UNIVERSIDADE FEDERAL DE JUIZ DE FORA

Departamento de Ciência da Computação

Arquitetura Orientado por Modelos aplicada a Linha de Produto de Software

Eduardo Barbosa da Costa

Juiz de Fora, MG
Julho de 2008

Arquitetura Orientado por Modelos aplicada a Linha de Produto de Software

Eduardo Barbosa da Costa

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharel em Ciência da Computação

Orientador: Profa. Regina Maria Maciel Braga Villela

Juiz de Fora, MG
Julho de 2008

Arquitetura Orientado por Modelos aplicada a Linha de Produto de Software

Eduardo Barbosa da Costa

Monografia submetida ao corpo docente do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora, como parte integrante dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Aprovada pela banca constituída pelos seguintes professores:

_____

Profa. Regina Maria Maciel Braga Villela - orientadora
Doutora em Sistemas Computacionais, COPPE/UFRJ

_____

Profa. Fernanda Claudia Alves Campos
Doutora em Sistemas Computacionais, COPPE/UFRJ

_____

Prof. Marco Antônio Pereira Araújo
Mestre em Sistemas Computacionais, COPPE/UFRJ

Juiz de Fora, MG
Julho de 2008

AGRADECIMENTOS

Gostaria de agradecer aos professores da UFJF, pelo inestimável conhecimento recebido e compartilhado na área da Ciência da Computação. Agradeço, em especial, à professora Regina pelas aulas ministradas, e principalmente, por ter aceitado ser a minha orientadora. Também agradeço a todos meus amigos do curso.

Agradeço também a minha família pelo apoio e incentivo para vencer mais essa etapa em minha vida.

SUMÁRIO

Lista de Figuras

Listagens

Resumo

A demanda crescente por software e o aumento da complexidade de sistemas, causa a necessidade de melhorar a produtividade no desenvolvimento de software. Diversas técnicas e métodos de desenvolvimento estão sendo utilizados nesse sentido, visando o aumento do nível de abstração e de reuso. Entretanto, o desenvolvimento de software, na sua maioria está centrado no código, sendo os artefatos gerados no processo de modelagem utilizados somente como guias ou documentação para os desenvolvedores e analistas. A arquitetura orientada por modelos permite o preenchimento dessa lacuna, focando na modelagem e aplicação de transformações sobre modelos, para obtenção do software de maneira automatizada, tornando também os artefatos modelados parte ativa no desenvolvimento.

Para aumentar os benefícios do reuso, essas técnicas podem ser combinadas e utilizadas em uma linha de produtos de software.

Este trabalho abordará o conceito da arquitetura orientada por modelos, aplicado ao contexto do modelo de linhas de produto de software. É apresentado também, um estudo de caso utilizando esses conceitos.

1 Introdução

Historicamente, o desenvolvimento de software mostra a elevação do nível de abstração. Novas linguagens de programação, como C e Pascal, introduziram alguns modelos de programação que aumentaram a longevidade dos programas. Inevitavelmente, a demanda por características complexas e melhores prazos de entrega, resultou em outra importante evolução a orientação a objetos (uma abordagem para estruturação dos dados e comportamentos junto a classes e objetos) através de linguagens como C++, Java e Smaltalk. Essas novas linguagens tornaram mais fácil o reuso de partes do sistema em diferentes tipos de contextos.

Apesar de todas as possibilidades que surgiram com tais avanços, ainda existem problemas de portabilidade, interoperabilidade e a necessidade de integração de dados heterogêneos e distribuídos. Neste sentido, o *Object Management Group* (OMG) tem patrocinado uma iniciativa, chamada de *Model Driven Architecture* (MDA) (OMG, 2003b), cujo objetivo principal reside no uso de modelos no processo de desenvolvimento de software, através da construção de modelos independentes de plataforma (*Platform-Independent Models* ou PIMs) e sua posterior transformação em modelos específicos de plataforma (*Platform-Specific Models* ou PSMs).

Para aumentar os benefícios do reuso, essas técnicas podem ser combinadas e utilizadas em uma linha de produtos de software (LPS) (CLEMENTS; NORTHROP, 2001). A abordagem de linha de produto de software (LP) tem como objetivo promover a geração de produtos específicos por meio da reutilização de uma infra-estrutura comum estabelecida para um determinado domínio. A abordagem de LP é adequada aos domínios em que existe demanda por produtos que possuem características comuns. Os principais benefícios da utilização da abordagem (MDA) no contexto da linha de produto de software são: a utilização dos modelos em diferentes níveis de abstração no desenvolvimento da infra-estrutura central e dos componentes variáveis de cada membro da linha de produto, e a utilização das técnicas de transformação entre os modelos.

1.1 Motivação

O processo de desenvolvimento de software necessita está em constante evolução. A orientação a objetos, desenvolvimento baseado em componentes, padrões, e infra-estruturas de sistemas distribuídas são exemplos de novas abordagens, que tem melhorado o processo de desenvolvimento permitindo maior produtividade, qualidade e longevidade do software.

A Arquitetura Orientada por Modelos oferece uma importante contribuição no processo de desenvolvimento de software, a separação da especificação da aplicação de sua implementação, permite a evolução do software independente da tecnologia utilizada, tal que, o foco principal são os aspectos de negócio da solução ao invés dos aspectos técnicos. A abordagem de linha de produtos software possibilita a reutilização de partes bem especificadas, desenvolvidas e testadas, permitindo o desenvolvimento do software em menor tempo e com maior confiabilidade.

1.2 Objetivo

O objetivo deste trabalho é estudar o conceito da arquitetura orientada por modelos, o modelo de linha de produto de software, o processo de desenvolvimento utilizando a arquitetura orientada por modelos aplicada à linha de produto de software. Para exemplificar o estudo realizado, uma das propostas é apresentar um estudo de caso utilizando os principais conceitos envolvidos.

1.3 Organização do trabalho

Este trabalho está organizado em cinco capítulos. No segundo capítulo, são ilustrados os principais conceitos sobre a arquitetura orientada por modelos.

No terceiro capítulo, é apresentado o conceito do modelo de linha de produto de software. Nesse capítulo, também, são apresentadas algumas abordagens para desenvolvimento da linha de produtos.

No quarto capítulo, é detalhada uma aplicação utilizando as abordagens apresentadas. Para exemplificar este processo de desenvolvimento, é implementado um exemplo de linha de produtos de software utilizando a plataforma J2EE.

Finalmente, no quinto capítulo, são apresentadas as considerações finais, assim como os trabalhos futuros.

2 Arquitetura Orientada por Modelos

2.1 Introdução

A Arquitetura Orientada por Modelos MDA (OMG, 2003b) é um *framework* definido pelo *Object Management Group* (OMG) para o desenvolvimento de software. O fator chave para o desenvolvimento utilizando essa abordagem está na importância dos modelos no processo de desenvolvimento do software. O *framework* MDA se concentra na separação dos requisitos que o sistema deve atender para um determinado domínio, de como esses requisitos devem ser implementados no sistema para uma determinada plataforma tecnológica (MELLOR *et al*., 2004). Essa separação é realizada através de modelos que possuem diferentes níveis de abstração. A MDA divide o desenvolvimento em três modelos diferentes: o modelo independente de computação – CIM (*Computation Independent Model*), o modelo independente de plataforma – PIM (*Platform Independent Model*) e o modelo específico de plataforma – PSM (*Platform Specific Model*). Entre cada um desses modelos há uma série de transformações, que são aplicadas para que se possa chegar até uma plataforma específica (OMG, 2003b).

Os três objetivos primários da MDA são portabilidade, interoperabilidade e reusabilidade através da separação arquitetural dos interesses (OMG, 2003b). Com isso temos o aumento do poder dos modelos no desenvolvimento. O *framework* MDA é considerado orientado por modelos, pois provê um meio de utilizar os modelos em diversas fases do processo de desenvolvimento, tal como especificação, projeto, desenvolvimento, teste e manutenção.

A Figura 2.1 demonstra como poderia ser o processo de desenvolvimento utilizando a abordagem MDA.

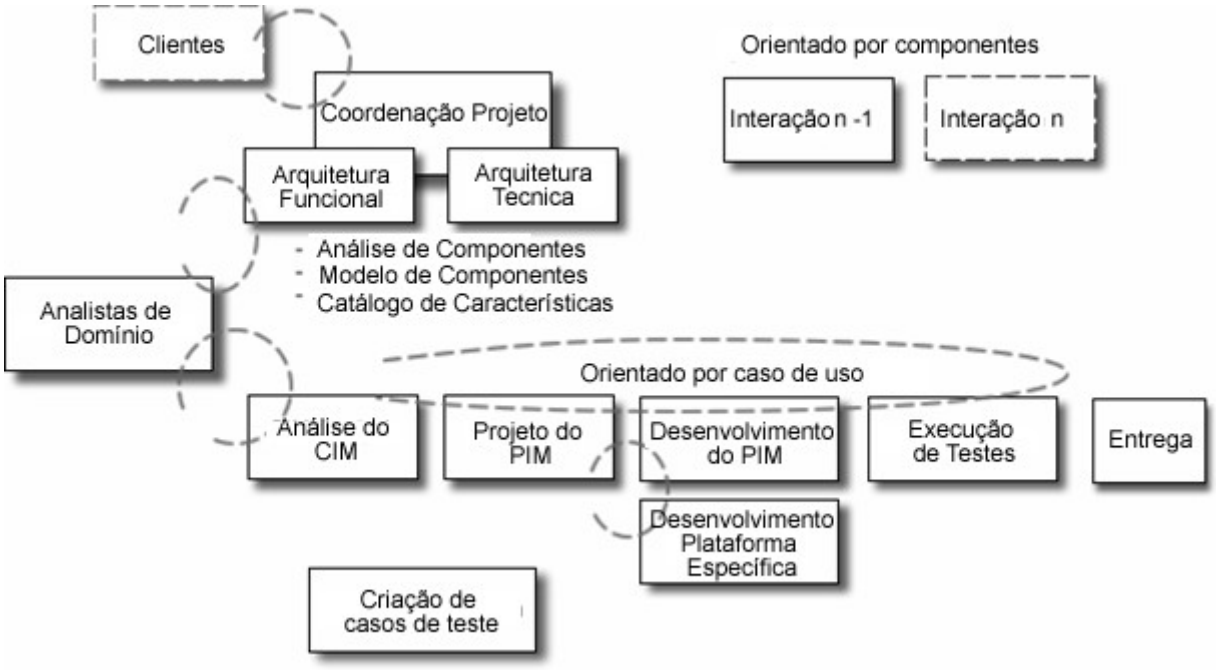


Figura 2.1: Processo de desenvolvimento de software utilizando MDA (GUTTMAN; PARODI, 2007)

Conforme dito anteriormente, a Arquitetura Orientada por Modelos especifica três pontos de vista sobre o sistema, um independente de computação (CIM), um independente De plataforma (PIM) e um específico de plataforma (PSM) (OMG, 2003b). O ponto de vista independente de computação está focado no ambiente de desenvolvimento e nos requisitos do sistema, onde os detalhes de estrutura e processamento do sistema são ocultados ou não são determinados. O ponto de vista independente de plataforma está focado na operação do sistema, sendo omitidos os detalhes necessários para executar em uma plataforma particular. Este ponto de vista independente de plataforma deve mostrar ainda que parte ou toda especificação não deva mudar de uma plataforma para outra. Por último, o ponto de vista específico de plataforma é uma combinação do ponto de vista independente de plataforma, com o foco adicional sobre os detalhes de uma plataforma.

O *framework* MDA define também a especificação da transformação entre modelos, como, por exemplo, de um PIM para PSM, e de um PSM para código. Essa transformação é realizada através da definição de mapeamentos (OMG, 2003b). No processo de utilização da abordagem MDA em MELLOR *et al.*,(2004), definimos um modelo PIM utilizando uma linguagem de modelagem, tal como a UML (*Unified Modeling Language*) (OMG, 2007a), posteriormente a execução das transformações sobre o PIM para obter PSM, seguindo da geração da implementação do sistema na plataforma tecnológica definida no mapeamento.

As seções seguintes detalham os principais conceitos.

2.2 Modelos

Os modelos podem ser definidos como um conjunto de elementos que descrevem alguma realidade física, abstrata ou hipotética (MELLOR *et al*., 2004).

No desenvolvimento utilizando MDA são criados diferentes modelos em diferentes níveis de abstração e depois interligados para formar uma implementação (WARMER; KLEPPE; BAST, 2003). Esses modelos podem ser independentes de computação CIM, independentes de plataforma PIM e específicos de uma plataforma PSM como citados anteriormente.

Modelo Independente de Computação (CIM) é uma visão do sistema do ponto de vista independente de computação, algumas vezes ele é chamado de modelo de domínio. O CIM não mostra detalhes de estrutura dos sistemas. Esse modelo possibilita uma melhor comunicação entre os desenvolvedores que têm grande experiência sobre o domínio e requisitos, com os desenvolvedores com melhor experiência em projetos e construção de artefatos, que juntos satisfazem os requisitos do domínio. Em geral durante a fase de análise de requisitos, é produzido um modelo que demonstra os requisitos do sistema. Esse modelo geralmente é produzido utilizando construções da UML como diagramas de atividades, classes, interação, colaboração, e modelos de caso de uso.

Modelo Independente de Plataforma (PIM) é uma visão do sistema do ponto de vista independente de plataforma, sendo independente de detalhes de plataforma. Pode-se fazer uma comparação com um sistema bancário, o PIM não precisaria capturar os detalhes de segurança, ou de um mecanismo de persistência.

Modelo Específico de Plataforma (PSM) é uma visão do sistema do ponto de vista especifico de plataforma. O PSM é a combinação do PIM com os detalhes que especificam como o sistema utiliza um tipo de plataforma particular. Pode existir ainda outro modelo denominado **Modelo de Plataforma**, cujo objetivo é prover ao PSM, um conjunto de conceitos técnicos que representam as partes que compõem a plataforma, assim como os diferentes serviços providos pela mesma. Um exemplo é o CORBA *Component Model* (OMG, 2002), que provê uma série de conceitos utilizados para especificar a utilização da plataforma de componentes CORBA por uma aplicação. A Figura 2.2 exemplifica um possível conteúdo de tais modelos.

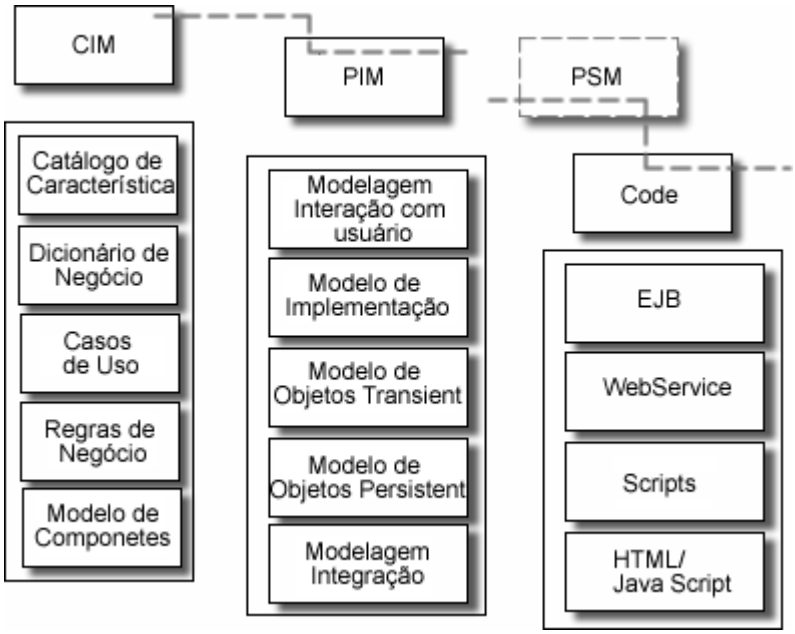


Figura 2.2: Conteúdo dos modelos CIM, PSM e PSM (GUTTMAN; PARODI, 2007)

Uma das bases no desenvolvimento utilizando MDA é a UML, que é considerada uma linguagem de modelagem padrão no desenvolvimento orientado a objetos que dá apoio a criação de modelos independentes de plataforma (MELLOR; BALCER, 2002), separando a semântica existente nos modelos da implementação destes conceitos. Para apoiar a modelagem independente de plataforma, a UML permite sua extensão através de perfis (UML *Profiles*) (OMG, 2007b). Através dos perfis, é possível definir e usar construções adicionais além daquelas já pré-definidas na UML. Isto é feito através da definição de conjuntos de estereótipos e valores etiquetados (*tagged values*). No quarto capítulo (estudo de caso) apresentamos um exemplo de modelo UML com perfis.

