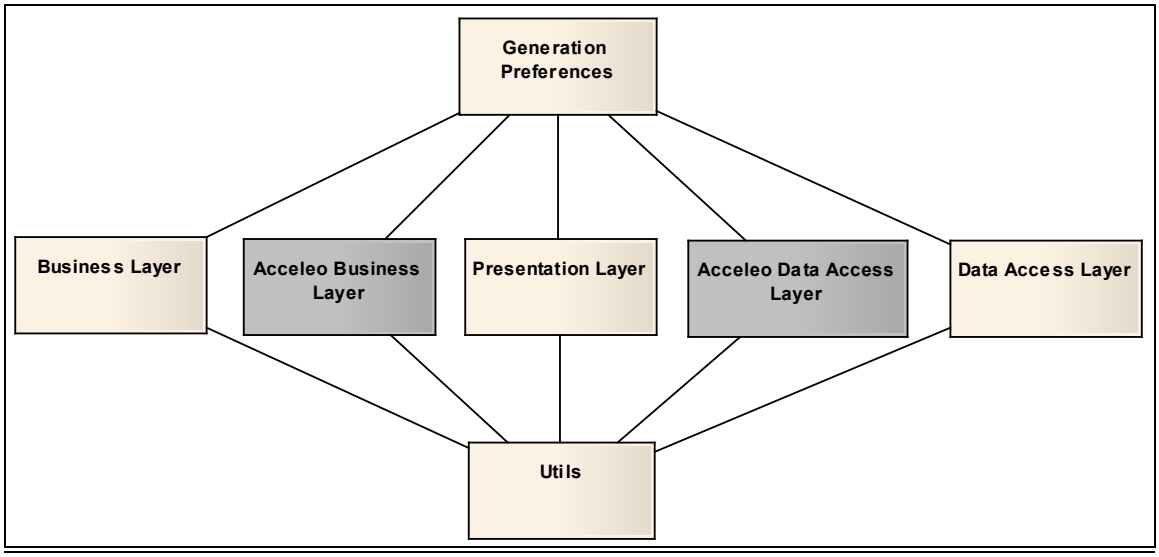

Figura 4.1: Visão lógica do novo motor de geração.

O diagrama de seqüência a seguir ilustra o funcionamento do sistema quando o motor Acceleo é escolhido.

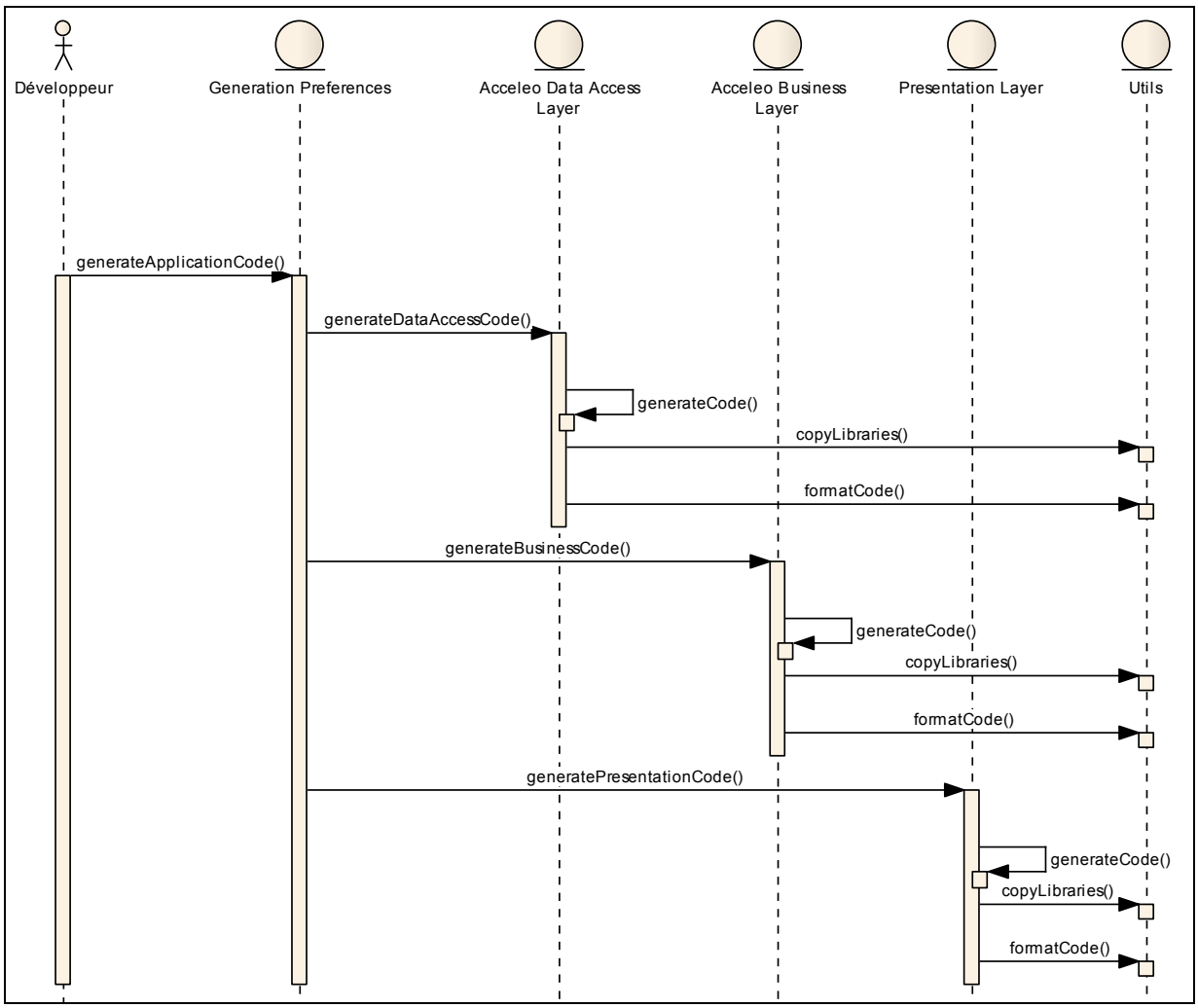

Figura 4.2: Diagrama de seqüência do novo Motor de geração.

4.1.2 Arquitetura do motor PHP

O objetivo é manter para o código PHP a mesma arquitetura lógica utilizada para o código J2EE (seção 3.4). Assim, os novos módulos necessários para esta extensão seriam (Figura 4.3): *PHP Business Layer*, *PHP Presentation Layer*, *PHP Data Access Layer* e *PHP Utils*, substituindo respectivamente, *Business Layer* (ou *Acceleo Business Layer*), *Presentation Layer*, *Data Access Layer* (ou *Acceleo Data Acess Layer*) et *Utils* quando a geração de código PHP é escolhida. O módulo *Generation Preferences* é comum às duas arquiteturas (J2EE e PHP) e agora, permite a escolha entre as duas tecnologias (Figura 4.4).

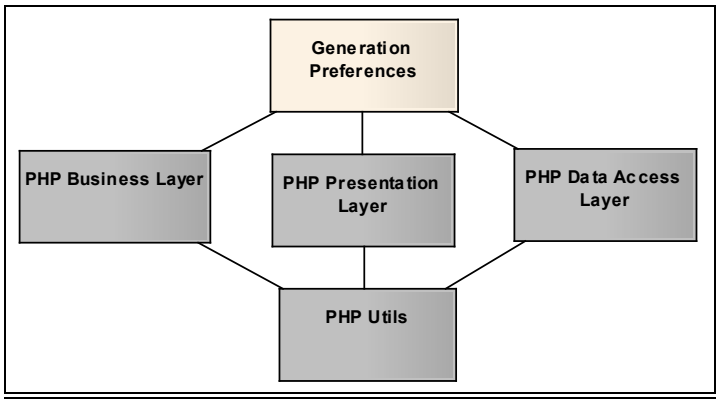

Figura 4.3: Arquitetura do motor PHP.

