FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO "EURÍPIDES DE MARÍLIA" – UNIVEM
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO

THIAGO ISHIO

Uma Experiência no Desenvolvimento de Sistemas utilizando Arquitetura Orientada por Modelos

MARÍLIA
2006

THIAGO ISHIO

Uma Experiência no Desenvolvimento de Sistemas utilizando Arquitetura Orientada por Modelos

Monografia apresentada ao Curso de Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha" de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Profº. Dr. Valter Vieira de Camargo
Co-Orientadora:
Profª. Dra. Maria Istela Cagnin

MARÍLIA
2006

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharelado em Ciência da Computação.

Nota:_____(_____)

Orientador: Valter Vieira de Camargo _____

1º. Examinador: Edmundo Sérgio Spoto _____

2º. Examinador: Ana Paula Piovesan Melchiori Peruzza _____

Marília, 05 de dezembro de 2006.

AGRADECIMENTOS

Agradeço primeiramente a Deus pela minha vida e pela capacidade de enfrentar as dificuldades que encontramos ao longo do tempo.

Agradeço aos meus pais Kunimasa Ishio e Hanako Yamashita Ishio pela compreensão e auxílio em todas as etapas da vida.

Agradeço ao meu filho Rafael Ishio, parceira e amigos pelas horas de ausência para a realização deste trabalho.

Agradeço ao orientador Valter Vieira de Camargo e a co-orientadora Maria Istela Cagnin por me orientarem com muita paciência, dedicação e compreensão. Agradeço também por todas as sugestões e questionamentos pertinentes que auxiliaram o andamento deste trabalho.

ISHIO, Thiago**. Uma Experiência no Desenvolvimento de Sistemas utilizando Arquitetura Orientada por Modelos.** 2006. 67f.
Dissertação (Graduação em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

RESUMO

Este trabalho propõe o estudo dos conceitos relacionados ao paradigma de desenvolvimento orientado por modelos (MDA - *Model Driven Architecture*), bem como um estudo de caso que utiliza a ferramenta *Enterprise Architect* (EA). O objetivo do trabalho é mostrar como utilizar esse paradigma, evidenciar suas principais características e destacar vantagens e desvantagens em relação a um processo de desenvolvimento convencional. O estudo de caso desenvolvido é um sistema para consultório dentário e consiste basicamente no cadastro de clientes, controle de consultas e de pagamentos.

Palavras-Chaves: Model Driven Architecture, UML, Orientado a Objetos, MDA.

ISHIO, Thiago**. Uma Experiência no Desenvolvimento de Sistemas utilizando Arquitetura Orientada por Modelos.** 2006. 67f.
Dissertação (Graduação em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha, Marília, 2006.

ABSTRACT

In this work is presented the study of the concepts related to MDA paradigm (Model Driven Architecture), as well as a case study that uses the Enterprise Architect (EA) tool. The aim is to show how to use this paradigm making evident its main characteristics and to show advantages and disadvantages against a conventional development process. A case study was conducted in a sense of register customers and control consultations and payments.

Keywords: Model Driven Architecture, UML,Object Oriented, MDA.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

CASE	*Computer Aided Software Engineering*
CORBA	*Common Object Request Broker Architecture*
CWM	*Common Warehouse Metamodel*
EJB	*Enterprise Java Beans*
MDA	*Model Driven Architecture*
MOF	*Meta Object Facility*
MOLAP	*Multidimenssional On-Line Analytical Processing*
OLAP	*On-line Analytical Processing*
OO	*Oriented Object*
OOSE	*Object Oriented Software Engineering*
OOPSLA	*Conference on Object-Oriented Programming System, Languages and Applications*
OMG	*Object Management Group*
OMT	*Object Modeling Technique*
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
UML	*Unified Object Facility*

INTRODUÇÃO

Há muito tempo as organizações que desenvolvem softwares vem reconhecendo a importância de se aproveitar o esforço empregado no desenvolvimento de sistemas para reduzir prazo, recursos e aumentar a qualidade e a produtividade das equipes (MELLOR *et al.,* 2005; FOWLER *et al.*, 2005).

Um grande desafio que as equipes enfrentam é a escolha das tecnologias a serem empregadas no desenvolvimento. O problema é que artefatos desenvolvidos com um forte vínculo tecnológico podem apresentar uma redução no seu potencial de reutilização, pois seu tempo de vida está associado ao tempo de vida da plataforma escolhida. Além disso, a incompatibilidade entre as plataformas limita o potencial de reutilização apenas às aplicações desenvolvidas na mesma plataforma (FOWLER *et al., 2005)*.

Visando a diminuir esses problemas, a *Object Management Group* (OMG), uma organização internacional, desenvolveu uma arquitetura conhecida como *Model Driven Architecture* (MDA*)* (MELLOR *et al.*, 2005). Essa arquitetura tem como objetivo escolher uma plataforma para o sistema e realizar uma transformação sobre um modelo inicial, a fim de obter uma especificação ou uma codificação automática do sistema na plataforma escolhida. Com o uso dessa abordagem, os desenvolvedores passam a dar maior importância à modelagem dos requisitos de negócio da aplicação, não se preocupando com a plataforma onde serão implementados esses requisitos por alguma ferramenta que utiliza o paradigma MDA.

O MDA é totalmente desenvolvido no paradigma Orientado a Modelos, um dos atuais desafios no desenvolvimento de software, na tentativa de elevar o nível de abstração no desenvolvimento para além das atuais linguagens de programação (MDA, 2002).