2.3 Metamodelos

Uma definição fundamental no desenvolvimento utilizando MDA é o conceito de metamodelo que é simplesmente um modelo da linguagem de modelagem (MELLOR *et al*.,2004). Ele define a estrutura, semântica e as restrições para uma família de modelos. Cada elemento utilizado em um modelo é captura do por um metamodelo definido através de uma meta-linguagem (WARMER; KLEPPE; BAST, 2003). Por exemplo, um modelo que utiliza diagramas da UML é capturado pelo metamodelo da UML, que descreve como os modelos UML podem ser estruturados, os elementos que o contêm, e as propriedades de uma

plataforma particular. A OMG padronizou e especificou o MOF (*Meta Object Facility*) (OMG,2006a) que é um padrão para definir linguagens de modelagem.

Os metamodelos são separados em uma hierarquia, onde são definidas quatro camadas M0, M1, M2, M3, que fundamentam a meta-modelagem (FRANKEL, 2003). A Camada M3 é chamada de meta-metamodelo. Pode-se citar como exemplo da camada M3 o padrão MOF. Um meta-metamodelo geralmente é mais compacto que um metamodelo e descreve e freqüentemente define vários metamodelos. Os meta-metamodelos e os metamodelos compartilham padrões e filosofias de construção comuns. Entretanto cada camada pode ser vista de maneira independente uma da outra. Um metamodelo é uma instância de um meta-metamodelo, ou seja, todo elemento do metamodelo é uma instância de elemento do meta-metamodelo. A responsabilidade primária da camada de um metamodelo é definir uma linguagem para especificar os modelos. Os metamodelos freqüentemente são referenciados pela camada M2. A UML e o OMG *Common Warehouse Metamodel* (CWM) (OMG, 2003a) são exemplos de metamodelos.

O metamodelo da UML é uma instância do meta-metamodelo MOF. Um modelo é uma instância de um metamodelo. A responsabilidade primária da camada de modelo é definir uma linguagem para descrever uma semântica de domínio, que permite definir uma variedade de soluções para diferentes problemas de domínio. Os modelos freqüentemente são referenciados pela camada M1. O usuário pode modelar uma instância do metamodelo usando a UML, por exemplo. A hierarquia termina com a camada M0, que contém as instâncias em tempo de execução de uma definição pertencente a um modelo, essas camadas são demonstradas na Figura 2.3.

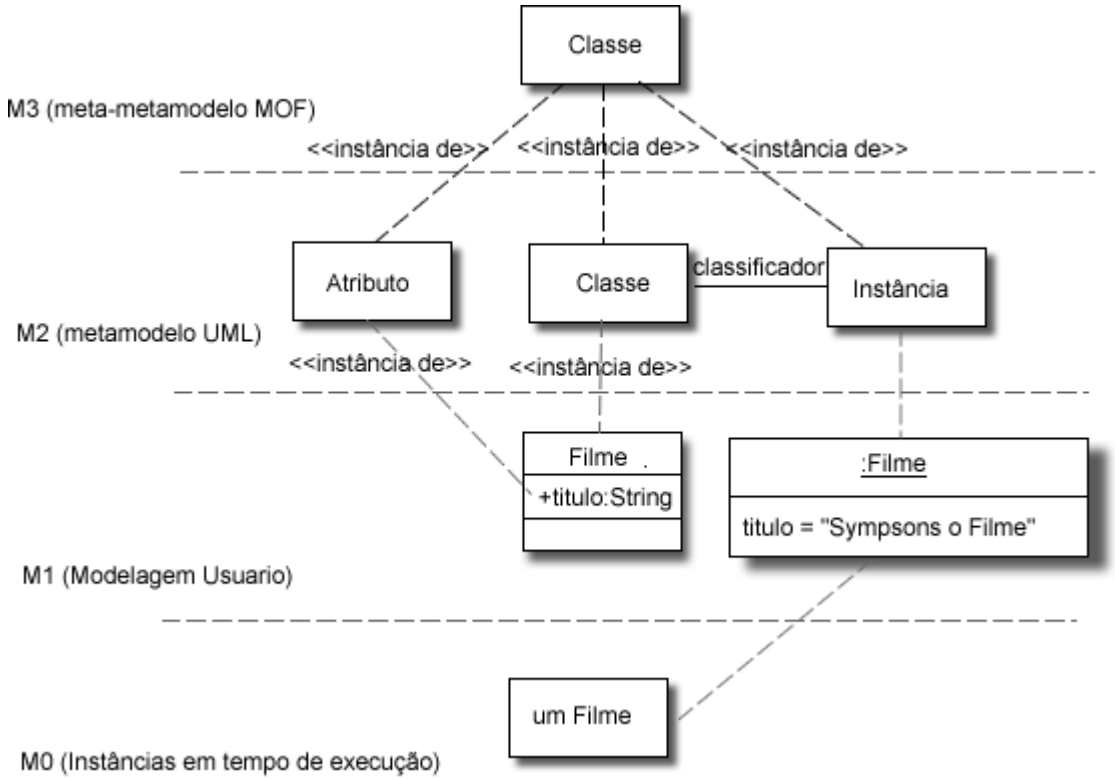


Figura 2.3: Hierarquia de metamodelos (MELLOR *et al*., 2004)

2.3.1 MOF

Um metamodelo usa MOF para definir formalmente a sintaxe abstrata do conjunto de construções de modelagem. Um metamodelo também pode especificar informalmente semânticas em linguagem natural. A combinação das definições formal e informal é chamada de MOF metamodel (OMG, 2003b).

MOF é a maneira universal para descrever construções de modelagem que um modelador possa utilizar para definir e manipular um conjunto de metamodelos interoperáveis (FRANKEL, 2003). Ele captura a estrutura e a semântica de metamodelos arbitrários.

A interoperabilidade de metamodelos através de domínios é exigida a fim de integrar ferramentas e aplicativos através de um ciclo de vida de desenvolvimento utilizando semântica comum (WARMER; KLEPPE; BAST, 2003). O MOF também fornece acesso de dados de forma padronizada, definindo padrões de suporte para serem utilizados por linguagens compatíveis com MOF, a fim de fazer intercâmbio e acesso a modelos compatíveis. Este intercâmbio pode ser baseado em XML *schema*, através de XMI (XML *Metadata Interchange*) (OMG, 2006c). O principal uso do XMI geralmente tem sido como

intercâmbio de formatos de modelos UML. Metamodelos MOF podem ser também mapeados para Java através de JMI (*Java Metadata Interface*) (SUN, 2001) que permite a interoperabilidade em nível de API para a manipulação de modelos e meta-modelos.

2.4 Mapeamentos entre Modelos

Transformação de modelos é o processo de conversão de um modelo para outro modelo do sistema (OMG, 2003b). Essa transformação pode ser feita de um PIM para PSM, depois é realizada a transformação do PSM para código fonte. Adicionalmente podem ser inseridas outras informações no processo de transformação. Essas informações são conhecidas como marcas, que indica qual elemento do modelo deve ser transformado ou manipulado. Essas transformações são essenciais no processo de desenvolvimento MDA, a Figura 2.4 demonstra o processo de transformação entre os modelos.

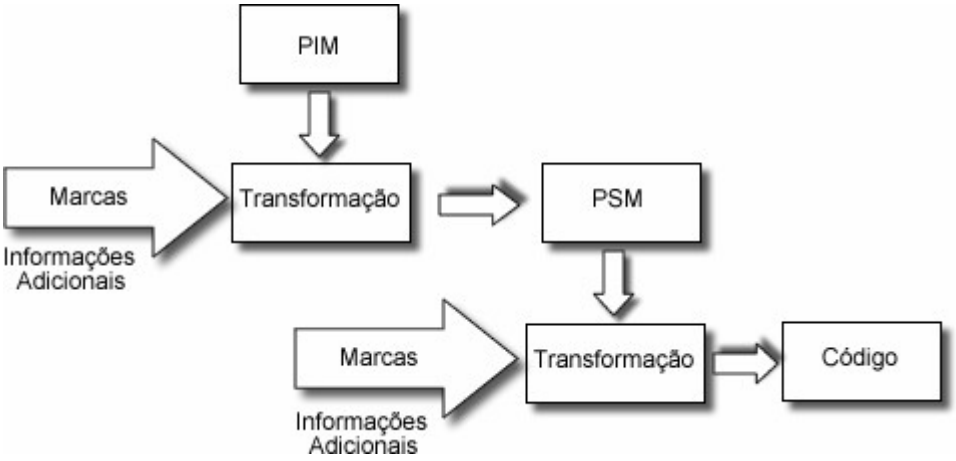


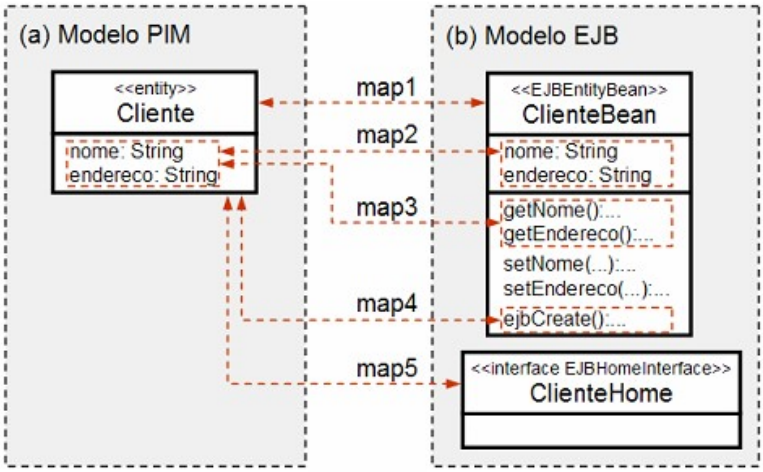
Figura 2.4: Processo de transformação entre modelos (OMG, 2003b)

A transformação entre os modelos é realizada através de um mapeamento, que é definido através de uma função de mapeamento composta de regras de mapeamento (MELLOR *et al*., 2004). As transformações entre os modelos aumentam a portabilidade e qualidade do software, reduzindo o tempo e o custo do desenvolvimento, pois as transformações são realizadas de maneira automatizada (OMG, 2003b).

Uma função de mapeamento é uma coleção de regras ou algoritmos que define como funciona um mapeamento em particular. Um mapeamento é a aplicação ou execução de uma função de mapeamento para transformar um modelo em outro (MELLOR *et al*.,2004).

Um mapeamento é especificado utilizando uma linguagem para descrever uma transformação de um modelo em outro (MAIA, 2006). Essa descrição deve ser em linguagem natural, em um algoritmo, em uma linguagem de ação, ou em uma linguagem de mapeamento de modelos. A linguagem de mapeamento de modelos proposta atualmente é a MOF *Query View Transformation* QVT (OMG, 2004), que propõe uma maneira para criar consultas a modelos, criar visualizações sobre o modelo e escrever definições de transformação.

A Figura 2.5 demonstra uma mapeamento entre um modelo independente de plataforma e um modelo específico de plataforma.




```
<transformation-map name="EJBComponents" ...>
    <classifier-map id="map1" name="Entity-Class-PIM - EntiyBean-Class-EJB" ...>
        <feature-map id="map2" name="Attribute-PIM - Attribute-EJB" ...> ...
        </feature-map>
        <feature-map id="map3" name="Attribute-PIM - GetterMethod-EJB" ...> ...
        </feature-map>
        <classifier-feature-map id="map4" name="Entity-Class-PIM - ejbCreate-Method-EJB" ...>...
        </classifier-feature-map>
    </classifier-map>
    <classifier-map id="map5" name="Entity-Class-PIM - EntityBean-HomeInterface-EJB" ...>...
    </classifier-map>
    ...
</transformation-map>
```

Figura 2.5: Transformação entre modelos (MAIA, 2006)

O *framework* MDA define quatro métodos para o mapeamento entre os modelos (OMG, 2003b):

- **Transformação manual** – semelhante à forma como o projeto de software vem sendo realizado, onde decisões são tomadas para criar um projeto de software de acordo com os requisitos previamente definidos. O *framework* MDA realiza uma distinção explícita de um modelo independente de plataforma, com a transformação para o modelo específico de uma plataforma.

- **Transformação de PIM preparado com perfil** – um modelo PIM é manipulado por um especialista, que realiza marcações utilizando um perfil UML independente de plataforma. Posteriormente, o PIM é transformado para um PSM expresso por um segundo perfil UML, este específico de plataforma.
- **Transformação utilizando padrões e marcações** – padrões podem ser utilizados para definir mapeamentos. Marcações correspondem a elementos desses padrões. Os elementos de um PIM são marcados e transformados de acordo com o padrão estabelecido no mapeamento, produzindo um PSM. Outra possibilidade é utilizar regras especificando que todos os elementos no PIM, que seguem um determinado padrão, serão transformados em instâncias de outro padrão, no PSM.
- **Transformação automática** – existem contextos onde um PIM pode conter toda a informação necessária para a implementação, ou seja, o PIM é completo em relação à sua classificação, estrutura, invariantes, pré e pós-condições. Nestes casos, o desenvolvedor pode especificar o comportamento diretamente no modelo, utilizando uma linguagem de ações. Essa linguagem faz com que o PIM seja computacionalmente completo, e automaticamente transformado para o código-fonte do programa.

As marcas são recursos importantes durante a transformação entre os modelos. Elas são entradas adicionais no processo de mapeamento, que capturam a informação exigida para a transformação de modelos sem poluir esses modelos (MELLOR *et al*., 2004). Um mapeamento pode usar várias marcas associadas aos modelos de origem. As marcas podem ser utilizadas em dois contextos: como entradas adicionais, que podem ser utilizadas para antecipar decisões de projeto ou em funções de mapeamentos, e como saídas adicionais, que servem como um tipo de registro do processo de transformação de um modelo de origem para um modelo de destino.

Existe também o conceito de *templates* (OMG, 2003b) que é definido por um conjunto de modelos parametrizados, que especificam uma transformação particular. Esses *templates* são como padrões de projeto (GAMMA; HELM; JOHNSON, 2000). Um conjunto de marcas pode ser associado com um *template* para indicar instâncias de um modelo que deve ser transformada de acordo com o *template*. Outras marcas podem ser utilizadas para indicar valores no modelo que serão utilizados para preencher parâmetros do *template*. Isto permite que valores no código do modelo possam ser copiados do modelo de destino, e alterados, se

necessário. A figura 2.6 demostra um exemplo de *template* que define uma arquitetura para os modelos específicos de plataforma.