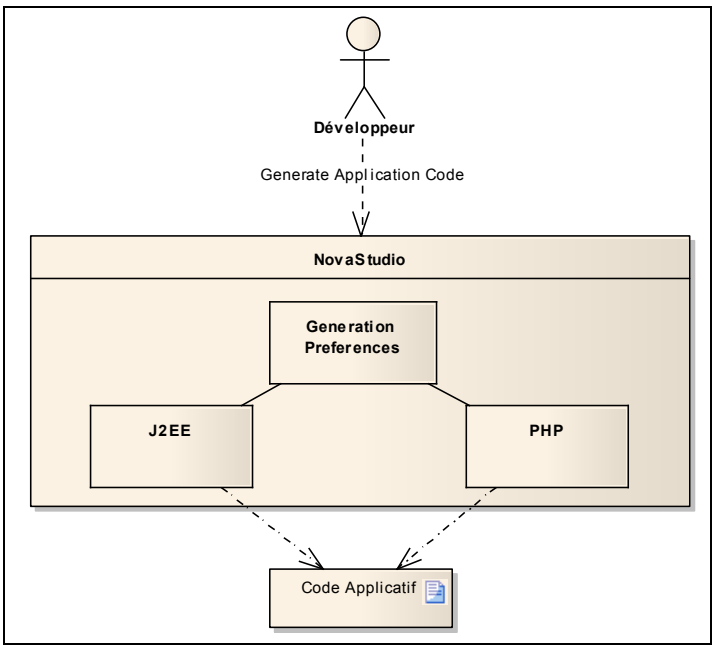

Figura 4.4: Visão global do motor de geração – J2EE e PHP.

Obviamente, primeiramente será necessário um estudo mais aprofundado da tecnologia para verificar, entre outros, a viabilidade de manter a mesma arquitetura lógica. Além disso, para cada módulo acrescentado será necessário desenvolver uma série de *templates* dedicados à geração de código PHP.

4.1.3 Arquitetura do motor .NET

Finalmente, seguindo o mesmo princípio do motor PHP, o motor .NET contará com os seguintes módulos (Figura 4.5): *.NET Business Layer*, *.NET Presentation Layer*, *.NET Data Access Layer* e *.NET Utils*. Note que o módulo *Generation Preferences* permitirá agora a escolha entre as três tecnologias (J2EE, PHP e .NET) (Figura 4.6).

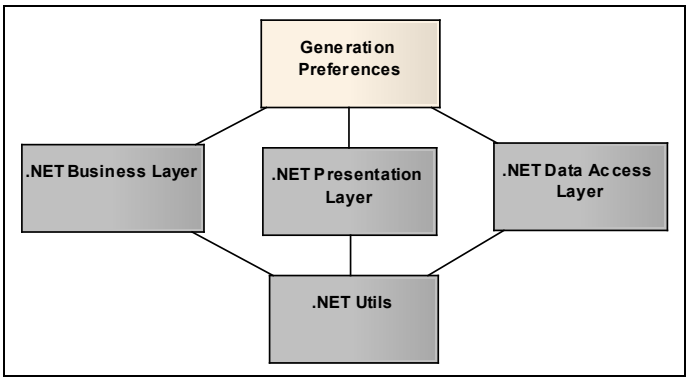

Figura 4.5: Arquitetura do motor .NET.

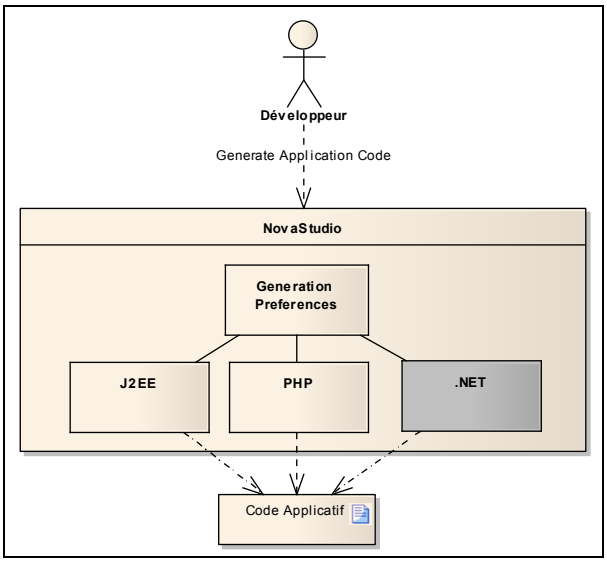

Figura 4.6: Visão global do motor de geração – J2EE, PHP e .NET.

Novamente ressalta-se a importância de um estudo mais aprofundado sobre a tecnologia para verificar a viabilidade de manter a mesma arquitetura lógica do código gerado e fato da necessidade do desenvolvimento de *templates* dedicados à geração de código .NET.

4.2 Considerações finais

Este capítulo apresentou as mudanças que devem ser feitas na arquitetura lógica do motor de geração do NovaStudio para permitir a utilização da solução de código aberto Acceleo. Foram propostas as arquiteturas lógicas dos motores de geração para cada uma das três tecnologias (J2EE, PHP e .NET), sendo que primeiramente será colocada em prática a implementação do novo motor de geração sobre a tecnologia J2EE. Isso será abordado no próximo capítulo.

5 IMPLEMENTAÇÃO DO NOVO MOTOR DE GERAÇÃO

Este capítulo apresenta os detalhes da implementação do novo motor de geração de código, dando uma visão geral do processo de geração de código utilizado pelo Acceleo.

5.1 Escolha do meta-modelo

Como visto na seção 3.2.1, o modelo criado na ferramenta de modelagem é primeiramente exportado no formato XMI para, para em seguida, ser transformado em um novo arquivo seguindo o meta-modelo específico do NovaStudio. Neste contexto, temos duas possibilidades para a integração do novo motor de geração com o NovaStudio:

- Utilizar o arquivo XMI exportado diretamente pela ferramenta de modelagem como arquivo (modelo) de entrada do Acceleo e, neste caso, utilizar o meta-modelo clássico da UML.
- Utilizar o arquivo gerado pelo NovaStudio a partir do arquivo XMI original e, neste caso, utilizar o meta-modelo específico do NovaStudio.

Estas duas opções serão detalhadas nas subseções seguintes.

5.1.1 Meta-modelo da UML

O Acceleo é nativamente integrado ao Eclipse e ao EMF (*Eclipse Modeling Framework*). Isso faz com que modelos criados em ferramentas de modelagem integradas ao Eclipse e que fazem uso do EMF possam ser usados diretamente como modelo de entrada para o Acceleo. Entretanto, no caso da utilização de ferramentas de modelagem externas, é necessário converter o arquivo XMI exportado em um novo arquivo baseado no EMF, utilizando-se para isto uma funcionalidade fornecida pelo Acceleo (MUSSET; JULIOT; LACRAMPE, 2009). Entretanto, isso impõe algumas restrições:

- Na sua versão 2.5 o Acceleo permite apenas a conversão de arquivos XMI baseados em UML 1.4, não suportando UML 2.0.
- A conversão falha se for utilizada a opção "*Export diagrams*" do Enterprise Architect (versão 7.0) ou se o modelo contiver diagramas de atividades. Assim, podemos exportar apenas as classes do modelo.

A tabela abaixo estabelece uma comparação entre as funcionalidades fornecidas pelo NovaStudio se for utilizado o motor nativo de geração e se forem utilizados *templates Acceleo* baseados no meta-modelo UML.