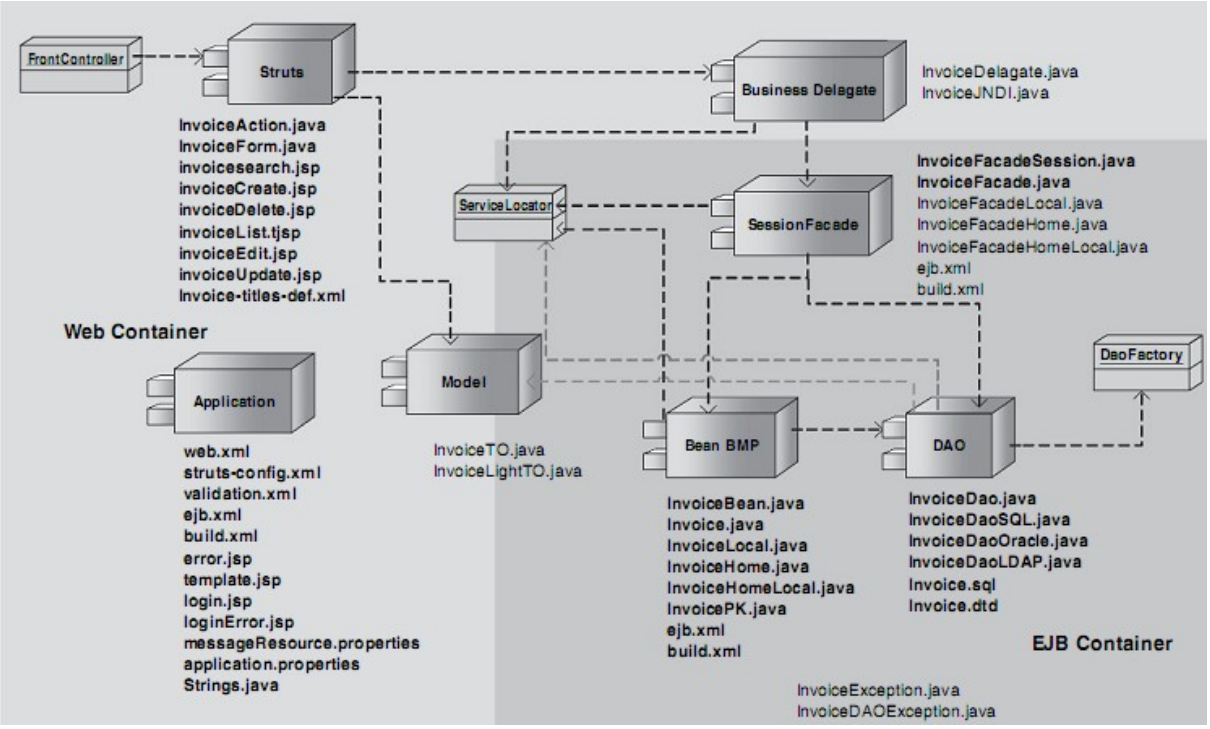


Figura 2.6: Template arquitetura J2EE (GUTTMAN; PARODI, 2007)

2.5 Linguagens de Modelagem

Uma linguagem é a sintaxe de abstração, é a estrutura da linguagem, separada de seus símbolos notacionais concretos (MELLOR *et al*., 2004). O metamodelo UML, por exemplo, não se refere aos formatos utilizados para representar os elementos de modelo em diagramas, e modelos UML não podem nem ao menos ser representados textualmente. O metamodelo captura somente a sintaxe abstrata de uma linguagem.

Uma maneira de definir uma linguagem envolve o uso de MOF, que suporta diversos conceitos importantes que podem ser utilizados como base para a nova linguagem. Outra maneira envolve a extensão da UML por meio de perfis, que são mecanismos para adaptar um metamodelo existente com construções específicas a um domínio, plataforma ou método particular. Não serão mostrados os detalhes de como criar uma linguagem de modelagem, mas um conceito muito importante deve ser citado que é o de restrições. Para mais detalhes sobre linguagem de modelagem, consultar (FRANKEL, 2003).

2.5.1 Restrições

A especificação da UML define uma sintaxe abstrata do conjunto de diagramas definidos por ela, mas ela não define exatamente como esses diagramas se encaixam juntos para formar um conjunto consistente (MELLOR; BALCER, 2002). A UML utiliza a *Object Constraint Language* (OCL) (OMG, 2006b) para definições dessas necessidades. A seguir são mostrados alguns conceitos e definições sobre OCL:

OCL é uma linguagem formal utilizada para expressar condições que devem se manter verdadeiras para o sistema que está sendo modelado (OMG, 2006b). Quando uma expressão OCL é avaliada, ela simplesmente retorna um valor, e não altera o modelo. O que significa que não altera o estado do sistema quando a expressão é avaliada. Informações independentes de plataforma como pré-condições, pós-condições e invariantes podem ser expressas em UML através de expressões OCL (FRANKEL, 2003). Expressões OCL geralmente aparecem em diagramas UML dentro de notas ou das definições de classes. Um dos fatos importantes para se utilizar uma linguagem de restrição, é que restrições agregam precisão aos modelos, e não são ambíguas.

As restrições também podem ser colocadas em prática quando você define estereótipos que estão relacionados uns com os outros de alguma maneira (MELLOR *et al*., 2004), por exemplo, foi definido um estereótipo chamado <<*Session*>>, aplicável a classes, e outro chamado <<*Entity*>> aplicável somente àquelas classes não estereotipadas com <<*Session*>>, Se não for restrita de outra forma, a especificação da UML permitirá que se atribua ambos os estereótipos a uma única classe ao mesmo tempo. Portanto, deve-se adicionar uma restrição OCL, que permitirá, no máximo, que um desses dois estereótipos seja aplicado a uma classe única a qualquer momento.

As expressões OCL requerem que as restrições estejam ligadas a um contexto de um modelo. O contexto de uma expressão pode ser uma classe de objetos ou pode ser uma operação aplicável a um objeto.

Para representar um contexto em OCL utilizamos a palavra reservada *context* <contexto>. A seguir é apresentado um exemplo de invariante.

"Todo veículo deve ter pelo menos uma roda"

context Veiculo **inv**:

self.numRodas > 0

2.6 Modelos Executáveis

Modelos executáveis (MELLOR; BALCER, 2002) atuam exatamente como o código, de certa forma, embora eles forneçam também a habilidade de interagir diretamente com o domínio do cliente. Eles não são exatamente como o código, porque eles precisam ser configurados com outros modelos para produzir o sistema. Isso geralmente é feito através de um compilador de modelos (MELLOR; BALCER, 2002).

Assim como as linguagens de programação são independentes de hardware, os modelos executáveis conferem à independência de plataforma de software, o que torna os modelos executáveis portáveis em vários ambientes de desenvolvimento. Uma maneira de especificar um modelo executável é através da UML executável (MELLOR; BALCER, 2002), que é um perfil da UML que define uma semântica de execução para um subconjunto da UML. O subconjunto é computacionalmente completo, portanto um modelo da UML executável pode ser diretamente executado.

2.6.1 UML Executável

A UML executável eleva o nível de abstração, fornecendo um meio de abstrair a linguagem especifica de programação de decisões sobre a organização do software (MELLOR; BALCER, 2002). Uma especificação construída com a UML executável pode ser executada em vários ambientes de software sem mudanças. Fisicamente, uma especificação UML executável inclui um conjunto de modelos representados como diagramas que descrevem e definem um formalismo e comportamento de um estudo do mundo real ou hipotético. Esse conjunto de modelos juntos abrange uma especificação simples que se pode examinar de vários pontos de vista diferentes. Existem três projeções fundamentais para uma especificação (MELLOR; BALCER, 2002), vistos a seguir. O primeiro modelo identifica, classifica e abstrai o estudo do mundo real ou hipotético, e organiza a informação dentro de uma estrutura formal. Os objetos são identificados e abstraídos como classes, características dos objetos abstraídas como atributos, e relações entre os objetos são abstraídas como relacionamentos. O segundo descreve o ciclo de vida dos objetos (comportamento no tempo), que são abstraídos através de máquina de estados, que é representado através de diagramas de estados da UML. O comportamento do sistema é dirigido pelo objeto, à mudança de estado do

objeto é gerada por eventos de outros objetos e cada estado pode ter um conjunto de procedimentos associados. O terceiro compreende um conjunto de ações associadas a um procedimento. Ações são partes fundamentais da computação de sistemas, e cada ação é uma unidade primitiva da computação, como um acesso a dado, uma seleção, ou um loop. A UML recentemente definiu uma semântica de ações, ainda não padronizada. A Tabela 2.1 resume essas projeções.

Conceito	Chamado	Modelado como	Expressado como
O mundo e todas as coisas	Dado	Classes, atributos, associações, restrições	Diagrama de classe UML
Coisas têm ciclo de vida	Controle	Estados, eventos, transições, procedimentos	Diagrama de estado UML
Coisas fazem coisas a cada estado	Algoritmo	Ações	Linguagem de ação UML

Tabela 2.1: Conceitos dos modelos da UML executável (MELLOR; BALCER, 2002)

A Figura 2.7 demonstra uma exemplo de especificação utilizando os conceitos da UML executável.

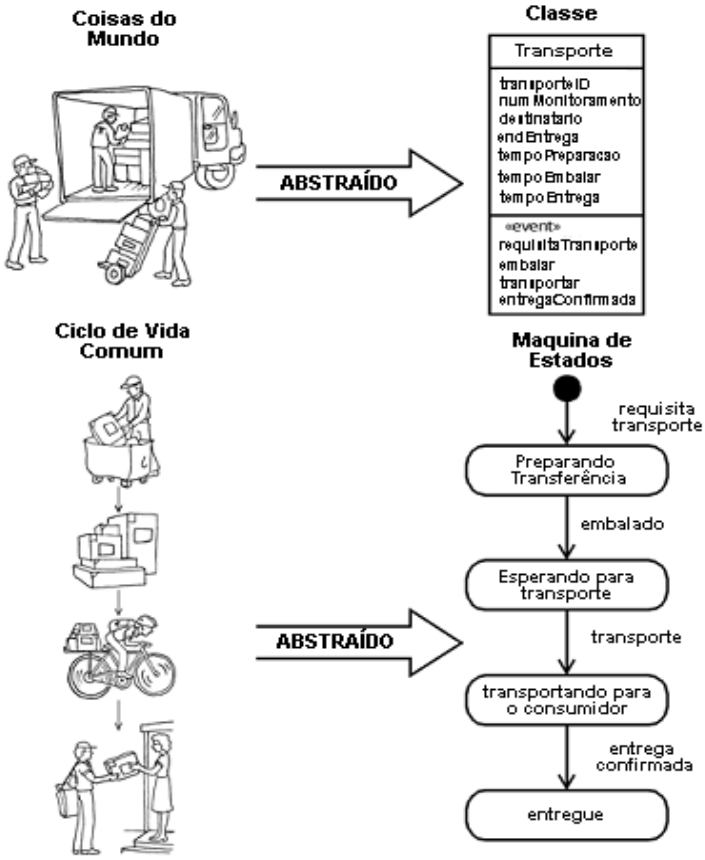


Figura 2.7: Exemplo especificação UML executavél (MELLOR; BALCER, 2002)

2.7 Considerações Finais

A Arquitetura Orientada por Modelos eleva o nível de abstração de modo que as pessoas possam se expressar elas mesmas através dos modelos com a máquina, aumentando a produtividade, pois os artefatos de modelagem deixam de ser apenas um custo "necessário", porém sem agregar valor significativo para o desenvolvimento do produto final, para ser parte ativa na maioria das fases do desenvolvimento do sistema. O nível de reuso também é elevado, pois os modelos podem ser reutilizados, tanto para combinação com outros modelos, como para a transformação para diferentes plataformas, com um custo bem menor que no desenvolvimento atual centrado em código, que para se conseguir essa mudança temos que reconstruir todo o código ou grande parte do sistema. Nesse sentido, o *framework* MDA permite conseguir a portabilidade de plataforma, especificando uma plataforma de destino e definindo como os modelos devem ser mapeados para essa plataforma.

A abordagem MDA também possibilita um salto qualitativo para o desenvolvimento, através da introdução do desenvolvimento dos modelos, onde os modelos independentes de

plataforma são transformados para a obtenção dos modelos específicos de plataforma, e então a implementação em uma plataforma particular de maneira automatizada.

Entretanto, para ser viável um processo de construção de software MDA é necessário ter, pelo menos, profissionais capacitados e ferramentas que auxiliem nas tarefas de modelagem e transformação.

Neste capítulo foram apresentados os conceitos fundamentais para o desenvolvimento utilizando a Arquitetura Orientada por Modelos, no capitulo seguinte será apresentado o conceito de modelo de linha de produto de software no contexto do MDA, que será utilizado como base para o estudo de caso desse trabalho.

3 Linha de Produto de Software

3.1 Introdução

O reuso de software tem sido um dos objetivos de muitas empresas de tecnologia (REINEHR *et al*., 2006) englobando diversas tecnologias como a orientação a objetos, desenvolvimento baseado em componentes (BACHMANN *et al*., 2000), entre outros. Todas essas tecnologias trouxeram várias possibilidades de reuso e aumento da qualidade no desenvolvimento do software. Entretanto, o resultado final tem sido menor que o esperado (GOMAA, 2004). O maior problema é que a maioria dessas tecnologias dá ênfase no reuso de código, e estudos indicam que a codificação representa somente 20% no custo total do software, ou seja, um investimento na reutilização somente do código não teria um valor muito expressivo. Desta forma deve-se reutilizar não somente o código, mas os requisitos e arquiteturas de software, que é um objetivo do modelo de linha de produto de software.

O modelo de linha de produto de software, *Software Product-Line* (SPL) (GOMAA, 2004), consiste de uma família de sistemas de software que possuem funcionalidades em comum e outras funcionalidades variáveis (funcionalidades que diferem entre os sistemas). Segundo Cohen (COHEN, 2002) a definição para *software product line* seria um conjunto de sistemas que usam software intensivamente, compartilhando um conjunto de características comuns e gerenciadas, que satisfazem às necessidades de um segmento particular de mercado ou missão, e que são desenvolvidos a partir de um conjunto comum de ativos principais e de uma forma preestabelecida.

O desenvolvimento de linha de produto de software surgiu a partir do reuso de software, quando os desenvolvedores e analistas perceberam que poderiam obter melhores benefícios com o reuso de toda uma arquitetura de software e não somente reutilizar componentes individuais (GOMAA, 2004), diferentemente do modelo de desenvolvimento de software convencional, que é centrado no desenvolvimento de um único sistema. A abordagem de desenvolvimento utilizando linha de produto de software considera uma família de sistemas de software (DONEGAN; MASIERO, 2007). Essa abordagem envolve análise de quais características (requisitos funcionais) da família de software têm em comum. Depois de analisar as características, o objetivo é desenvolver a arquitetura para linha de

produtos, identificando componentes comuns (*core*), componentes variáveis e componentes opcionais.

Algumas abordagens de desenvolvimento de uma linha de produto de software provêem uma arquitetura genérica, ou modelo de referência, para a linha de produto, onde são consideradas as similaridades da linha de produto, mas ignorando as variações (WFMC, 1995). Cada aplicação começa com uma arquitetura genérica, que é adaptada manualmente aos requisitos que são variáveis na aplicação. Deve-se ressaltar o fato de que essa abordagem é um bom ponto de partida em comparação ao desenvolvimento sem reuso, mas falha ao não capturar o conhecimento sobre a variabilidade da família de produtos (GOMMA, 2004).

Uma abordagem mais adequada para a modelagem da arquitetura da linha de produtos de software é a modelagem de ambos os requisitos da linha de produto, os comuns e os variáveis. Dependendo da abordagem de desenvolvimento utilizada (funcional ou orientada por objetos), a similaridade da linha de produtos é descrita em termos de modelos comuns, classes, ou componentes, e a variabilidade da linha de produtos e descrita em modelos opcionais ou módulos variáveis, classes ou componentes (GOMAA, 2004).

Os principais objetivos da modelo de linha de produto de software é o de reunir semelhanças e gerir as variações para obter melhorias na qualidade, tempo e esforço de desenvolvimento, custo e complexidade na criação e manutenção de linhas de produtos de sistemas de software similares, e ainda aumento do numero de produtos dessa linha que podem ser desenvolvidos (NADA *et al*., 2000).