Tabela 5.1: Motor nativo VS meta-modelo UML

	Acceleo (meta-modelo UML)	*Motor nativo*
Versão UML/XMI	UML1.4/XMI1.2	UML2.x/XMI 1.x
Suporte exportação de diagramas	Não	Sim
Suporte exportação de modelos contendo diagramas de atividade	Não	Sim
Suporte a propriedades	Sim	Sim
Suporte à estereótipos	Não verificado[21]	Sim
Associações	Não verificado	Sim
Modelo representado por	Arquivo "*.uml14"	Arquivo "*.generation"

5.1.2 Meta-modelo específico

A segunda abordagem para estabelecer a integração entre NovaStudio e Acceleo, consiste em utilizar o meta-modelo específico do NovaStudio[22] como meta-modelo nos *templates* de geração. Desta forma o arquivo gerado pelo NovaStudio a partir do arquivo XMI original poderá ser usado como o modelo sobre o qual os *templates* serão aplicados.

A tabela abaixo estabelece uma comparação entre as funcionalidades fornecidas pelo NovaStudio se for utilizado o motor nativo de geração e se forem utilizados *templates Acceleo* baseados no meta-modelo específico do NovaStudio.

Tabela 5.2: Motor nativo VS meta-modelo específico

	Acceleo (Meta-modelo específico)	*Motor Nativo*
Versão UML/XMI	UML2.x/XMI 1.x	UML2.x/XMI 1.x
Suporte exportação de diagramas	Sim	Sim
Suporte exportação de modelos contendo diagramas de atividade	Sim	Sim
Suporte a propriedades	Sim	Sim
Suporte à estereóitpos	Sim	Sim
Associações	Sim	Sim
Modelo	Arquivo "*.generation"	Arquivo "*.generation"

[21] Como esta primeira abordagem já apresentava limitações notáveis, passou-se diretamente para a análise da segunda abordagem sem a verificação do suporte aos estereótipos ou às associações.

[22] O meta-modelo específico do NovaStudio é compatível com meta-modelo UML, porém são acrescentadas algumas restrições que devem ser obedecidas por um diagrama UML para que o mesmo possa ser utilizada como entrada para o NovaStudio. Além disso, não se trata de um novo meta-modelo desenvolvido especificamente para atender as necessidades do Acceleo, mas sim do antigo meta-modelo que já era utilizado pelo motor nativo de geração.

5.1.3 Considerações e tomada de decisão

Com base nas duas subseções anteriores pode-se constatar que a utilização do meta-modelo UML impõe limitações significativas com relação ao motor nativo de geração: limita-se à utilização da UML 1.4, enquanto o NovaStudio utiliza UML 2.0. Além disso, não é possível exportar modelos que contenham diagramas de atividade. Isso gera um grande inconveniente, pois o NovaStudio utiliza diagramas de atividade para a geração das interfaces gráficas com o usuário.

Por outro lado, a utilização do meta-modelo específico do NovaStudio, resolve todos estes problemas e obtemos exatamente as mesmas funcionalidades oferecidas pelo motor nativo. Além disso, esta segunda abordagem possibilitará uma integração mais fácil do novo motor de geração ao NovaStudio, visto que o modelo será representado pelo mesmo arquivo utilizado pelo motor nativo (arquivo "*.generation").

Finalmente, convém ressaltar que a escolha do meta-modelo a ser utilizado nos *templates* de geração de código serão totalmente transparentes ao usuário final do NovaStudio, exigindo apenas que desenvolvedor dos *templates* tenha conhecimento do meta-modelo.

5.2 Integração do novo motor de geração ao NovaStudio

Como visto na seção 3.8.1.4, o lançamento da geração de código com o Acceleo é feito manualmente através de *cadeias de lançamento*. Entretanto, para uma perfeita integração ao NovaStudio é necessário lançar o novo motor de geração (motor Acceleo) de maneira automatizada através de chamadas de métodos. Isso é possível graças a API fornecida pelo Acceleo através dos mecanismos extensão fornecidos pelos *plug-ins* Eclipse. Desta maneira, basta desenvolver um novo *plug-in* Eclipse com dependência ao *plug-in* Acceleo, para ter acesso a esta API.

5.3 Artefatos gerados

Esta seção apresenta o conjunto de artefatos que será gerado a partir de um diagrama de classes fornecido no modelo de entrada. Para as subseções que seguem considere que os diagramas de classes apresentados pertencem ao domínio funcional *família* (*family*).

5.3.1 Entidades

O diagrama de classes a seguir (Figura 5.1) mostra uma classe abstrata, *Pessoa* (*Person*), que é especializada em duas classes concretas, *Homem* (*Man*) e *Mulher* (*Woman*). Estas classes representam as entidades que compõem o domínio funcional *família*, ou seja, para o NovaStudio trata-se de classes com o estereótipo *Entity*. Além disso, as associações entre as classes indicam que um homem pode exercer o papel de marido (*husband*) para uma mulher que por sua vez exercerá o papel de mulher (*wife*) para o homem. Analogamente, o homem e a mulher exercem, respectivamente, o papel de pai (*father*) e mãe (*mother*) para seus filhos (*children*). Finalmente, a área delimitada por *tags* indica as propriedades (*tagged values*) que são utilizados na classe em questão[23]. A classe *Man*, por exemplo, contém as propriedades "Class Business", "Class DiscriminatorValue" e "Class Entity", sendo que "Class Entity = True" indica que se trata de uma entidade. Este diagrama de classes será

[23] Obs: na figura estão visíveis apenas as propriedades associados à classe, existem ainda outras propriedades que estão associados aos atributos e que não são visíveis diretamente no diagrama de classes.

utilizado para ilustrar o processo de geração de código para as classes que representam uma entidade.

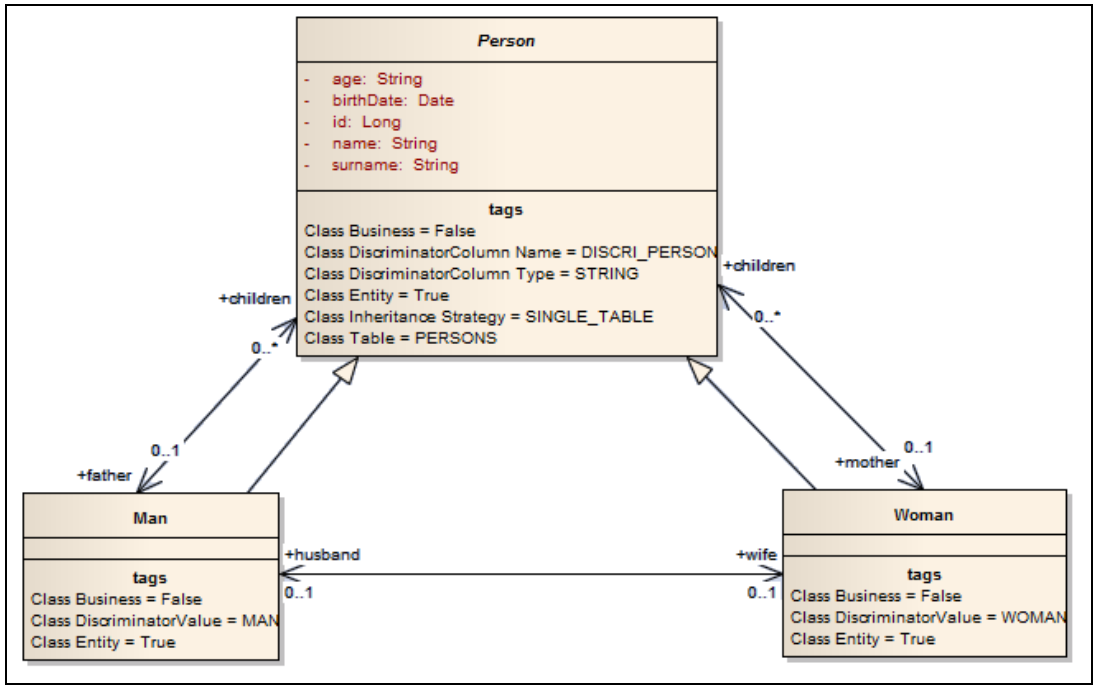

Figura 5.1: Classes de entidades (estereótipo “Entity”).