Nas seções seguintes serão detalhados alguns conceitos importantes no desenvolvimento utilizando a abordagem de linhas de produto de software.

3.2 Atividades Essenciais

O SEI (*Software Engineering Institute*) (SEI, 2008), através da iniciativa PLP (*Product Line Practice*), estabeleceu as atividades essenciais de uma abordagem de LP (*Product-Lines*) (SEI, 2008). Essas atividades são o Desenvolvimento do Núcleo de Artefatos, também conhecida como Engenharia de Domínio, o Desenvolvimento do Produto, também conhecida como Engenharia de Aplicação, e o Gerenciamento da Linha de Produto.

A Figura 3.1 demonstra as interações entre essas atividades.

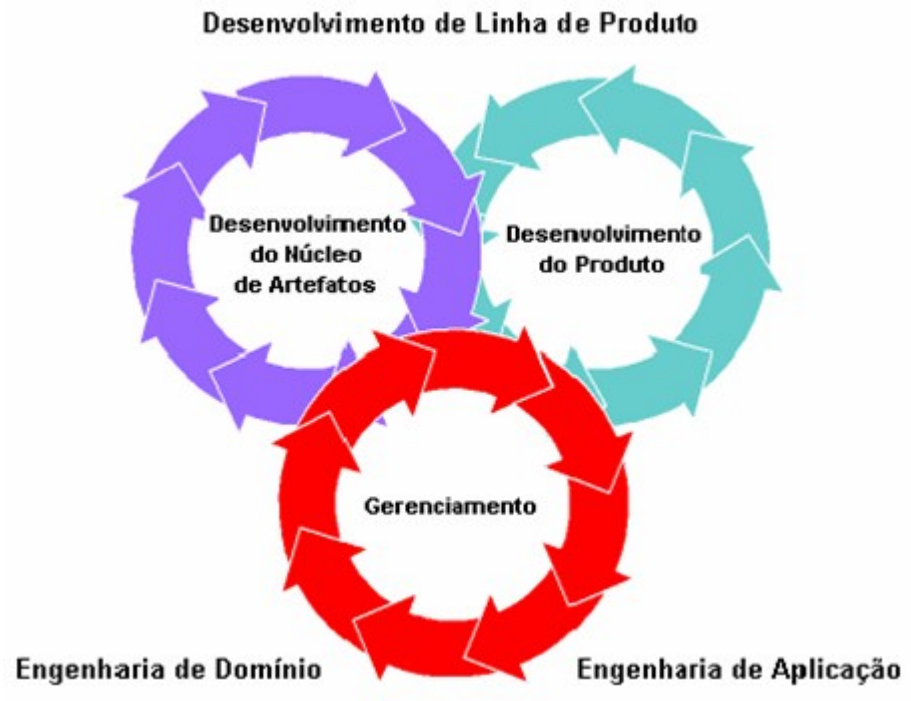


Figura 3.1: Atividades essenciais de linha de produto de software (SEI, 2008)

Desenvolvimento do núcleo dos artefatos

No desenvolvimento do núcleo de artefatos três aspectos devem ser considerados: a definição do contexto da linha de produto, o núcleo de artefatos e o plano de produção. Um artefato é um item reutilizável de software utilizado como bloco de construção de uma linha de produto. Pode ser uma arquitetura de software, um componente, um *framework*, ou alguma documentação relativa à arquitetura de software ou a algum componente.

Desenvolvimento do produto

A atividade de desenvolvimento do produto tem como principal objetivo a geração de produtos de uma LP. Porém, é possível identificar requisitos que, inicialmente, não haviam sido especificados e, conseqüentemente, atualizar o núcleo de artefatos da LP. Essa atividade depende substancialmente dos artefatos de saída da atividade anterior, que lhe servem como entrada. Esses artefatos são o contexto da LP, o núcleo de artefatos e o plano de produção. Além desses artefatos, são necessários também os requisitos para um produto específico.

Gerenciamento da linha de produtos

A atividade de gerenciamento de linha de produto deve garantir que todas as atividades técnicas sejam realizadas de acordo com um planejamento coordenado. O gerenciamento pode ser dividido em duas categorias (SEI, 2008):

- **gerenciamento técnico** - coordena as atividades de desenvolvimento do núcleo de artefatos e desenvolvimento do produto para garantir que as equipes de desenvolvimento sigam os processos definidos para a linha de produto e coletem dados suficientes para acompanhar o progresso desta.
- **gerenciamento organizacional** – garante que as unidades organizacionais recebem os recursos corretos (ex. treinamento) em quantidades suficientes. Uma das ações mais importantes da atividade de gerenciamento da linha de produto é a criação de um plano de adoção que descreva o estado desejado da organização e uma estratégia para alcançar tal estado.

O gerenciamento de linha de produto também deve estabelecer como as atividades de desenvolvimento do núcleo de artefatos e desenvolvimento do produto devem interagir para permitir a evolução da linha de produto e o gerenciamento das semelhanças e das variabilidades de cada artefato da linha de produto.

3.3 Modelagem de Variabilidade

Um conceito importante no desenvolvimento utilizando a abordagem de linhas de produto de software é o conceito de *features* (características) (KANG, 1990). Uma *feature* representa os requisitos reutilizáveis do sistema. Essas características são aplicadas em qualquer linha de produto e não somente a linha de produto de software, e são utilizadas para a identificação das características comuns, alternativas e opcionais da linha de produto.

Existem alguns métodos de análise de domínio que utilizam *features*. O modelo FODA (*Feature-Oriented Domain Analysis*) (KANG, 1990). é um exemplo. O método FODA organizada as características em uma árvore. As *features* podem ser obrigatórias, opcionais ou mutuamente exclusivas. Essas características devem ser características funcionais (ex: hardware ou software), características não funcionais (ex: segurança ou desempenho) ou parâmetros (ex: remota ou local). Existem outros métodos na literatura dentre eles pode-se citar: *Family-Oriented Abstraction Specification and Translation* (FAST) (WEISS; CHITAU, 1999), *Product Line Software Engineering* (PuLSE) (BAYER; FLEGE, 1999), a abordagem proposta por Bosch (BOSCH, 2000), a iniciativa *Product Line Practice* (PLP) (CLEMENTS; NORTHROP, 2001), o método KobrA (ATKINSON; NORTHROP, 2001), e mais recentemente, a abordagem *Product Line UML-Based Software Engineering* (PLUS) (GOMAA, 2004).

GOMAA, (2004) descreve três tipos de modelos de variabilidade de linha de produto de software:

- **Modelagem de variabilidade utilizando parametrização** – parâmetros podem ser utilizados para introduzir variabilidade entre a linha de produto de software. Os valores dos parâmetros definidos nos componentes da linha de produto de software definem onde começa a variabilidade. Diferentes membros da linha de produto devem possuir diferentes valores de parâmetro.
- **Modelagem de variabilidade utilizando informações escondidas** – informações escondidas podem também ser utilizadas para introduzir variabilidade dentro da linha de produto de software. Diferentes versões dos componentes têm a mesma interface, mas possuem diferentes implementações para diferentes membros da linha de produto. Neste caso, as variantes são as diferentes versões do mesmo componente, mas que devem ter a mesma interface.
- **Modelagem de variabilidade utilizando herança** - a terceira maneira de introduzir a variabilidade da linha de produtos de software é utilizando herança. Neste caso, diferentes versões de uma classe utilizam herança para herdar operações de uma superclasse, e então às operações especificadas na interface da superclasse são redefinidas e/ou a interface é estendida para adicionar novas operações.

Cada uma dessas abordagens têm suas vantagens e desvantagens, conforme apresenta Gomaa em um comparativo (GOMAA, 2004):

- **Modelagem de variabilidade utilizando parametrização** - permite que a aplicação defina os valores dos atributos da linha de produtos que são mantidos pelos vários componentes da linha de produtos. Se toda a variabilidade puder ser definida em termos de parâmetros, então essa é uma abordagem simples de prover variabilidade. Porém, a variabilidade fica limitada, pois nenhuma funcionalidade pode ser alterada.
- **Modelagem de variabilidade utilizando informações escondidas** – permite escolher variantes de um conjunto limitado, no qual a interface é comum, mas a implementação é variável. Informações escondidas permitem um maior grau de variabilidade que o modelo que utiliza parametrização, pois as funcionalidades e parâmetros podem ser variáveis.

- **Modelagem de variabilidade utilizando herança** - permite escolher variantes cuja funcionalidade pode variar, essa abordagem permite larga extensão de variantes, por estender a interface da superclasse na subclasse variante.

Em muitas linhas de produto, uma combinação dessas três abordagens é necessária. A abordagem orientada a objetos para o desenvolvimento de software ajuda por suportar todas as três abordagens para modelagem de variabilidade.

3.4 Modelagem de Linha de Produto de Software

No final da década de 90, início de 2000, o principal tipo de artefato de software reutilizável era componente. Com o advento da linha de produto de software, o foco voltou novamente para o reuso em larga escala (DONEGAN; MASIERO, 2007). As abordagens de modelagem de software estão sendo amplamente utilizadas no desenvolvimento de software e têm grande importância no desenvolvimento de linha de produto de software. Com a utilização de uma abordagem de modelagem, por exemplo, utilizando a UML, pode-se conseguir um maior entendimento e gerenciamento das similaridades e variabilidades, por modelar a linha de produtos em diferentes perspectivas (GOMAA, 2004).

Um método para modelagem de linhas de produto utilizando a UML é o PLUS (*Product Line UML-Based Software Engineering*) descrito em (GOMAA, 2004), que estende a modelagem baseada em UML com o intuito de contemplar o conceito de linhas de produto de software. O objetivo do PLUS é modelar explicitamente as similaridades e variabilidades da linha de produto. PLUS provê um conjunto de conceitos e técnicas para estender métodos de desenvolvimento baseado em UML e processos para sistemas *standalone*, para manipular linhas de produto de software.

O PLUS é similar a outros métodos orientados a objeto baseados em UML quando utilizado para análise e modelagem de sistemas. A diferença com relação aos outros métodos está na forma como estende os métodos orientados a objeto para modelar famílias de produtos.

3.5 Modelo de Processo SPL

Quando se considera o desenvolvimento de uma família de produtos que constitui uma linha de produtos, é necessário modificar o modelo de processo tradicional, que desenvolve

atividades para sistemas simples através de modelagem de requisitos, modelos de análise, e modelagem de projetos, para desenvolvimento de atividades relacionadas a família de produtos (DONEGAN; MASIERO, 2007).

No processo de desenvolvimento utilizado SPL segundo GOMAA, (2004) existem dois tipos de processos principais, engenharia da linha de produto e engenharia da aplicação.

Engenharia da linha de produto: durante esse processo, a similaridade e variabilidade são analisadas levando em consideração os requisitos globais da linha de produto. Está atividade consiste em desenvolver a arquitetura, casos de uso, modelos de análise, e os componentes reutilizáveis da linha de produto. Testando e validando os componentes, bem como algumas configurações da linha de produto. Os artefatos gerados durante o processo são persistidos em um repositório para linha de produto.

Engenharia da aplicação: durante esse processo, uma aplicação individual da linha de produto é desenvolvida. Em lugar de partir do zero, como é geralmente feito em sistemas simples, os desenvolvedores da aplicação fazem uso de todos os artefatos gerados na engenharia da linha de produto de software. Tratando todos os requisitos da aplicação individualmente, os modelos de caso de uso da linha de produto são adaptados para derivar os modelos de caso de uso da aplicação, os modelos de análise são adaptados para derivar os modelos de análise da aplicação, e a arquitetura da linha de produto de software é adaptada para a arquitetura de software da aplicação. Utilizando os componentes apropriados do repositório da linha de produto de software, a execução da aplicação é disponibilizada.

A Figura 3.2 demonstra um processo de desenvolvimento utilizando linha de produto de software.

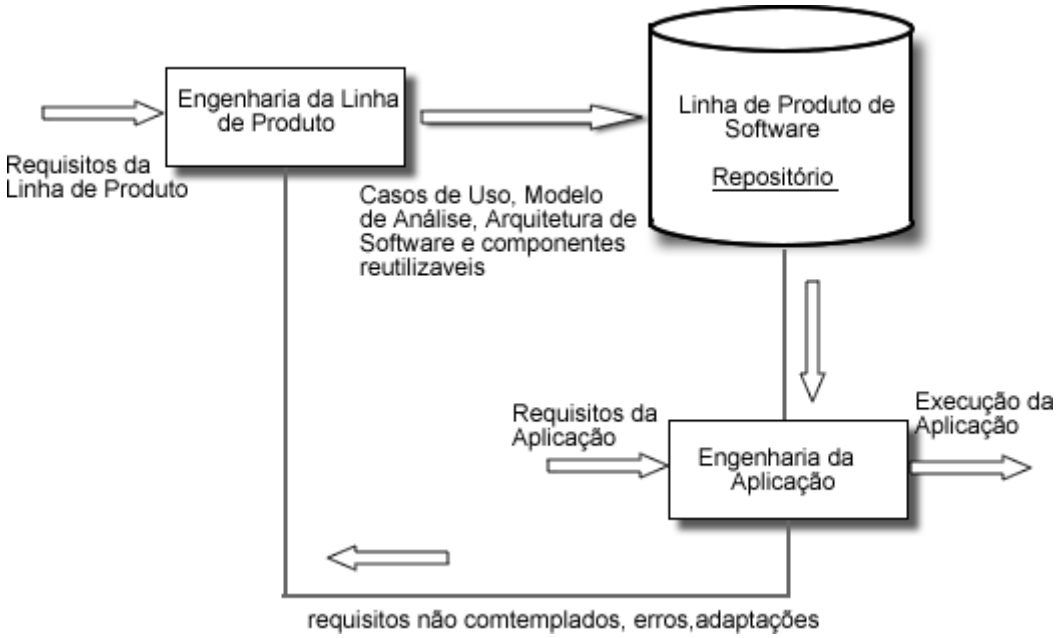


Figura 3.2: Processos de desenvolvimento de linha de produto de software (GOMAA, 2004)

3.6 MDA e SPL

Como foi apresentado no capitulo anterior, MDA é um *framework* padronizado pela OMG para o desenvolvimento de software, que é baseado no desenvolvimento de modelos em diferentes níveis de abstração, e a transformação desses modelos para a obtenção do software de maneira automatizada. Nesse contexto, a UML pode ser utilizada para a modelagem dos modelos. A arquitetura orientada por modelos pode ser utilizada para desenvolver arquiteturas de software baseada em componentes para linha de produto, expressando os artefatos via UML em modelos independentes de plataforma (PIM) (DEELSTRA *et al*., 2003).

Os métodos de análise e projeto orientados a objeto utilizados no desenvolvimento de sistemas simples necessitam ser estendidos para o desenvolvimento utilizando a abordagem de linha de produtos, em particular, para atender os modelos de similaridade e variabilidade. O objetivo é estender a notação UML utilizando o mecanismo de extensão padrão da UML através do uso de perfis.

Em (DEELSTRA *et al*., 2003), é discutido sobre o gerenciamento da variabilidade em linha de produto de software. Conceitualmente um sistema de software que é especificado de acordo com abordagem MDA, especifica um modelo de aplicação para uma linha de produto consistindo de produtos que implementam a mesma funcionalidade sobre plataformas diferentes. Essa escolha por plataformas diferentes é um ponto de variação para uma linha de

produtos. O principal benefício do desenvolvimento utilizando MDA é que o gerenciamento do ponto de variação da plataforma é realizado de forma automatizada através das transformações. A plataforma, entretanto, não é um ponto de variação que precisa ser gerenciado em uma linha de produtos (DEELSTRA *et al*., 2003). Os vários membros da linha de produtos diferem a nível conceitual, em suas características funcionais, bem como em nível de infra-estrutura, além de compartilharem um núcleo do software e implementarem características alternativas. O desenvolvimento utilizando MDA pode ser estendido para se adaptar as características da variabilidade da linha de produtos, por adicionar um modelo de domínio que especifica características alternativas que podem ser selecionadas. A seleção de diferentes características do modelo de domínio resulta em diferentes modelos de aplicação, provendo uma maneira correta de definição da transformação de acordo com a abordagem utilizando MDA. Neste contexto (DEELSTRA *et al*., 2003) introduz um modelo de família de produtos de software configurável que é utilizada na transformação do modelo, podendo variar a plataforma ou utilizando outro tipo de variabilidade.

A Figura 3.3 demonstra o processo de transformação MDA adaptado às necessidades da variabilidade da linha de produto de software a nível conceitual. As características do domínio especificam as funcionalidades da linha de produtos independente de detalhes de implementação do núcleo ativo (*asset base*). A definição da transformação então define como as características do domínio são mapeadas sobre o núcleo ativo. Baseado nos requisitos do produto, um modelo de aplicação é derivado selecionando características alternativas especificadas no modelo de domínio. O modelo de aplicação é então compilado para uma aplicação utilizando a definição da transformação, o núcleo base e os requisitos do produto.

A Figura 3.3 também demonstra dois principais processos da engenharia de linha de produtos utilizando MDA, que se pode comparar como os processos definidos em (GOMAA, 2004) para linha de produtos:

- **Engenharia de Domínio (Engenharia da Linha de Produto)** - envolve três principais atividades. A primeira atividade envolve a criação e gerenciamento do núcleo base, que consiste de características reutilizáveis ou variáveis. A segunda atividade envolve a especificação de um modelo de domínio, que consiste das características do domínio a nível conceitual. A terceira atividade envolve a definição da transformação, que especifica como as instâncias do modelo de domínio são mapeadas em uma aplicação utilizando o núcleo base. Está definição de transformação

poderia ser feita utilizando um compilador para uma linguagem específica de domínio ou em algum outro ambiente.

- **Engenharia da aplicação** - é criado um modelo de aplicação selecionando características alternativas do modelo de domínio. Neste processo características alternativas são selecionadas baseadas nos requisitos do produto. O modelo de aplicação é então compilado para aplicação utilizando a definição da transformação, o núcleo base e os requisitos do produto. Essencialmente, o processo de transformação liga a variabilidade em nível de infra-estrutura que não é visível no modelo de domínio, e os pontos de variação são selecionados a nível conceitual, para a realização desses pontos de variação na infra-estrutura.

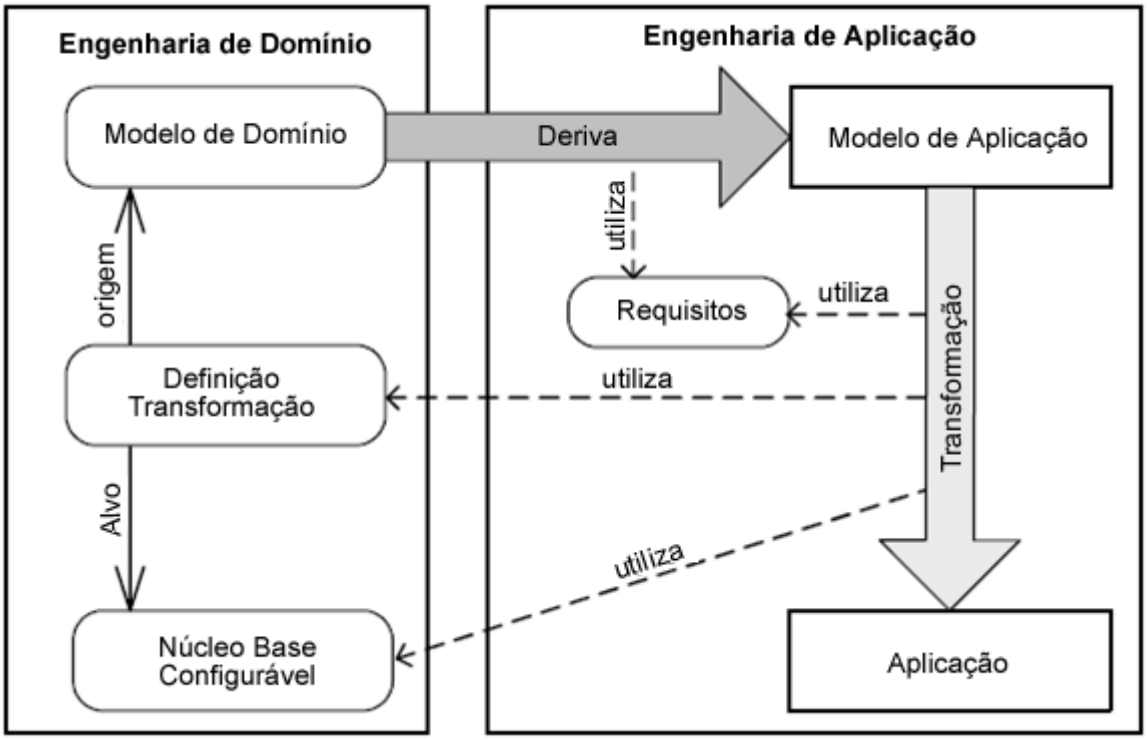


Figura 3.3: Engenharia de linha de produto utilizando a abordagem MDA (DEELSTRA *et al*., 2003)

3.7 Considerações Finais

Neste capítulo foram apresentados os principais fundamentos do modelo de linha de produto de software, o desenvolvimento utilizando a abordagem de linha de produto de software traz vários benefícios (SAGARDUY; BANDINELLI; LERCHUNDI, 2000), tais como, menor tempo de entrega do software, maior produtividade, e software com melhor qualidade. Técnicas como análise de domínio e modelagem de casos de uso apóiam o desenvolvimento utilizando linha de produtos, na identificação da similaridade e variabilidade da família de produtos. Contudo, deve-se destacar também alguns pontos que são muito discutidos sobre o assunto (DURSCKI *et al*., 2004), tais como, o risco da implantação de um desenvolvimento baseado em linha de produto de software onde existem diversas resistências nas empresas como resistência gerencial, organizacional, falta de recursos devidamente treinados.

Como foi apresentado, podem-se utilizar os conceitos da arquitetura orientada por modelos ao contexto de linha de produto de software a fim de prover uma arquitetura genérica para a linha de produtos bem como o desenvolvimento dos componentes variáveis de cada membro da linha de produtos. No capítulo seguinte será apresentado e detalhado um estudo de caso, utilizando os conceitos de MDA e linha de produto de software.

4 Estudo de Caso

Para uma melhor análise do uso da tecnologia MDA e SPL, optou-se por desenvolver um sistema no contexto de email *marketing*.

4.1 Contextualização

O email *marketing* é hoje uma poderosa ferramenta de relacionamento com o cliente e uma das mais utilizadas por organizações que se preocupam em manter um contato próximo com seu público. E a forma mais rápida e precisa de comunicação, pois é feita de forma totalmente pessoal e direcionada.

O email atinge seu público alvo individualmente, sendo assim um contato muito mais pessoal e direto se comparado com a mídia convencional.

A possibilidade de mensurar os resultados de uma campanha de email *marketing* também coloca está ferramenta em destaque, onde a tecnologia é usada como diferencial competitivo.

Os tipos de campanhas de email marketing são classificados de acordo com os seus objetivos. Existem basicamente três tipos de campanhas que utilizam diferentes modelos de mensagens, que são: *newsletter*, mala-direta eletrônica e pesquisa (ZORZI, 2008).

- *Newsletter* - este modelo de email caracteriza-se por possuir conteúdo informativo. O *newsletter* é utilizado para fornecer notícias, sobre a empresa que está enviando, associações que desejam informar seus associados, veículos de conteúdo jornalístico ou instituições de ensino provendo informações aos seus estudantes;
- Mala-direta Digital - este tipo de mensagem cumpre o papel de uma mala-direta convencional, porém na versão digital via email. Caracteriza-se por ser uma mensagem totalmente comercial para venda de um produto ou serviço. A grande vantagem da mala-direta eletrônica sobre a mala-direta convencional é a rapidez e o baixo custo, a possibilidade de mensuração de resultado por meio da visitação do site, que deve apresentar um aumento gradativo, assim como pela taxa de conversão (quantidade de compradores em relação à quantidade de visitantes), que também deve aumentar ao longo do tempo;

- Pesquisa - o email é uma maneira bastante eficiente de realizar pesquisas, sejam elas de satisfação ou de mercado. Existe, basicamente, duas maneiras de fazer o usuário responder a pesquisa. Uma das formas é o envio de formulário por email, na qual os dados que o destinatário preenche são enviados à empresa pesquisadora. Outra forma, mais segura, é o envio de um link que remete o usuário a uma página do site da empresa que está realizando a pesquisa. Está página conterá o formulário com as questões propostas;

O email *marketing* pode ser usado também como uma ferramenta de CRM (*Customer Relationship Management*) para criar ou manter um relacionamento com o cliente. Através de promoções, pesquisas de satisfação, informativos, a empresa cria um vínculo com o cliente, desde que lhe forneça conteúdo relevante.

4.2 Modelo de Domínio

Durante a especificação de requisitos, foi elaborado um modelo representando os objetos do domínio e identificando as semelhanças entre os produtos.

As funcionalidades básicas foram extraídas em análise de algumas aplicações existentes de email *marketing* (NITRONEWS, 2008) (MAILSENDER, 2008) (NESOX, 2008). O modelo elaborado não contém todos os pontos de variabilidade possíveis, pois o foco principal do estudo de caso será o desenvolvimento do núcleo comum da linha de produtos, para posteriormente aplicar os conceitos da abordagem MDA.

.

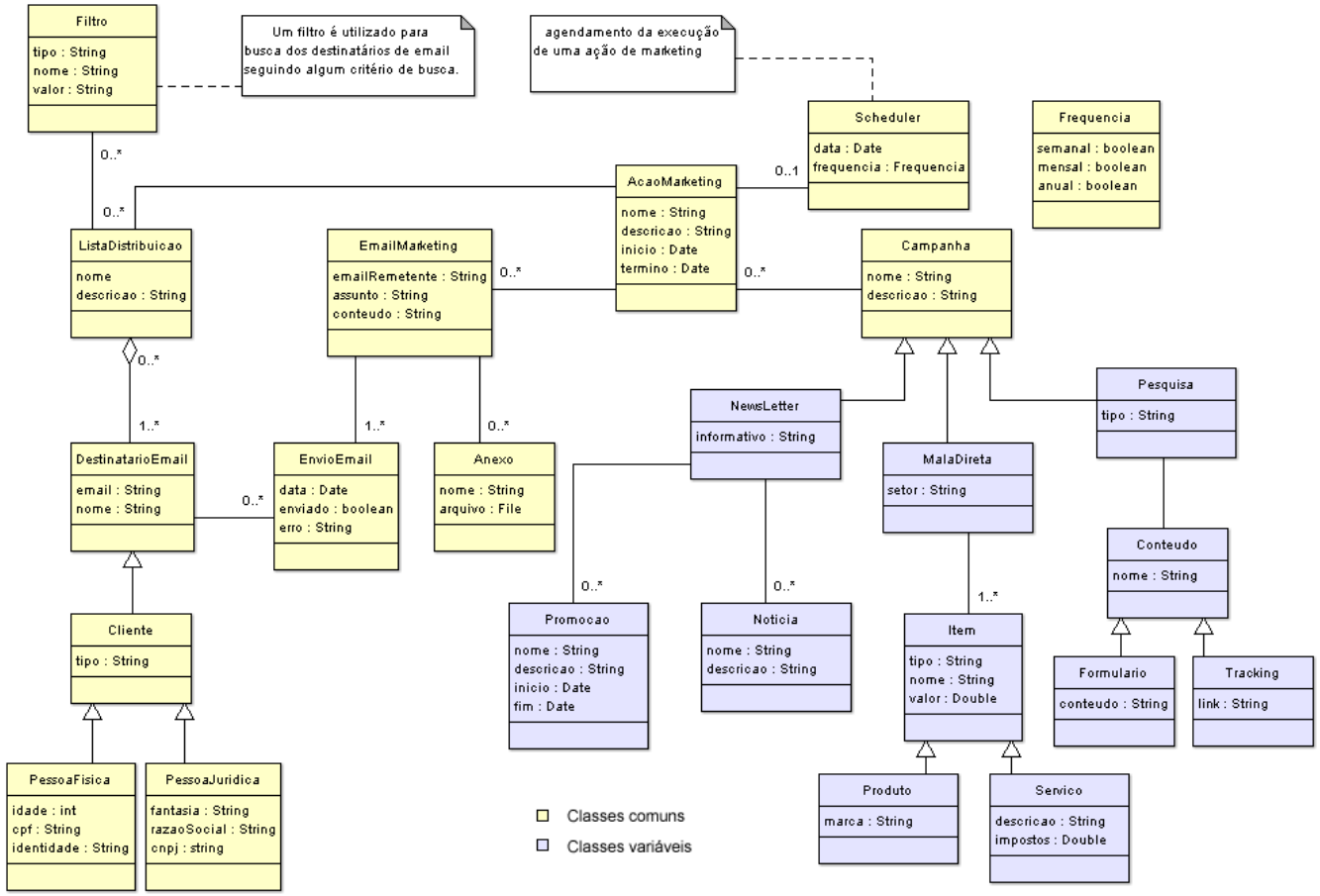


Figura 4.1: Modelo de domínio (CIM) email marketing

A Figura 4.1 demonstra um modelo de domínio no contexto de email *marketing*.

A seguir as classes do núcleo comum do modelo são descritas de forma sucinta.

Campanha – tipo de campanha de email marketing, como foi descrito anteriormente existem basicamente três tipos *newsletter*, mala-direta eletrônica e pesquisa;

AcaoMarkenting – ação de *marketing* que será realizada na campanha, por exemplo, envio de artigos da SBC (Sociedade Brasileira de Computação) para os alunos dos cursos de computação do país;

Scheduler – agenda a execução de uma ação de *marketing*;

Frequencia – freqüência que será executada atarefa agendada no *Scheduler*, podendo ser semanal, mensal ou anual;

EmailMarketing – email contendo as informações do email a ser enviado por uma ação de *marketing*, por exemplo, imagem, textos, links, formulários, etc.;

Anexo – arquivo anexo de um email marketing;

EnvioEmail – informações sobre o envio do email se foi realizado com sucesso, ou registrou algum erro durante o envio;

ListaDistribuicao – lista contendo os destinatários (clientes potenciais) que se deseja atingir com a ação de marketing;

Filtro – utilizado como critério de busca de um destinatário de email durante a geração de uma lista de distribuição;

DestinatarioEmail – cliente cadastrado na base de dados, que pode ser selecionado para receber emails de alguma ação de marketing;

Cliente – especialização do destinatário de email representa um cliente que pode ser uma pessoa física ou jurídica;

PessoaFisica – especialização de cliente em uma pessoa física;

PessoaJuridica – especialização de cliente em uma pessoa jurídica;

Para criar o modelo de domínio, foram utilizados os conceitos da engenharia de domínio da linha de produtos de software onde é definido o contexto da linha de produto e o núcleo dos artefatos como citado nesse trabalho seção 3.2, a fim definir uma arquitetura genérica para uma linha de produtos no contexto de uma aplicação de email *marketing*. Partindo da arquitetura genérica pode-se criar outros componentes reutilizáveis ou alternativos para os membros da linha de produtos, por exemplo, um componente para integração com outras aplicações onde os dados referentes aos destinatários de email poderiam, por exemplo, serem retornados de outras fontes de dados tais como: planilhas, documentos de texto, etc. Pode-se citar também alguns pontos de variação no modelo, tais como:

- A plataforma: que na abordagem utilizando MDA esse ponto de variação é realizado de maneira automática através das transformações entre os modelos onde pode-se gerar a mesma funcionalidade sobre plataformas diferentes.
- Em nível de desenvolvimento, por exemplo, pode-se citar a implementação da lista de distribuição que poderia ser obtida através da utilização de técnicas de *data mining* como a segmentação (AGRAWAL; IMIELINSKI; SWAMI, 1993), a fim de se obter grupos de clientes em potencial de maneira mais precisa.