5.3.1.1 Interfaces e classes Java

Para cada classe do tipo entidade no modelo de entrada, será gerada uma interface Java com o mesmo nome da classe no diagrama. Será gerada igualmente a classe Java que implementa esta interface, cujo nome é obtido pelo acréscimo do sufixado “*Bean*” ao nome da interface. Por exemplo, para a classe “*Person*” do diagrama de classes serão geradas a interface “*Person*” (Figura 5.2) e a classe “*PersonBean*” (Figura 5.3).

```
/**
 * Person Business Object notes from design.
   @version 1.0
 */
public interface Person extends Serializable {
```

Figura 5.2: Código gerado – Declaração da interface *Person*.

```
/**
 * Person Business Object notes from design.
   @version 1.0
 */
@Entity
@Table(name = "PERSONS")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DISCRI_PERSON", discriminatorType =
DiscriminatorType.STRING)
@SequenceGenerator(name = "person_id", sequenceName = "PersonSequence",
allocationSize = 1)
@SuppressWarnings("unchecked")
public abstract class PersonBean extends java.lang.Object
    implements Serializable, Person {
```

Figura 5.3: Código gerado – Declaração da classe *Person*.

5.3.1.2 Atributos diretos

A classe Java gerada conterá os atributos especificados no modelo juntamente com as respectivas anotações EJB3 extraídas a partir de propriedades (*tagged values*). Além disso, também serão gerados os respectivos *getters* et *setters*. A figura a seguir mostra os atributos gerados para a classe *Person*.

```
    /**
     *age class attribute
     */
    @Column(name = "AGE", unique = false, nullable = true,
                    insertable = true, updatable = true)
    private String age;

    /**
     * FEATURE 307088 Temporal annotation on Date field
     */
    @Temporal(TemporalType.DATE)
    @Column(name = "BIRTHDATE", unique = false, nullable = true,
                    insertable = true, updatable = true)
    private Date birthDate;

    /**
     *id class attribute
     */
    @Id
    @Column(name = "PK_PERSON")
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
            generator = "person_id")
    private Long id;

    /**
     *name class attribute
     */
    @Column(name = "NAME", unique = false, nullable = true,
                    insertable = true, updatable = true)
    private String name;

    /**
     *surname class attribute
     */
    @Column(name = "SURNAME", unique = false, nullable = true,
                    insertable = true, updatable = true)
    private String surname;
```

Figura 5.4: Código gerado – Atributos da classe *Person*.

5.3.1.3 Atributos derivados de associações

Além dos atributos diretos, para cada associação serão gerados dois novos atributos (com respectivos *getters* e *setters*), um correspondendo à associação propriamente dita e outro, com valor booleano, para indicar a estratégia que deve ser utilizada para o preenchimento deste atributo. O nome do atributo correspondente à associação é dado pelo nome do papel (*role*) desempenhado pela classe destino. Este atributo será um atributo simples caso o associação tenha multiplicidade 1 ou uma coleção caso a associação tenha multiplicidade n. Por exemplo, a classe *Man* conterá o atributo *wife* (atributo simples) proveniente da associação com a classe *Woman* e o atributo *children* (coleção) proveniente da associação com a classe *Person*. Analogamente a classe *Woman* conterá o atributo *husband* (atributo simples) proveniente da associação com a classe *Man* e o atributo *children* (coleção) proveniente da associação com a classe *Person*. Já o nome do atributo que define a estratégia de preenchimento é obtido pelo acréscimo do prefixo *forceLoad* ao nome do atributo correspondente, obtendo-se por exemplo, os atributos *forceLoadWife*, *forceLoadChildren* e *forceLoadHusband*. Caso este atributo seja verdadeiro indica que será utilizada a estratégia ansiosa de preenchimento (*Eager loading*), ou seja, o atributo referente à associação será

preenchido automaticamente no momento da instanciação do objeto. Já quando o atributo *forceLoad* é falso indica que será utilizada a estratégia preguiçosa de preenchimento (*Lazy loading*), ou seja, o atributo correspondente à associação apenas será preenchido no momento em que a aplicação o referenciar. Caso não haja nenhuma referência a este atributo ele nunca será preenchido.

A figura a seguir mostra os atributos da classe *Man* derivados das associações desta classe.

```
@OneToOne(cascade =  {
    CascadeType.PERSIST, CascadeType.MERGE}
, fetch = FetchType.EAGER)
@JoinColumn(name = "FK_WOMAN", insertable = true, updatable = true)
private WomanBean wife = null;

/**
 * When <code>forceLoadWife</code>
 * is set to <code>true</code> the
 * <code>Wife</code>
 * Collection will be populated automatically when the
 * ManBean class is retrieved.
 */
@Transient
private boolean forceLoadWife;
@OneToMany(mappedBy = "father", cascade =  {
    CascadeType.ALL}
, fetch = FetchType.LAZY)
private Collection<PersonBean> children;

/**
 * When <code>forceLoadChildren</code>
 * is set to <code>true</code> the
 * <code>Children</code>
 * Collection will be populated automatically when the
 * ManBean class is retrieved.
 */
@Transient
private boolean forceLoadChildren;
```

Figura 5.5: Código gerado – Atributos da classe *Man* derivados de associações.

5.3.2 Serviços

Considere agora as classes do modelo que são responsáveis por definir a lógica de negócios da aplicação. Estas classes conterão os métodos que manipularão os objetos de negócios (entidades) apresentados anteriormente e são representadas no NovaStudio com o estereótipo "*Business*". A figura abaixo apresenta as classes de serviços de negócios para o domínio de negócio *Família* e será utilizada para ilustrar o código que é gerado a partir das classes de serviços do modelo.

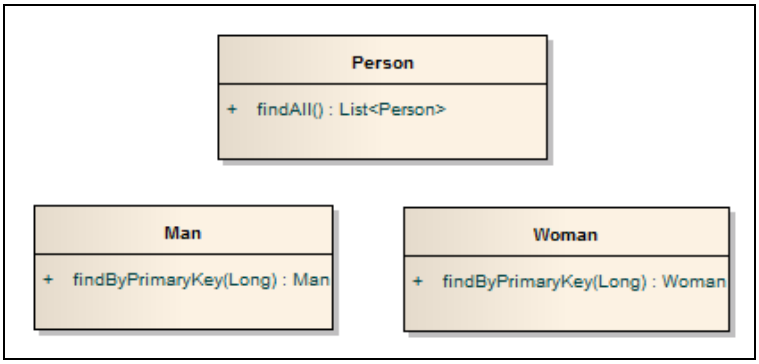

Figura 5.6: Classes de serviços de negócios (estereótipo “Business”).

5.3.2.1 Interfaces e classes Java

Para cada classe de serviço do modelo será gerada uma interface Java, cujo nome é obtido pelo nome da classe no modelo sufixado por “*BusinessService*”, e uma classe Java (com anotações EJB3) cujo nome é obtido pelo nome da classe no modelo do sufixado por “*Business*”. Desta forma serão geradas, por exemplo, a interface *PersonBusinessService* (Figura 5.7) e a classe *PersonBusiness* (Figura 5.8).

```
/**
 * Person Service notes from design.
   @version 1.0
 */
public interface PersonBusinessService {
```

Figura 5.7: Código gerado – Declaração da interface *PersonBusinessService*.

```
/**
 * Person Service notes from design.
   @version 1.0
 */
@Stateless
@Local(PersonBusinessService.class)
public class PersonBusiness implements PersonBusinessService {
```

Figura 5.8: Código gerado – Declaração da classe *PersonBusiness*.

5.3.2.2 Métodos CRUD

Através de uma propriedade (*tagged value*) específica, o usuário pode indicar ao NovaStudio que os métodos CRUD (Create, Read, Update, Delete) devem ser gerados automaticamente. Estes métodos são responsáveis, respectivamente, pela criação, leitura, atualização e remoção de entidades da base de dados. A figura abaixo ilustra os métodos CRUD gerados para a classe *Person*.

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person findByPK(Person person) {
        person = (Person) entityManager.find(PersonBean.class,
                                             person.getPersonPK());
        if (person == null) {
            // record not found
            return null;
        }

        //Init associations according to lazy loading
        //forceLoad<Association> attributes
        if (person.getForceLoadMother()) {
            person.getMother();
        }
        if (person.getForceLoadFather()) {
            person.getFather();
        }
        return person;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person create(Person person) {
        person = entityManager.merge(person);
        entityManager.persist(person);
        return person;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person update(Person person) {
        person = entityManager.merge(person);
        entityManager.persist(person);
        return person;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person remove(Person person) {
        person = entityManager.merge(person);
        entityManager.remove(person);
        return person;
    }
```

Figura 5.9: Código gerado – Métodos CRUD da classe *PersonBusiness*.

5.3.2.3 Métodos do usuário

O NovaStudio gerará o esqueleto de cada método que o usuário definiu no diagrama de classes, sendo que o acréscimo do código específico da lógica de negócios fica a cargo do desenvolvedor. A figura a seguir, mostra o código gerado para o método *findAll* da classe *Person*. Note que ao contrário das classes de entidade (*Entity*) onde todo o código necessário é gerado automaticamente pelo NovaStudio, aqui o desenvolvedor deve acrescentar o código relativo à lógica de negócios. Desta forma, é necessário garantir que este código manual não será apagado caso uma segunda geração faça-se necessária. Como visto na seção 3.8.1.3 o Acceleo fornece balizas de início e fim de código do usuário para resolver este problema, como é ilustrado na Figura 5.10.

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public List<Person> findAll() throws java.lang.Exception {
    //Start of user code not default 4 @warning: do not remove this comment
    //TODO : must be implemented
    return null;

    //End of user code
}
```

Figura 5.10: Código gerado – Métodos do usuário

5.3.3 Domínios funcionais

Para cada domínio funcional será gerado uma única *session façade* (representada por uma interface e uma classe Java) que reagrupa todos os serviços oferecidos pelas classes que pertencem a este domínio funcional. O nome da interface Java é obtido pelo nome do domínio funcional sufixado por "*Service*", enquanto o nome da classe Java é obtido pelo nome do domínio funcional sufixado por "Facade". Desta forma, para o domínio funcional *family* serão geradas a interface *FamilyService* e a classe *FamilyFacade* reagrupando todos os serviços oferecidos pelas classes *PersonBusiness*, *ManBusiness* e *WomanBusiness*. Os nomes dos métodos são obtidos pelo acréscimo do nome da classe a que pertencem. Por exemplo, os métodos *findAll* da classe *PersonBusiness* e *create* da classe *Man* darão origem, respectivamente, aos métodos *findAllPerson* e *createMan* da classe *FamilyFacade* (figuras abaixo).

```
@Stateless
@Remote(FamilyService.class)
@Local(FamilyService.class)
/**
 * Applications use this Session Facade to execute the business methods
 * of the family domain.
 */
public class FamilyFacade implements FamilyService {
```

Figura 5.11: Código gerado – Declaração da classe *FamilyFacade*.

```
    @EJB
    private PersonBusinessService personBusinessService;
    @EJB
    private WomanBusinessService womanBusinessService;
    @EJB
    private ManBusinessService manBusinessService;
```

Figura 5.12: Código gerado – Referência para cada domínio funcional.

```
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public List<Person> findAllPerson() throws java.lang.Exception {
        return (List) BeanUtilsConverter.convertCollectionProperty(null,
                      (List) personBusinessService.findAll(), null);
    }
```

Figura 5.13: Código gerado – Exemplo de método da *session Façade*.

5.4 Análise dos resultados

Com bases nas regras apresentadas na seção anterior foram criados *templates* Acceleo para a geração do código Java das subcamadas que compõem a camada *Business Layers*[24]. Para testar estes *templates* utilizamos o mesmo cenário de testes que já era utilizado para testar o motor nativo do NovaStudio. Este cenário, que é parcialmente apresentado na sessão anterior (Figura 5.1 e Figura 5.6), contém um conjunto mínimo de características a serem testadas: associações, heranças, estereótipos e propriedades. Além disso, após ter sido validado no cenário de testes, o novo motor de geração será utilizado em projetos reais do setor de serviços da Bull o que permitirá eventuais melhoras do motor.

O código gerado pelos *templates* Acceleo foi comparado àquele gerado pelo motor nativo do NovaStudio. Mais precisamente, verificou-se de maneira semi-automatizada através do comparador de códigos do Eclipse, que todos os artefatos (classes, interfaces e arquivos de configuração) gerados pelo motor nativo também são gerados pelo motor Acceleo e que o conteúdo destes artefatos é o mesmo, ou seja, contendo os mesmos métodos e atributos. Como resultado obtiveram-se apenas pequenas diferenças referentes a comentários ou formatação de código. Por exemplo, os comentários que indicam o início e o fim do código do usuário estão presentes no código gerado pelo Acceleo, mas não estão presentes no código gerado pelo motor nativo. Considerando o exemplo do método *findAll* da classe serviços *Person*, o motor nativo gerará o código da Figura 5.14, enquanto o Acceleo gerará o código da Figura 5.15. O resultado fornecido pelo comparador do Eclipse (Figura 5.16) ilustra justamente a diferença causada pelos comentários de início e fim do código de usuário. Além de serem de fundamental importância para a fusão do código do usuário com o código gerado, estes comentários servirão como indicativo para o programador saber onde o código deve ou não ser alterado.

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public List<Person> findAll() throws java.lang.Exception {
    return null;
}
```

Figura 5.14: Código gerado pelo motor nativo.

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public List<Person> findAll() throws java.lang.Exception {
    //Start of user code not default 4 @warning: do not remove this comment
    //TODO : must be implemented
    return null;

    //End of user code
}
```

Figura 5.15: Código gerado pelo Acceleo.

Desta forma, a utilização do novo motor de geração de código para a tecnologia J2EE foi concluída com sucesso, o que nos permite considerar que o acréscimo das funcionalidades de geração de código PHP e .NET é viável. De fato, bastaria desenvolver os novos *templates* de geração para estas tecnologias e integrá-los à arquitetura existente. Obviamente, seria necessário realizar um estudo mais aprofundando do impacto causado pelo acréscimo destas

[24] Os trechos de código apresentados na seção 5.3 foram gerados pelos *templates* Acceleo.

funcionalidades. Um aspecto importante diz respeito, por exemplo, ao conjunto de propriedades (*tagged values*) utilizadas, uma vez que algumas delas podem deixar de fazer sentido para uma dada tecnologia e que outras precisem ser acrescentadas.