Nesse estudo de caso, o foco principal é a definição de uma arquitetura genérica para a aplicação e posteriormente a transformação dos modelos seguindo os conceitos do *framework* MDA.

4.3 AndroMDA

Optou-se por utilizar o *framework* AndroMDA http://www.andromda.org, por apresentar, inicialmente, um melhor perfil a este trabalho, pois se adéqua as necessidades da tecnologia MDA, como apresentado em comparação realizada em MAIA, (2006).

Dentre as principais motivações para o uso do AndroMDA pode-se citar:

- Open-source;
- Suporte a modelos UML 1.4 e UML 2.0;
- Suporte a expressões em OCL;
- PSMs encapsulados na forma de "cartuchos", com destaque para:
    - BPM4Struts – geração dos artefatos para a camada de apresentação;
    - Hibernate – geração dos artefatos para a camada de persistência;
    - Spring - geração dos artefatos para a camada de integração com a camada de persistência;
    - WebService – geração de artefatos para acesso via *web services*;

O AndroMDA é um gerador de código que recebe um modelo UML como entrada, em formato XMI, e gera um código-fonte como saída (ANDROMDA, 2008). A geração de código é realizada através de uma série de *templates* configuráveis específicos para plataformas e linguagens de programação. Estes conjuntos de *templates* são denominados cartuchos (*cartridge*). A ferramenta AndroMDA pode ser integrada com alguns ambientes de modelagem, ou ainda importar, através de XMI, modelos armazenados em arquivos XML, representando, assim, uma abordagem independente de ferramenta CASE. Os cartuchos pré-definidos na abordagem são utilizados para gerar código-fonte na linguagem Java para a plataforma J2EE(SUN, 2008). No entanto, a extensão do AndroMDA é possível com a criação de novos cartuchos que agregam os novos *templates* para atender a necessidades específicas.

O funcionamento de uma ferramenta MDA típica Figura 4.2, no caso desse estudo de caso, o AndroMDA, envolve a criação dos modelos UML em uma ferramenta de modelagem,

que os exportam no formato XMI. A ferramenta MDA então, com base no XMI e nos *templates* de transformação, inicia a geração dos artefatos.

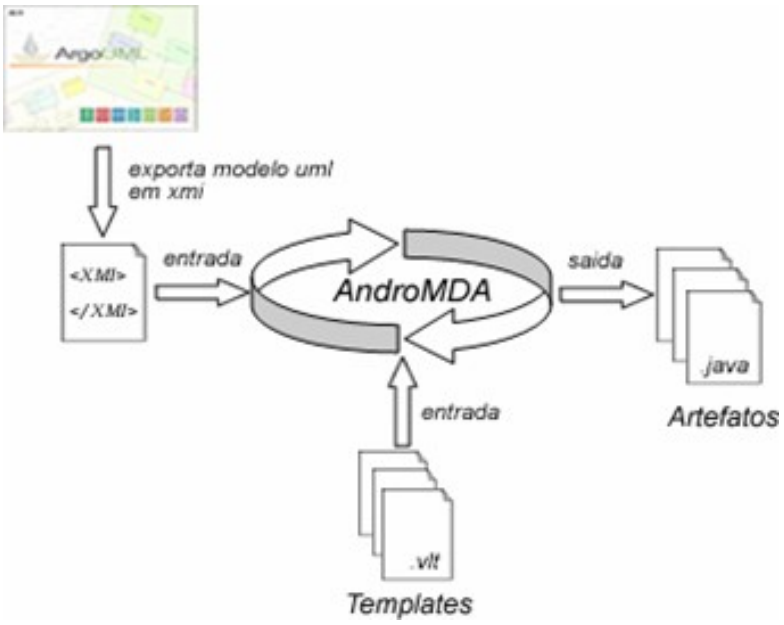


Figura 4.2: Funcionamento do AndroMDA (ANDROMDA, 2008)

O AndroMDA cria um estrutura de projeto básica para uma aplicação através de um *wizard*, onde são configurados os *templates* que serão utilizados durante a transformação, esse processo é executado através da integração com o Maven (APACHE, 2008b) e Ant (APACHE 2008a). A Figura 4.3 demonstra uma configuração de um projeto no AndroMDA através do *Shell Script* do Ant. Depois da geração do projeto através do *wizard*, o modelo UML do projeto é editado em uma ferramenta CASE na qual o modelo possa ser exportado no formato XMI, para posteriormente serem gerados os artefatos da aplicação através da integração com o Maven que executa comandos para iniciar os processos de construção da aplicação através das transformações do modelo UML.

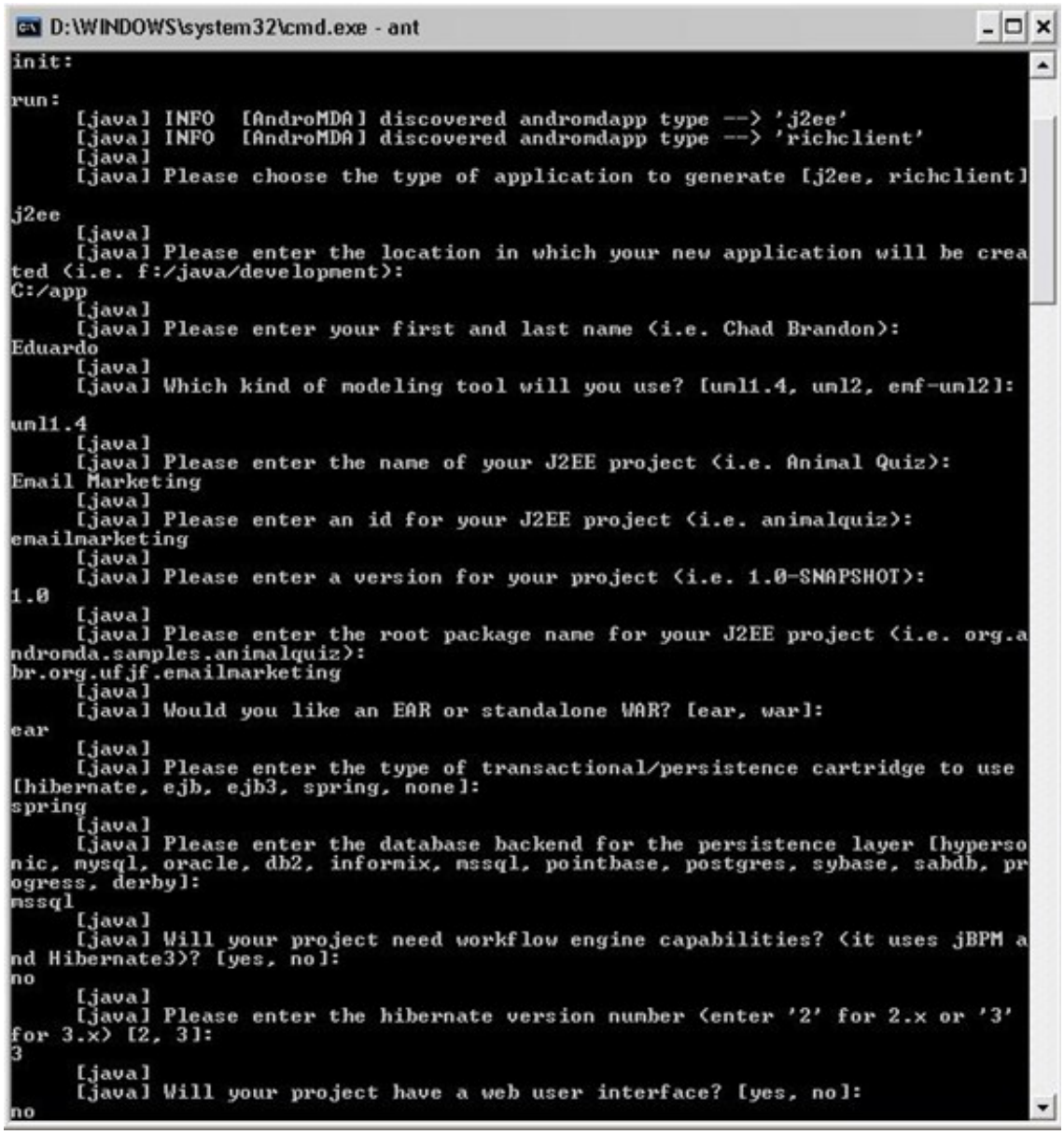


Figura 4.3: Configuração projeto AndroMDA

4.4 Processo de Desenvolvimento

Nesta seção detalha-se as fases que devem ser executadas durante o desenvolvimento seguindo a abordagem MDA utilizando a ferramenta AndroMDA.

A seguir são detalhadas as fases do processo de desenvolvimento de maneira sucinta:

1. Definido o modelo de domínio, este deve ser transformado no modelo independente de plataforma e modelado em uma ferramenta CASE;

2. Exportação do modelo independente de plataforma no formato XMI;

3. Configuração dos *templates* que serão utilizados no projeto durante o processo de transformação;

4. Inicializaroprocessodetransformaçãodomodeloconformeasconfiguraçõesdefinidas no projeto;

5. Realizar a implementação dos métodos que não são automatizados pela transformação;

6. Realização de Testes;

Esse processo pode ser executado de maneira iterativa e incremental durante o desenvolvimento da aplicação.

4.4.1 Modelo Independente de Plataforma

O modelo de domínio da aplicação de email marketing é transformado para o modelo independente de plataforma, para posteriormente realizar as transformações para obter o modelo especifico de plataforma e o código fonte. Essa transformação geralmente é feita manualmente, pois o modelo de domínio é um modelo a nível conceitual independente de computação, e não possui informações de plataforma. Nessa transformação classifica-se os objetos como entidades persistentes, e cria-se relações necessárias para que o modelo possa ser transformado para um modelo específico de plataforma. Essas relações são utilizadas para inserir informações da arquitetura da aplicação que, neste estudo de caso, será uma arquitetura em camadas que possui classes de serviço, classes de negócio, classes que são compartilhadas entre as camadas.

Para modelagem do modelo independente de plataforma Figura 4.4 foi utilizado o perfil UML definido pelo AndroMDA e a utilização do cartucho Spring (SPRING, 2008) do AndroMDA. O Spring é um *framework* que envolve as três camadas de uma arquitetura J2EE. No entanto, o cartucho Spring do AndroMDA, foca somente sobre a camada de persistência e a camada de negócio. O perfil UML do AndroMDA define um conjunto de estereótipos e *tagged values* que são utilizados pelos cartuchos do AndroMDA durante o processo de transformação.

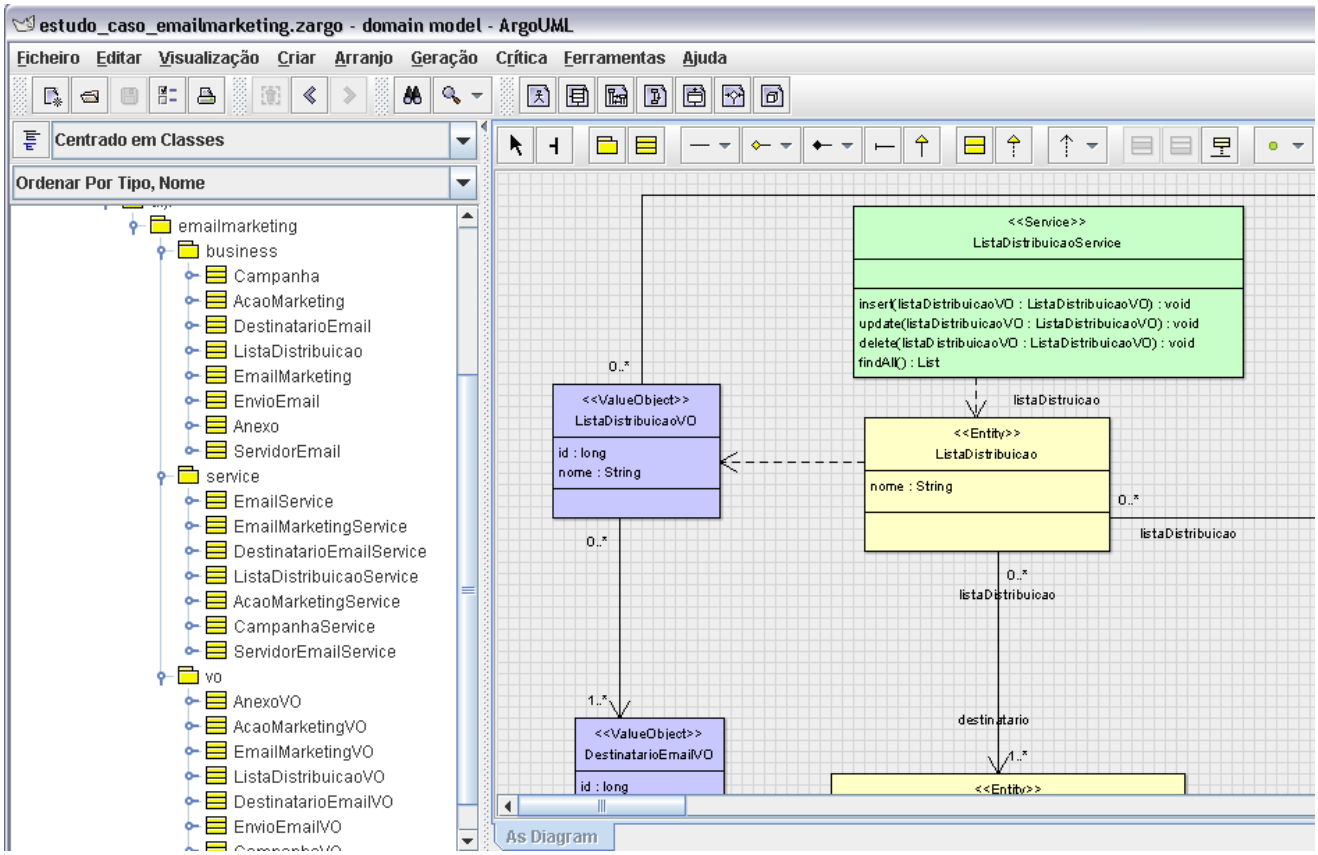


Figura 4.4: Modelagem do Modelo Independente de Plataforma

A Figura 4.5 demonstra uma parte do modelo independente de plataforma da aplicação de email *marketing* com os seus estereótipos inseridos(a especificação completa pode ser consultada no Apêndice A), que são utilizados durante o processo de transformação. O modelo possui o estereótipo <<*Entity*>> indica que representa um conceito de entidade do domínio do sistema, o estereótipo <<*Service*>> indica que a classe representa um serviço da aplicação, que utiliza de uma estratégia comum de desenvolvimento para separar a camada de apresentação da camada de persistência, e o estereótipo <<*Value Object*>> indica que a classe representa um objeto simples para a transferência de dados entre as camadas, essa estratégia foi utilizada, pois não é uma boa idéia expor sua camada de persistência na camada de apresentação diretamente, essa dependência poderia causar uma dificuldade na manutenção.

Durante o processo de transformação, esses estereótipos são utilizados, para determinar quais artefatos devem ser gerados pelos *templates*.

A arquitetura do modelo independente de plataforma é uma arquitetura em camadas como citado anteriormente, que possui basicamente três camadas principais *service*, *business*, *value object*.