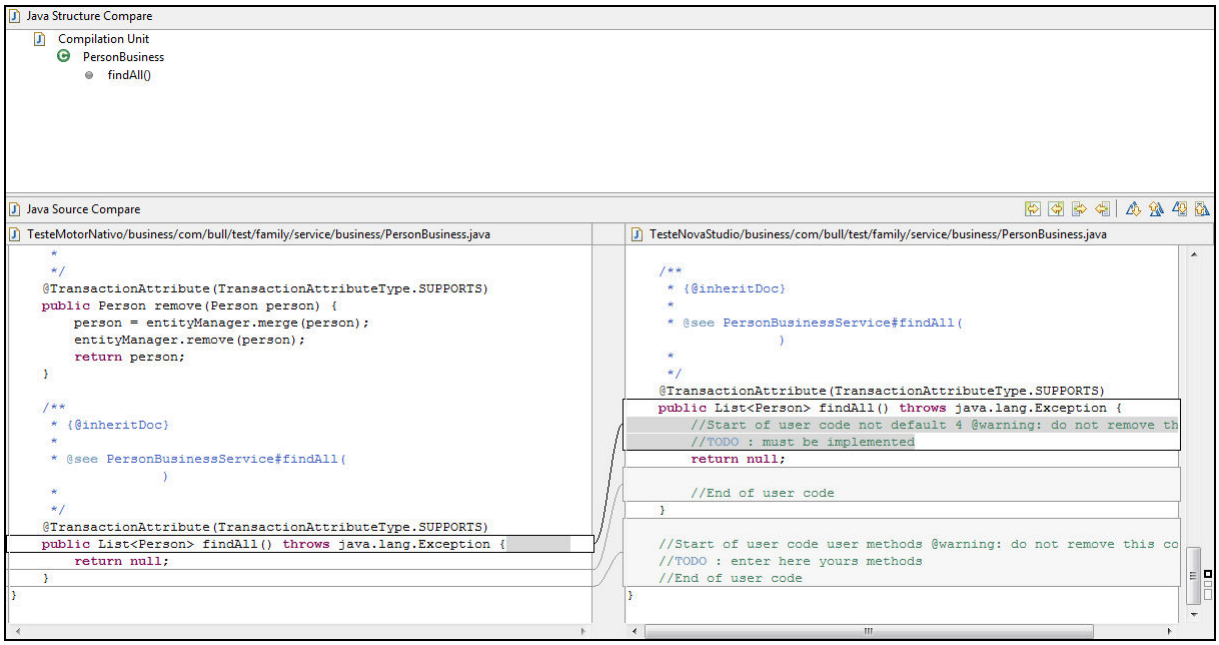

Figura 5.16: Comparador do Eclipse.

6 CONCLUSÃO

Em uma realidade em que os sistemas de informação tornam-se cada vez mais complexos devido à grande quantidade de informação que devem tratar e à sua heterogeneidade, a Engenharia Dirigida pelos Modelos (MDE) apresenta-se como uma possível solução ao problema de complexidade destes sistemas. Neste contexto, o NovaStudio é uma ferramenta desenvolvida pelo grupo Bull S.A.S que utiliza a *Arquitetura Dirigida pelos Modelos* (MDA) para gerar automaticamente uma boa parte do código necessário para aplicações J2EE. Para isto, ele utiliza um motor de geração de código baseado em *templates Velocity*. No entanto, como o *Velocity* não foi concebido com o intuito de atender à arquitetura MDA, ele traz alguns inconvenientes como, por exemplo, o fato de não levar em consideração o meta-modelo utilizado. Desta forma, antes de acrescentar as funcionalidades de geração de código PHP e .NET ao NovaStudio, havia a necessidade de validar a utilização do Acceleo (candidato a substituto do *Velocity*) como novo motor de geração, sendo este o objetivo deste trabalho.

A principal contribuição deste trabalho foi a realização de um estudo aprofundado da solução Acceleo permitindo a sua validação como nova tecnologia a ser utilizada pelo motor geração do NovaStudio. Primeiramente verificou-se que Acceleo realmente contornava os problemas identificados com os *templates Velocity* no contexto da MDA, provendo uma programação de *templates* de geração de código menos propensa a erros, na qual o próprio meta-modelo guia o processo de criação do *template.* Em seguida, no capítulo 4, foram propostas algumas alterações que devem ser feitas na atual arquitetura do motor de geração do NovaStudio para permitir a integração dos *templates Acceleo*. A primeira decisão importante a ser tomada foi a escolha do meta-modelo a ser utilizado pelos *templates* de geração de código, chegando-se à conclusão de que o meta-modelo específico do NovaStudio é o mais adequado. Além disso, vale relembrar que a escolha do meta-modelo implicará conseqüências apenas para o desenvolvedor dos *templates* do motor de geração, sendo totalmente transparente ao usuário final.

Após a escolha do meta-modelo, o próximo passo foi o desenvolvimento de *templates* para a geração de código J2EE para a camada de negócios utilizando-se EJB3. Escolheu-se começar pela geração de código J2EE pelo fato deste processo ser bem conhecido para esta tecnologia, visto que isto já é feito pelo motor nativo. Além disso, estimou-se que a geração de código para a camada de negócios já seria suficiente para fazer uma análise dos resultados e validar ou não a utilização do Acceleo. Uma comparação semi-automatizada do código gerado pelos *templates Acceleo* com o código gerado pelo motor nativo mostrou que se obtinha exatamente o mesmo código com eventuais diferenças relativas a comentários. Desta forma, os resultados obtidos confirmam que o Acceleo é uma boa alternativa para o novo motor de geração de código do NovaStudio. Primeiramente, porque o Acceleo procura seguir as normas e padrões definidos pela OMG, garantindo que o NovaStudio também estará em conformidade com estes. Além disso, os *templates* de geração de código são fáceis de

escrever e bastante flexíveis graças à utilização de meta-modelos específicos e ao uso de serviços Java. Isso permite acrescentar o suporte à geração de código aplicativo em novas linguagens de maneira bastante eficiente. Outra característica importante do Acceleo é o gerenciamento automático da fusão do código do usuário com o código gerado caso uma segunda geração se faça necessária. Esta característica é extremamente importante quando o desenvolvedor já acrescentou parte da lógica de negócios ao código gerado e uma segunda geração é imposta devido às modificações no modelo. Neste caso, o código acrescentado pelo desenvolvedor será automaticamente acrescentado ao novo código gerado. Finalmente, o Acceleo fornece uma API rica que permite uma fácil integração com novos *plug-ins Eclipse* e possui uma comunidade ativa que disponibiliza novas versões do produto regularmente, garantindo um suporte a longo prazo.

Desta forma, este trabalho permitiu concluir que a utilização do Acceleo para a extensão das funcionalidades do NovaStudio é viável. Além disso, ele também mostra a importância da reutilização, quando possível, de soluções prontas. Isso fica evidenciado pelo simples fato de que a utilização do Acceleo para o desenvolvimento de *templates* para a geração de código fornece um conjunto de garantias que seriam necessárias serem reimplementadas caso o motor de geração usasse uma tecnologia não adaptada ao contexto da MDA.

6.1 Trabalhos futuros

A geração de código J2EE inicialmente prevista foi concluída com sucesso, o que permitiu a validação do Acceleo como novo motor de geração de código do NovaStudio. Assim, para completar a cobertura do novo motor de geração de código é necessário desenvolver os *templates* e os módulos para a geração de código PHP e .NET, sendo que uma arquitetura para o motor de geração já é proposta nas seções 4.1.2 e 4.1.3. Além disso, é necessário prever uma nova interface com o usuário possibilitando a escolha entre as diferentes tecnologias.

Convém ressaltar que antes do desenvolvimento dos *templates* para a geração de código PHP e .NET é necessária a realização de um estudo mais aprofundado para cada tecnologia para identificar, por exemplo, a necessidade do acréscimo de novas propriedades (*tagged values*) aos estereótipos utilizados pelo NovaStudio.

Finalmente, pode-se pensar no desenvolvimento de *templates* para a geração de testes unitários.

7 REFERÊNCIAS

AMBLER, S.W. Be Realistic About the UML: It's Simply Not Sufficient. Disponível em: <http://www.agilemodeling.com/essays/realisticUML.htm>. Acesso em: set. 2009.

BEZIVIN, J. In Search of a Basic Principle for Model Driven Engineering. **UPGRADE, The European Journal for the informatics professional**, v.V, n.2, p.21-24, abr. 2004. Disponível em: <http://www.upgrade-cepis.org/issues/2004/2/up5-2Bezivin.pdf>. Acesso em : ago. 2009.

BEZIVIN J.; GERBE O. Towards a Precise Definition of the OMG/MDA Framework. **Ase, p.273, 16th IEEE International Conference on Automated Software Engineering (ASE'01)**, 2001.