- ***Service*** – realiza a comunicação com a camada de negócio e contém toda a lógica utilizada para realizar as operações nas camadas inferiores. Essa lógica de negócio é responsável pelo comportamento da aplicação, é inserida manualmente depois da transformação;
- ***Business*** – domínio da aplicação representa o problema propriamente dito, contém as classes (entidades) que descrevem o contexto da aplicação;
- ***Value Object*** – objetos que são transferidos entre as camadas, é a implementação de um padrão de projeto (GAMMA; HELM; JOHNSON, 2000) utilizado para melhorar o desempenho da rede;

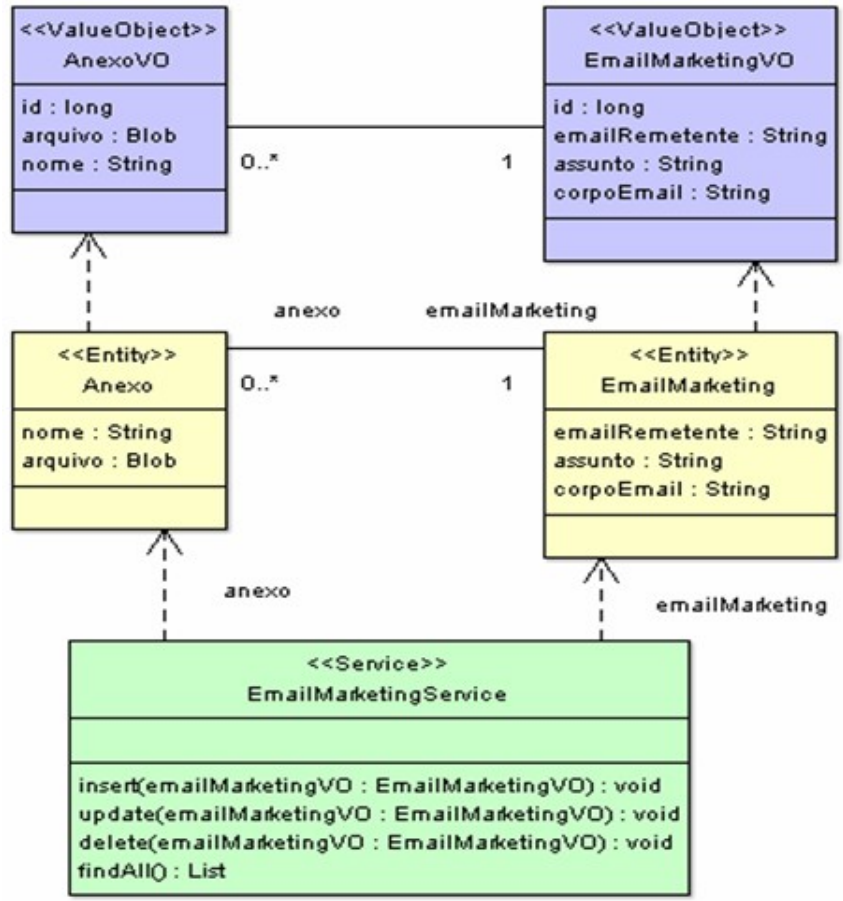


Figura 4.5: Parte do modelo independente de plataforma

4.4.2 Configurações do Projeto

Depois da modelagem do modelo independente de plataforma e sua exportação no formato XMI, realiza-se a configuração dos *templates* que serão utilizados no processo de transformação do modelo.

Como apresentado anteriormente, a configuração de AndroMDA é feita através de um *wizard* onde são configurados os parâmetros da aplicação, estes parâmetros determinam quais *templates* serão utilizados na transformação.

Nesse estudo de caso foi utilizada a seguinte configuração:

- Plataforma da aplicação – J2EE;
- Repositório – C:/app;
- Autor – Eduardo Barbosa da Costa;
- Versão UML – UML 1.4;
- Nome do projeto – Email Marketing;
- Identificador do projeto – emailmarketing;
- Versão do projeto – 1.0;
- Pacote principal – br.org.ufjf.emailmarketing;
- Tipo de projeto – .war (extensãodoempacotamentodoprojetoJ2EE);
- Base de dados – mysql;
- Possui compatibilidade com *workflow* – não;
- Versão Hibernate (*framework* de persistência) – 3.0;
- Possui interface com usuário – não;
- Publicação de funcionalidades via *web services* – não;

Após a configuração do projeto o AndroMDA é gerada uma estrutura básica para o projeto.

A estrutura da Figura 4.6 demonstra os módulos gerados pelo *wizard*:

- **mda** – é um dos módulos mais importantes, pois neste modulo o AndroMDA busca as informações necessárias para gerar os artefatos da aplicação. O modelo UML da aplicação no formato XMI deve ser inserido no diretório src/main/uml/;
- **common** – este módulo contém um conjunto de artefatos que são compartilhados por outros módulos da aplicação, por exemplo, contêm os *value objects*, as *interfaces* de serviço, classes para tratamento de exceção, etc.;

- **core** – este módulo contém o núcleo da aplicação, contém todas as classes da lógica de negócio da aplicação, por exemplo, contém a implementação das classes de serviço, classes de negócio, classes de integração, etc.;
- **web** – este módulo contém os artefatos da camada de apresentação da aplicação, tais como: *controllers*, páginas jsp, páginas HTML, etc.;
- **app** – contém os recursos e classes necessários para construir o empacotamento da aplicação para ser executada em um servidor de aplicação;

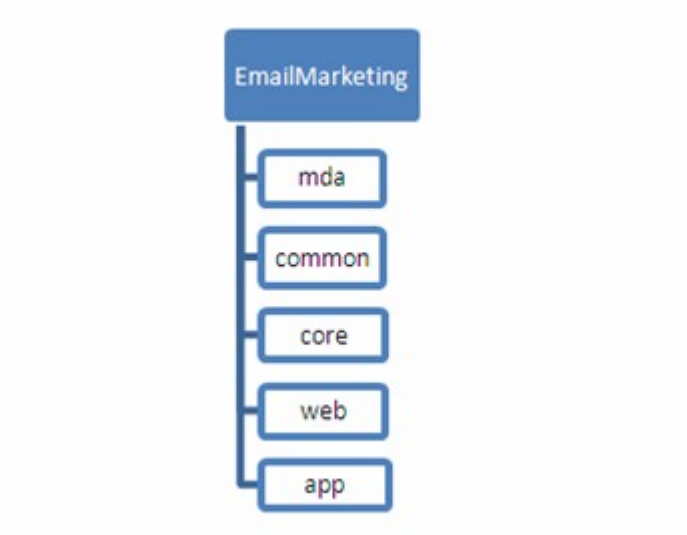


Figura 4.6: Estrutura do projeto gerada pelo AndroMDA

4.4.3 Transformação do Modelo

A partir do modelo independente de plataforma será gerado o código fonte, conforme a configuração da aplicação. O modelo marcado com os estereótipos e *tagged values* é inserido no modulo **mda** da aplicação, como descrito anteriormente, para posteriormente o AndroMDA gerar os artefatos da aplicação através dos processos de transformações. A seguir estão listados alguns dos artefatos gerados para a plataforma Java a partir do modelo parcial apresentado na Figura 4.5.

Anexo.hbm.xml	EmailMarketing.java
AnexoImpl.java	EmailMarketingDao.java
AnexoDaoBase.java	EmailMarketingDaoImpl.java
AnexoVO.java	EmailMarketingService.java

EmailMarketingServiceImpl.java
Anexo.java
AnexoDao.java
AnexoDaoImpl.java
EmailMarketing.hbm.xml
EmailMarketingImpl.java
EmailMarketingDaoBase.java
EmailMarketingVO.java
EmailMarketingServiceBase.java

A Figura 4.6 demonstra como é o processo de transformação no AndroMDA.

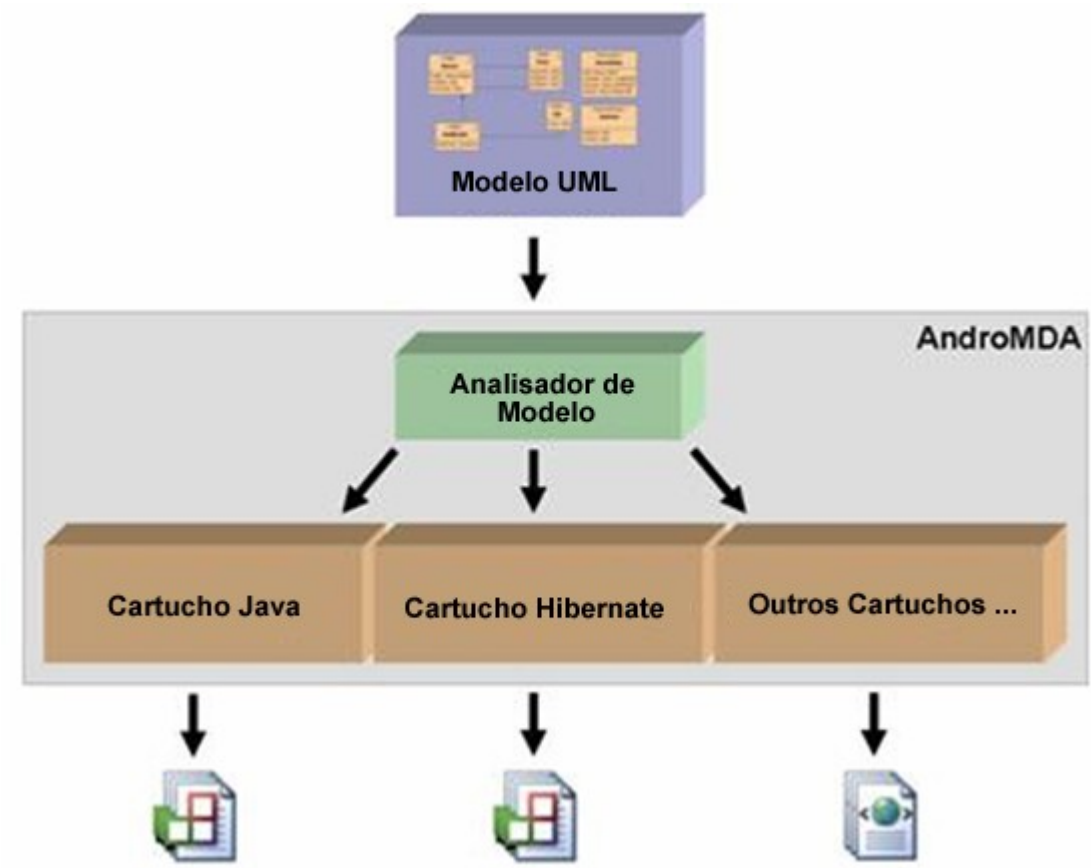


Figura 4.6: Processo de transformação do modelo UML utilizando o AndroMDA

4.4.4 Implementação de métodos não automatizados pela transformação

Após o processamento do AndroMDA serão gerados alguns artefatos que possuem características não implementadas automaticamente durante o processo de transformação, que devem ser implementadas manualmente utilizando alguma IDE da preferência do desenvolvedor. Um projeto criado pelo AndroMDA possui integração com a IDE Eclipse (IBM, 2008), logo a mesma foi utilizada nesse estudo de caso Figura 4.7.

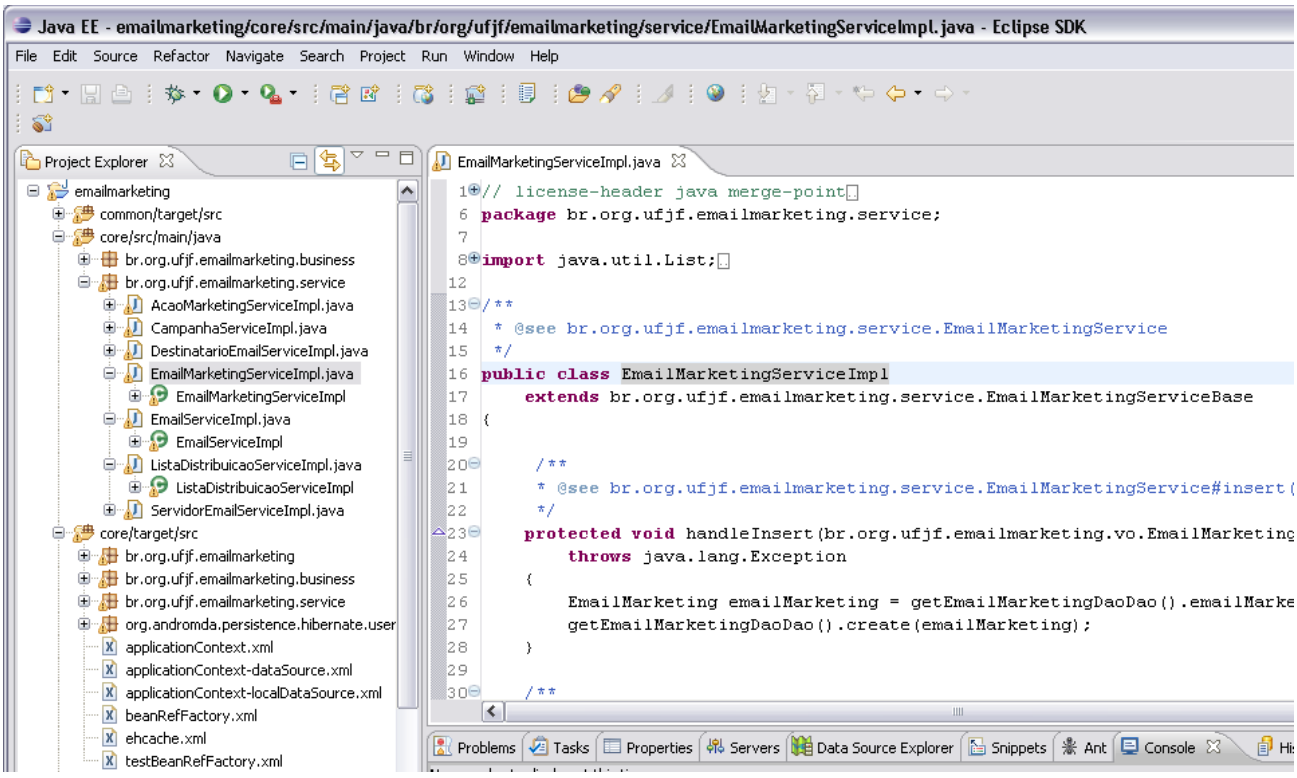


Figura 4.7: Customização do projeto na ferramenta Eclipse

A seguir na Listagem 4.1 é demonstrada a classe serviço gerada a partir do modelo parcial apresentado na Figura 4.5, seguida Listagem 4.2 de sua implementação onde são implementa dos os métodos de acesso à camada de persistência.