BEZIVIN, J. et al. Rapport de synthèse de l'AS CNRS sur le MDA (Model Driven Architecture). **CNRS**, nov. 2004.

BEZIVIN J. et al. Bridging the MS/DSL Tools and the eclipse EMF Framework. **OOPSLA Workshop on Software Factories**, 2005. Disponível em <http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Bezivin.pdf >. Acesso em: set. 2009.

DALGARNO, M.; FOWLER, M. UML vs. Domain-Specific Languages. **Methods and Tools, Practical knowledge for the software developer, tester and project manager, summer 2008**. V.16, n.2, p. 2-8, 2008.

DEMIR, A. Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development. Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third international Workshop on Model-Based Methodologies For Pervasive and Embedded Software (MBD-MOMPES'06). **IEEE Computer Society**. Washington, DC: p. 75-83, 2006.

FAVRE, J.M.; ESTUBLIER, J.; BLAY-FORNARINO M. **L'ingénierie dirigée par les modèles – au-delà du MDA**. 1ª Ed. , [S.l.]: Editora Hermès – Lavoisier, 2006.

FEIKAS, M. How to represent Models, Languages and Transformations?. **Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)**. P. 169-176, 2006.

FRANCE, R.; RUMPE, B. Model-driven Development of Complex Software: A Research Roadmap. **IEEE Computer Society**, p. 37-54, 2007.

FRANCE, R. et al. Model-Driven Development Using UML 2.0: Promises and Pitfalls. **Computer**. V.39 n.2, p. 59-66, Feb. 2006.

GREENFIELD, J.; SHORT, K. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Indianápolis: editoraWiley Publishing, 2004.

HAILPERN, B.; TAR, P. Model-Driven development: The good, the bad, and the ugly. **IBM Systems Journal**, v.45, n.3, p. 451-461, 2006.

KELLY S. Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. Workshop on Best Practices for Model Driven Software Development. Out. 2004.

KRASNER, G.E.; POPE S. T. A Cookbook for Using View-Controller User the Model-Interface Paradigm in Smalltalk-80. **Journal of Object-Oriented Programming**. V.1, N. 3, p. 26-49, ago./set. 1988.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and How to Develop Domain-Specific Languages. **ACM Computing surveys**. V.37, N. 4, p. 316 – 344, dec. 2005.

MUSSET, J.; JULIOT, E.; LACRAMPE. **Acceleo User Guide**. Acceleo 2.6, 2009. Disponível em <http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf >. Acesso em: set. 2009.

OBJECT MANAGEMENT GROUP - OMG. **Model Driven Architecture**. [S.l.], 2009. Disponível em: <http://www.omg.org/mda/>. Acesso em: ago. 2009.

OBJECT MANAGEMENT GROUP – OMG. **MDA Guide**. Version 1.0.1, [S.l.], Juin 2003. Disponível em: <http://www.omg.org/docs/omg/03-06-01.pdf>. Acesso em: ago. 2009.

PELECHANO, V. et al. Building Tools for Model Driven Development Comparing Microsoft DSL Tools and Eclipse Modeling Plug-ins. **Proceedings of the 11th Conference on Software Engineering and Database (JISBD'06)**. Barcelone, 2006.

SELIC B. Model-Driven Development: Its Essence and Opportunities. **Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)**, IEEE Computer Society. Washington, DC: p. 313-319, 2006.

SELIC B. The Pragmatics of Model-Driven Development. **IEEE Software**, v.20 n.5, p. 19-25, 2003.

SCHIMIDT, D.C. Model-Driven Engineering. **IEE Computer**, v.39 n.2, p.25-31, fev. 2006. Disponível em: <http://www.cs.wustl.edu/~schmidt/PDF/GEI.pdf>. Acesso em: ago., 2009.

SUN MICROSYSTEMS. **Core J2EE Patterns - Business Delegate**. [S.l.], 2009-a. Disponível em: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html>. Acesso em: 16 out. 2009.

SUN MICROSYSTEMS. **Core J2EE Patterns – Service Locator**. [S.l.], 2009-b. Disponível em: < http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html>. Acesso em: 16 out. 2009.

SUN MICROSYSTEMS. **Core J2EE Patterns – Session Facade**. [S.l.], 2009-c. Disponível em: < http://java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.html>. Acesso em: 16 out. 2009.

SUN MICROSYSTEMS. **Core J2EE Patterns – Data Access Object**. [S.l.], 2009-d. Disponível em: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>. Acesso em: 16 out. 2009.

THOMAS, D.; BARRY, B.M. Model driven development: the case for domain oriented programming . **Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications** (Anaheim, CA, USA, October 26 - 30, 2003). OOPSLA '03. ACM, New York, NY, P. 2-7, 2003.

WEIGERT, T.; WEIL, F. Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems. **Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC`06)**, p.208-217, 2006.

APÊNDICE

Este apêndice apresenta alguns trechos do código gerados a partir do cenário de testes apresentado na seção 5.3 (Figura 5.1 e Figura 5.6). Os trechos de código abaixo concernem apenas as classes *Person* e *Man*, o código referente às outras classes segue o mesmo princípio e não será apresentado no contexto deste documento.

Camada Business Object & Data Acess

Classe *Person*

Declaração da classe com as anotações EJB3 correspondentes:

```
/**
 * Person Business Object notes from design.
   @version 1.0
 */
@Entity
@Table(name = "PERSONS")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "DISCRI_PERSON", discriminatorType =
DiscriminatorType.STRING)
@SequenceGenerator(name = "person_id", sequenceName = "PersonSequence",
allocationSize = 1)
@SuppressWarnings("unchecked")
public abstract class PersonBean extends java.lang.Object
    implements Serializable, Person {
```

Declaração dos atributos diretos com as anotações EJB3 correspondentes:

```
/**
 *age class attribute
 */
@Column(name = "AGE", unique = false, nullable = true,
              insertable = true, updatable = true)
private String age;

/**
 * FEATURE 307088 Temporal annotation on Date field
 */
@Temporal(TemporalType.DATE)
@Column(name = "BIRTHDATE", unique = false, nullable = true,
              insertable = true, updatable = true)
private Date birthDate;

/**
 *id class attribute
 */
@Id
@Column(name = "PK_PERSON")
@GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "person_id")
private Long id;

/**
 *name class attribute
 */
@Column(name = "NAME", unique = false, nullable = true,
              insertable = true, updatable = true)
private String name;

/**
 *surname class attribute
 */
@Column(name = "SURNAME", unique = false, nullable = true,
              insertable = true, updatable = true)
private String surname;
```

Declaração dos atributos provenientes das associações:

```
    @ManyToOne(cascade =  {
        CascadeType.PERSIST, CascadeType.MERGE}
    , fetch = FetchType.EAGER)
    @JoinColumn(name = "FK_WOMAN", insertable = true, updatable = true)
    private WomanBean mother = null;

    /**
     * When <code>forceLoadMother</code>
     * is set to <code>true</code> the
     * <code>Mother</code>
     * Collection will be populated automatically when the
     * PersonBean class is retrieved.
     */
    @Transient
    private boolean forceLoadMother;
    @ManyToOne(cascade =  {
        CascadeType.PERSIST, CascadeType.MERGE}
    , fetch = FetchType.EAGER)
    @JoinColumn(name = "FK_MAN", insertable = true, updatable = true)
    private ManBean father = null;

    /**
     * When <code>forceLoadFather</code>
     * is set to <code>true</code> the
     * <code>Father</code>
     * Collection will be populated automatically when the
     * PersonBean class is retrieved.
     */
    @Transient
    private boolean forceLoadFather;
```

Classe *Man*

Declaração da classe com as anotações EJB3 correspondentes:

```
/**
 * Man Business Object class notes from design.
   @version 1.0
 */
@Entity
@DiscriminatorValue("MAN")
@SuppressWarnings("unchecked")
public class ManBean extends PersonBean implements Serializable, Man {
```

Atributos provenientes das associações:

```
@OneToOne(cascade =  {
    CascadeType.PERSIST, CascadeType.MERGE}
, fetch = FetchType.EAGER)
@JoinColumn(name = "FK_WOMAN", insertable = true, updatable = true)
private WomanBean wife = null;

/**
 * When <code>forceLoadWife</code>
 * is set to <code>true</code> the
 * <code>Wife</code>
 * Collection will be populated automatically when the
 * ManBean class is retrieved.
 */
@Transient
private boolean forceLoadWife;
@OneToMany(mappedBy = "father", cascade =  {
    CascadeType.ALL}
, fetch = FetchType.LAZY)
private Collection<PersonBean> children;

/**
 * When <code>forceLoadChildren</code>
 * is set to <code>true</code> the
 * <code>Children</code>
 * Collection will be populated automatically when the
 * ManBean class is retrieved.
 */
@Transient
private boolean forceLoadChildren;
```

Métodos provenientes da associação com a classe *Woman*:

```
/**
 * wife
 * @return the wife attribute
 *
 */
public Woman getWife() {
    return wife;
}

/**
 * {@inheritDoc}
 */
public void setWife(Woman wife) {
    this.wife = (WomanBean) wife;
}

/**
 * This method has to be used to set the association
 * @param c_wife the association element
 *
 */
public void addWife(Woman c_wife) {
    this.wife = (WomanBean) c_wife;

    c_wife.setHusband(this);
}

/**
 * {@inheritDoc}
 */
public void setForceLoadWife(boolean force) {
    this.forceLoadWife = force;
}

/**
 * {@inheritDoc}
 */
public boolean getForceLoadWife() {
    return this.forceLoadWife;
}
```

Métodos provenientes da associação com a classe *Person*:

```
    /**
     * {@inheritDoc}
     */
    public Collection<Person> getChildren() {
        return (Collection) children;
    }

    /**
     * {@inheritDoc}
     */
    public void setChildren(Collection<Person> children) {
        this.children = (Collection) children;
    }

    /**
     * This method has to be used to add a new child in the collection
     * @param c_children a new element in the collection children
     *
     */
    public void addChildren(Person c_children) {
        c_children.setFather(this);
        if (children == null) {
            children = new java.util.HashSet<PersonBean>();
        }
        children.add((PersonBean) c_children);
    }

    /**
     * {@inheritDoc}
     */
    public void setForceLoadChildren(boolean force) {
        this.forceLoadChildren = force;
    }

    /**
     * {@inheritDoc}
     */
    public boolean getForceLoadChildren() {
        return this.forceLoadChildren;
    }
```

Camada *Business Service*

Classe *Person*

Declaração da classe com as anotações EJB3 correspondentes:

```
/**
 * Person Service notes from design.
   @version 1.0
 */
@Stateless
@Local(PersonBusinessService.class)
public class PersonBusiness implements PersonBusinessService {
```

Serviços CRUD (Create, Read, Update e Delete) automaticamente gerados pelo NovaStudio:

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person findByPK(Person person) {
        person = (Person) entityManager.find(PersonBean.class,
                                             person.getPersonPK());
        if (person == null) {
            // record not found
            return null;
        }

        //Init associations according to lazy loading
        //forceLoad<Association> attributes
        if (person.getForceLoadMother()) {
            person.getMother();
        }
        if (person.getForceLoadFather()) {
            person.getFather();
        }
        return person;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person create(Person person) {
        person = entityManager.merge(person);
        entityManager.persist(person);
        return person;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person update(Person person) {
        person = entityManager.merge(person);
        entityManager.persist(person);
        return person;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Person remove(Person person) {
        person = entityManager.merge(person);
        entityManager.remove(person);
        return person;
    }
```

Para os serviços não-CRUD, o desenvolvedor deve acrescentar a lógica de negócios específica.

Note que todo o código acrescentado pelo desenvolvedor deve estar entre as balizas `//start of user code` e `//end of user code`. O Acceleo utiliza estas balizas para conservar o código do usuário quando uma segunda geração se faz necessária devido a modificações no modelo. Tudo o que estiver entre estas duas balizas será conservado e tudo o que estiver fora será perdido.

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
public List<Person> findAll() throws java.lang.Exception {
    //Start of user code not default 4 @warning: do not remove this comment
    //TODO : must be implemented
    return null;

    //End of user code
}
```

Classe *Man*

Declaração da classe com as anotações EJB3 correspondentes:

```
/**
 * Man Business Object class notes from design.
   @version 1.0
 */
@Stateless
@Local(ManBusinessService.class)
public class ManBusiness implements ManBusinessService {
```

Serviços CRUD (Create, Read, Update e Delete) automaticamente gerados pelo NovaStudio:

```
    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Man findByPK(Man man) {
        man = (Man) entityManager.find(ManBean.class, man.getPersonPK());
        if (man == null) {
            // record not found
            return null;
        }

        //Init associations according to lazy loading
        // forceLoad<Association> attributes
        if (man.getForceLoadWife()) {
            man.getWife();
        }
        if (man.getForceLoadChildren()) {
            man.getChildren();
        }
        return man;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Man create(Man man) {
        man = entityManager.merge(man);
        entityManager.persist(man);
        return man;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Man update(Man man) {
        man = entityManager.merge(man);
        entityManager.persist(man);
        return man;
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    public Man remove(Man man) {
        man = entityManager.merge(man);
        entityManager.remove(man);
        return man;
    }
```

Camada *Session Facade*

Domínio funcional *Family*

Como visto anteriormente existe uma única sessão *façade* por domínio funcional que reagrupa todos os serviços oferecidos pelas classes que pertencem a este domínio funcional. Desta forma, a sessão *façade FamilyFacade* reagrupa todos os serviços oferecidos pelas classes *PersonBusiness*, *ManBusiness* e *WomanBusiness*.

Declaração da classe com as anotações EJB3 correspondentes:

```
@Stateless
@Remote(FamilyService.class)
@Local(FamilyService.class)
/**
 * Applications use this Session Facade to execute the business methods
 * of the family domain.
 */
public class FamilyFacade implements FamilyService {
```

Referência para cada classe do domínio funcional:

```
    @EJB
    private PersonBusinessService personBusinessService;
    @EJB
    private WomanBusinessService womanBusinessService;
    @EJB
    private ManBusinessService manBusinessService;
```

Métodos utilizados pela camada *Presentation Layers*: cada método chama o seu correspondente na camada *Business Service* depois de ter convertido os objetos do formato utilizado pela *view* em objetos do formato utilizado pela camada *Business Layers* (classes *Entity*).

```
    /**
     * @param person
     */
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Person createPerson(Person person) {
        return (Person) BeanUtilsConverter.copyProtertiesDtoToView(null,
                           personBusinessService.create((Person)
                           BeanUtilsConverter.copyPropertiesViewToDto(null,
                                                              person)));
    }

    /**
     * @throws java.lang.Exception
     */
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public List<Person> findAllPerson() throws java.lang.Exception {
        return (List) BeanUtilsConverter.convertCollectionProperty(null,
                           (List) personBusinessService.findAll(), null);
    }


    /**
     * @param man
     */
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Man createMan(Man man) {
        return (Man) BeanUtilsConverter.copyProtertiesDtoToView(null,
                         manBusinessService.create((Man)
                         BeanUtilsConverter.copyPropertiesViewToDto(null,
                                                                man)));
    }

    /**
     * @param man
     */
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Man findByPKMan(Man man) {
        return (Man) BeanUtilsConverter.copyProtertiesDtoToView(null,
                         manBusinessService.findByPK((Man)
                         BeanUtilsConverter.copyPropertiesViewToDto(null,
                                                                man)));
    }
```