Listagem 4.1: Classe de Serviço gerada pela transformação

```
package br.org .ufjf .emailmarketing .service;
import java.util.List;
/**
* @see  br.org.ufjf.emailmarketing.service.EmailMarketingService
*/
public class EmailMarketingServiceImpl
        extends br.org .ufjf .emailmarketing .service.EmailMarketingServiceBase{
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#insert
(br.org.ufjf.emailmarketing.vo.EmailMarketingVO)
*/
protected void handleInsert(br.org .ufjf .emailmarketing .vo .
        EmailMarketingVO emailMarketingVO) throws java.lang.Exception{
// TODO Auto-generated method stub
}
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#update
(br.org.ufjf.emailmarketing.vo.EmailMarketingVO)
 */
protected void handleUpdate(br.org .ufjf .emailmarketing .vo .
        EmailMarketingVO emailMarketingVO) throws java.lang.Exception{
// TODO Auto-generated method stub
 }
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#delete
(br.org.ufjf.emailmarketing.vo.EmailMarketingVO)
*/
protected void handleDelete(br.org.ufjf.emailmarketing.vo.
        EmailMarketingVO emailMarketingVO) throws java.lang.Exception{
// TODO Auto-generated method stub
}
```

```
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#findAll()
*/
protected java.util.List handleFindAll()
        throws java .lang.Exception{
//  TODO Auto-generated method stub
return null ;
  }
}
```

Listagem 4.2: Implementação dos Métodos da classe de Serviço gerada pela transformação

---

```
package br.org .ufjf .emailmarketing .service;
import java.util.List;
import br.org .ufjf .emailmarketing .business.EmailMarketing;
import br.org .ufjf .emailmarketing .business.EmailMarketingDao;
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService
*/
public class EmailMarketingServiceImpl
        extends br.org .ufjf .emailmarketing .service.EmailMarketingServiceBase{
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#insert
(br.org.ufjf.emailmarketing.vo.EmailMarketingVO)
*/
protected void handleInsert(br.org.ufjf.emailmarketing.vo.
        EmailMarketingVO emailMarketingVO)
        throws java .lang.Exception{
EmailMarketing emailMarketing = getEmailMarketingDao() .
emailMarketingVOToEntity(emailMarketingVO);
getEmailMarketingDao() .create( emailMarketing);
}
```

```
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#update
(br.org.ufjf.emailmarketing.vo.EmailMarketingVO)
*/
protected void handleUpdate(br.org.ufjf.emailmarketing.vo.
        EmailMarketingVO emailMarketingVO)
        throws java .lang.Exception{
EmailMarketing emailMarketing = getEmailMarketingDao() .
emailMarketingVOToEntity(emailMarketingVO);
getEmailMarketingDao() .update(emailMarketing);
}
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#delete
(br.org.ufjf.emailmarketing.vo.EmailMarketingVO)
*/
protected void handleDelete(br.org .ufjf .emailmarketing .vo .
        EmailMarketingVO emailMarketingVO)
        throws java .lang.Exception{
EmailMarketing emailMarketing = getEmailMarketingDao() .
emailMarketingVOToEntity(emailMarketingVO);
getEmailMarketingDao().remove(emailMarketing);
}
/**
* @see br.org.ufjf.emailmarketing.service.EmailMarketingService#findAll()
*/
protected java.util.List handleFindAll()
        throws java .lang.Exception{
return (List) getEmailMarketingDao().loadAll(EmailMarketingDao .
TRANSFORMEMAILMARKETINGVO);
 }
}
```

Pode-se observar que todas as classes e métodos necessários para o acesso à camada de persistência foram gerados automaticamente através do processo de transformação do AndroMDA, sendo necessário apenas a implementação da lógica de negócio que envolve cada operação do serviço. Os artefatos para a camada de persistência são gerados utilizando o padrão de projeto DAO (*Data Access Object*) (GAMMA; HELM; JOHNSON, 2000).

4.4.5 Teste

Depois da implementação dos métodos não automatizados pelo processo transformação pode-se realizar os testes unitários na camada de serviço da aplicação para validar o seu comportamento. Os testes realizados nesta fase, não são gerados a partir do modelo, portanto, são definidos de maneira manual. Para a realização dos testes da aplicação foi utilizado o framework JUnit (SOURCEFORGE, 2008). A Listagem 4.3 demonstra a implementação do teste para a classe de serviço EmailMarketingService.

Listagem 4.3: Classe de Teste para o Serviço Email Marketing

```
package br.org .ufjf .emailmarketing .service.test;
import java.util.List;
import org .apache .commons .logging .Log;
import org .apache .commons .logging .LogFactory;
import org .junit .Before;
import org .junit .Test;
import br.org .ufjf .emailmarketing .ServiceLocator;
import br.org .ufjf .emailmarketing .service.EmailMarketingService;
import br.org .ufjf .emailmarketing .vo .EmailMarketingVO;

public class EmailMarketingServiceTest {
        private EmailMarketingService emailMarketingService;
        private Log logger = LogFactory.getLog(EmailMarketingServiceTest.class);

@Before
```

```
public void initializeTestSuite() {
// Initialize ServiceLocator
logger .info("Initializing ServiceLocator" );
ServiceLocator locator = ServiceLocator.instance();
locator .init("testBeanRefFactory.xml" , "beanRefFactory" );
// Initialize UserService
logger .info("Initializing EmailMarketingService");
emailMarketingService =  locator .getEmailMarketingService();
}
@Test
public void testFindAllEmailMarketing () {
        logger.info("findAllEmailsMarketing:");
                List<EmailMarketingVO> emails = emailMarketingService.findAll();
                for (EmailMarketingVO emailMarketingVO : emails) {
                        logger.info(emailMarketingVO.getAssunto());
                }
        }
}
```

4.5 Considerações Finais

Utilizando os conceitos da MDA no desenvolvimento da linha de produtos de software, pode-se elevar o nível de abstração de modo que os artefatos gerados na fase de análise e modelagem sejam partes ativa do desenvolvimento do produto final. A MDA apóia o desenvolvimento do núcleo comum da linha de produto bem como o desenvolvimento dos componentes variáveis e alternativos dos membros da linha de produto.

Com base nesses conceitos, foi possível demonstrar, através de um estudo de caso, que o processo de desenvolvimento software pode ser incrementado utilizando os conceitos da tecnologia MDA e SPL.

Nesse estudo de caso foi utilizado ferramenta AndroMDA que apóia o desenvolvimento utilizando a arquitetura orientada por modelos. O AndroMDA permite a construção de sistemas de maneira rápida e incremental, utilizando de boas práticas e padrões de projeto.

A utilização do AndroMDA foi fundamental no processo de desenvolvimento deste estudo de caso, pois permitiu demonstrar os principais conceitos envolvidos na arquitetura orientada por modelos apresentados nesse trabalho, tais como – a modelagem independente de plataforma, transformação entre modelos. O AndroMDA permite a construção de uma variedade de artefatos, alguns não explorados neste estudo de caso,por exemplo, a criação de *web services*, a criação do *front end* (camada de apresentação) da aplicação.

Entretanto existem alguns pontos que precisam ser considerados, por exemplo, o AndroMDA ainda não permite a transformação de modelo para modelo, pode-se citar também sobre a criação do modelo independente de plataforma que requer um esforço considerável e pessoal experiente.

5 Conclusão

5.1 Considerações Finais

Os conceitos da arquitetura orientada por modelos e linha de produto de software se mostram importantes no desenvolvimento de sistemas, pois oferecem uma maneira planejada e sistemática para a reutilização de software.

A arquitetura orientada por modelos é uma visão onde o modelo deixa de ser apenas um documento para ser parte central do processo de desenvolvimento de software. A partir de modelos abstratos pode-se transformá-los em modelos concretos de forma sucessiva até obter o código fonte da aplicação. Pode-se citar alguns benefícios da abordagem MDA para o desenvolvimento de software:

- **Produtividade** - com a definição de mapeamentos entre os modelos em diferentes níveis de abstração, o desenvolvimento da aplicação se torna um processo automatizado através das transformações, reduzindo o tempo de desenvolvimento;
- **Interoperabilidade** - as transformações realizadas entre os modelos permite implementar a mesma funcionalidade em diferentes plataformas através do processo de transformação;
- **Qualidade** - com a utilização dos conceitos da arquitetura orientada por modelos, pode-se focar o desenvolvimento do software na análise e modelagem, onde a aplicação final reflete os modelos. Através da introdução do desenvolvimento dos modelos, onde os modelos independentes de plataforma são transformados para a obtenção dos modelos específicos de plataforma, e então a implementação em uma plataforma particular de maneira automatizada. Reduzindo a disparidade existente no desenvolvimento de software atual entre os artefatos modelados e a aplicação final.

O desenvolvimento de linha de produtos software, por sua vez, deve ser considerado quando as organizações desenvolvem um conjunto de produtos que possuem alguma similaridade. Através do desenvolvimento de um núcleo comum compartilhado entre os produtos é possível reduzir consideravelmente o tempo de desenvolvimento do software. Na literatura existem vários exemplos bem sucedidos da utilização desses conceitos como em (CLEMENTS; NORTHROP, 2001).

Técnicas como análise de domínio e modelagem de casos de uso apóiam o desenvolvimento utilizando linha de produtos de software, na identificação da similaridade e variabilidade da família de produtos. Neste sentido, a arquitetura orientada por modelos pode ser inserida para incrementar o processo de desenvolvimento de software utilizando essa abordagem.

Para apoiar o desenvolvimento de aplicações envolvendo os conceitos apresentados neste trabalho, existem tanto ferramentas livres como AndroMDA, quanto ferramentas comerciais como Compuware, OptimalJ e ArcStyler, onde pode-se explorar estes conceitos.

5.2 Trabalhos Futuros

O desenvolvimento de software utilizando a arquitetura orientada por modelos e linha de produtos de software envolve ainda alguns conceitos não abordados nesse trabalho.

Um processo para o gerenciamento da variabilidade da linha de produtos de software está disponível em (DEELSTRA *et al*., 2003). Métodos de mapeamento, modelagem e transformação dos modelos podem ser encontrados em (FRANKEL, 2003). A utilização de uma notação gráfica para a definição das transformações de forma visual está disponível em (WILLINK, 2003). Um estudo de outros métodos e abordagens para o desenvolvimento de software utilizando a arquitetura orientada por modelos permite maior compreensão das propostas apresentadas.

Referências

AGRAWAL, R.; IMIELINSKI, T.; SWAMI, A. *Database mining*: *A performance perspective*. IEEE *Transactions on Knowledge and Data Engineering*, 1993.

ANDROMDA. *AndroMDA framework*. http://www.andromda.org, 2008. Ultimo acesso em maio/2008.

APACHE. *Ant framework*. http://ant.apache.org, 2008. Ultimo acesso em maio/2008.

APACHE. *Maven framework*. http://maven.apache.org, 2008. Ultimo acesso em maio/2008.

ATKINSON, C.; NORTHROP, L. *Component-Based Product-Line Engineering with* UML.[S.l.]: Addison-Wesley, 2001.

BACHMANN, F. et al. *Technical concepts of Component-Based Software Engineering*. [S.l.], 2000.

BAYER, J.; FLEGE, O. Pulse: *A methodology to develop software product lines*. *Symposium o Software Reusability*, p. 122 a 131, 1999.

BOSCH,J. *Design and use of software architectures: adopting and evolving a product-line approach*.[S.l.]: Addison-Wesley, 2000.

CLEMENTS, P.; NORTHROP, L. *Software product lines: practices and patterns*.[S.l.]: Addison-Wesley, 2001.

COHEN, S. *Product Line State of the Practice Report* (CMU/SEI-2002-TN-017).[S.l.], 2002.

DEELSTRA, S. et al. *Model driven architecture as approach to manage variability in software product families*. In: UNIVERSITY OF TWENTE. *Proceedings Model Driven Architecture*: *Foundations and Applications*. [S.l.], 2003. p. 109 a 114.

DONEGAN, P. M.; MASIERO, P. C. Geração de Famílias de Produtos de Software com Arquitetura Baseada em Componentes. Dissertação (Mestrado) — Universidade de São Paulo(USP), 2007.

DURSCKI, R. C. *et al*. Linhas de produto de software: riscos e vantagens de sua implantação.VI Simpósio Internacional de Melhoria de Processos de Software, 2004.

FRANKEL, D. S. *Model Driven Architecture: Applying* MDA to *Enterprise Computing*. 1st. ed.[S.l.]: JohnWiley andSons,2003.

GAMMA, E.; HELM, R.; JOHNSON, R. Padrões de Projeto.2st. ed.[S.l.]: Bookman Companhia Editora, 2000.

GOMAA, H. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*.1st. ed.[S.l.]: AddisonWesley,2004.

GUTTMAN, M.; PARODI, J. *Real-Life* MDA: *Solving Business Problems with Model Driven Architecture*.1st. ed.[S.l.]: Elsevier,2007.

IBM. Eclipse. http://www.eclipse.org/documentation/, 2008. Ultimo acesso em março/2008.

KANG, K. *Feature-Oriented Domain Analysis* (FODA) *Feasibility Study* (CMU/SEI-90TR-21, ADA 235785).[S.l.], 1990.

MAIA, N. ODYSSEY-MDA: Uma Abordagem Para a Transformação de Modelos. Dissertação (Mestrado) - COPPE/UFRJ, Rio de Janeiro,2006.

MAILSENDER. MailSender. http://www.mailsender.com.br, 2008. Ultimo acesso em junho/2008.

MELLOR, S. J.; BALCER, M. J. *Executable* UML: *A Foundation for Model-Driven Architecture*.1st. ed.[S.l.]: Addison-Wesley,2002.

MELLOR, S. J. et al. MDA *Distilled Principles of Model-Driven Architecture*. 1st. ed. [S.l.]: Addison-Wesley, 2004.

NADA, N. et al. *Product line viewpoint and validation models. In: Proceedings of Product Lines: Economics, Architectures, and Implications*.[S.l.: s.n.], 2000. p. 141 a 148.

NESOX. Nesox. http://www.nesox.com, 2008. Ultimo acesso em junho/2008.

NITRONEWS. Nitronews. http://www.nitronews.com.br, 2008. Ultimo acesso em junho/2008.

OMG. CORBA *Component Model*, Version 3.0.[S.l.], 2002.

OMG. *Common Warehouse Metamodel* (CWM) *Specification*, version1.1.[S.l.], 2003.

OMG. MDA *Guide, version* 1.0.[S.l.], 2003.

OMG. Revised submission for MOF2.0 *Query/Views/Transformations* RFP.[S.l.], 2004.

OMG. *Meta Object Facility* (MOF) *Core Specification*, OMG *Available Specification*, version 2.0.[S.l.], 2006.

OMG. *Object Constraint Language*, version 2.0.[S.l.], 2006.

OMG. XML *Metadata Interchange Specification*, version 2.0.1.[S.l.], 2006.

OMG. *Unified Modeling Language: Infrastructure*. OMG Specification, version 2.1.2. [S.l.], 2007.

OMG. *Unified Modeling Language: Superstructure*. OMG Specification, version 2.1.2. [S.l.], 2007.

REINEHR, S. dos S. *et al*. Linhas de produto de software: Tornando realidade o reuso sistematizado de software. projeto MILPS, 2006.

SAGARDUY, G.; BANDINELLI, S.; LERCHUNDI, R. *Product-line analysis. In: Proceedings of Product Lines: Economics, Architectures, and Implications*. [S.l.: s.n.], 2000.p.23 a26.

SEI. *A framework for software product line practice* 4.2. http://www.sei.cmu.edu/plp/framework.html, 2008. Ultimo acesso em março/2008.

SOURCEFORGE. *JUnit framework*. http://junit.sourceforge.net/, 2008. Ultimo acesso em maio/2008.

SPRING. *Spring framework*. http://static.springframework.org/spring/docs/2.0.x/reference, 2008. Ultimo acesso em março/2008.

SUN. JSR40: Java *Metadata Interface* (JMI) *Specification* -version1.0.[S.l.], 2001.

SUN. The J2EE 1.4 Tutorial. http://java.sun.com/j2ee/1.4/docs/tutorial/doc, 2008. Ultimo acesso em maio/2008.

WARMER, J.; KLEPPE, A.; BAST, W. MDA *Explained: The Model Driven Architecture: Practice and Promise*.1st. ed.[S.l.]: Addison-Wesley,2003.

WEISS, D.; CHITAU, R. L. *Software product-line engineering: a family-based software development process*.[S.l.]: Addison-Wesley, 1999.

WFMC. *Workflow Management Coalition. Workflow Reference* Model.[S.l.], 1995.

WILLINK, E. D. Umlx: *A graphical transformation language for mdas*. In: UNIVERSITY OF TWENTE. *Proceedings Model Driven Architecture: Foundations and Applications*. [S.l.], 2003. p. 13 a 24.

ZORZI, E. Introdução Email Marketing. http://www.nitronews.com.br/emailmarketing.php, 2008. Ultimo acesso em maio/2008.

Apêndice A – Especificação do PIM