Os principais objetivos do MDA são a portabilidade, a interoperabilidade e a reusabilidade. Para atingir tais objetivos, o MDA baseia-se em três padrões: a UML (*Unified Modeling Language*) (FOWLER *et al.*, 2005), MOF (*Meta Object Facility*) (DSTC, 2001) e CWM (*Common Warehouse Metamodel*) (CHANG, 2001).

Este trabalho, então é feito um estudo deste novo paradigma, estudando seus conceitos, a fim de observar se o MDA apóia os objetivos de portabilidade, interoperabilidade e a reusabilidade, e evidenciar as principais vantagens e desvantagens.

MOTIVAÇÃO

O MDA ainda é um conceito de arquitetura recente. A motivação para a monografia é o pouco conhecimento do paradigma MDA por partes de desenvolvedores. Em conseqüência desse pouco conhecimento, poucos desenvolvedores e organizações adotam esse novo paradigma com receio de que ela não apresente os benefícios prometidos. Uma segunda motivação para este trabalho é cada vez mais aprender e ganhar experiência com esse novo paradigma de desenvolvimento

OBJETIVOS

Este trabalho tem por objetivo fazer um estudo da metodologia de desenvolvimento de software baseada no MDA, observando as vantagens que a Arquitetura Orientada a Modelos pode trazer em relação aos processos tradicionais de desenvolvimento. Para isto é feito tanto um estudo teórico quanto um estudo prático desta tecnologia. Neste último, um sistema de pequeno porte é desenvolvido, utilizando ferramentas de apoio desta tecnologia.

ORGANIZAÇÃO DA MONOGRAFIA

O primeiro capítulo da monografia fornece uma compreensão do contexto do trabalho e sua importância, a motivação, objetivos e a organização do trabalho desenvolvido.

No segundo capítulo são apresentados os conceitos necessários para se entender o *framework* MDA, suas principais formas de construção, conceitos sobre os padrões UML (*Unified Modeling Language*), MOF (*Meta Object Facility*) e CWM (*Common Warehouse Metamodel*), processos de desenvolvimento baseado em MDA e um breve estudo sobre ferramentas que apóiam o MDA.

No terceiro capítulo é apresentado o estudo de caso proposto para a análise do MDA.

No quarto capítulo apresenta-se a conclusão, trazendo as desvantagens e vantagens no uso do paradigma MDA.

2. ARQUITETURA ORIENTADA POR MODELO (MDA)

O MDA é um método muito novo em relação ao desenvolvimento de software, possui métodos e procedimentos a fim de auxiliar os desenvolvedores atuais. Possibilita em um desenvolvimento mais rápido, com menor custo e com melhor qualidade.

Este capítulo apresenta parte da revisão bibliográfica, precisamente no que se diz respeito aos conceitos em MDA. Na Seção 2.2 são apresentados os principais conceitos para se entender MDA, apresentando também os principais padrões no qual são baseados, como o UML, MOF e CWM. Na Seção 2.3 são apresentados os conceitos dos Processos de MDA. Na Seção 2.4 são apresentadas algumas ferramentas usadas atualmente com base ao MDA.

2.1. Conceitos de MDA

A maioria dos sistemas de informação é desenvolvida especificamente para uma determinada plataforma, no entanto cada vez mais se faz necessário a integração de sistemas para promover soluções mais complexas, a migração de sistemas para outra plataforma ou até mesmo uma mudança da tecnologia usada para uma mais nova e mais eficaz (FOWLER *et al.*, 2005, MELLOR *et al.,* 2005).

Em 2000, a *Object Management Group* (OMG), inspirada na importância dos modelos em processos de desenvolvimento de software e na sua missão de prover soluções para o problema da interoperabilidade entre os sistemas, criou o *framework* MDA (*Model-Driven Architecture*) (MELLOR *et al.*, 2005), totalmente dirigido a modelos. O MDA utiliza modelos formais, ou seja, que possam ser entendidos e interpretados por computadores. Esse *framework* define e especifica alguns modelos, como eles devem ser usados e o relacionamento entre eles durante todas as fases do ciclo de vida de um sistema.

O MDA se baseia em 3 padrões criados pela OMG (MDA, 2002):

1. UML (*Unified Modeling Language*) (UML; OMG) é uma linguagem gráfica padrão para elaboração da estrutura de projetos de *software.* A UML pode ser empregada para visualizar, especificar, construir e documentar os artefatos de sistemas de *software* (FOWLER *et al.*, 2005).

2. MOF (*Meta Object Facility*) (HEATON, 2005) é uma tecnologia adotada pela OMG (*Object Management Group*), usadas nos ambientes CORBA (conjunto de serviços que várias aplicações podem compartilhar) para gerência de meta-informação (é uma informação

da informação). O MOF tem como objetivo o seu uso numa larga variedade de cenários, desde gerência de tipos até desenvolvimento de *software*, gerência de informação e *data warehouse*.

3. CWM (*Common Warehouse Metamodel*) (HEATON, 2004) é um padrão de metadados (dados que fazem referência a outros dados) que permite integrar sistemas de *data warehouse, e-business* e sistemas de negócios inteligentes em ambientes heterogêneos e distribuídos. O padrão CWM foi criado pelo grupo OMG (*Object Management Group),* baseados nos padrões MOF, CWM e XML (CHANG, 2001).

A técnica de especificação de sistemas definida pela MDA separa a especificação da funcionalidade do sistema da implementação de uma funcionalidade em uma plataforma de tecnologia específica. Os principais artefatos produzidos durante o processo de desenvolvimento baseado em MDA são os modelos e o código fonte (MUKERJI E MILLER, 2006).

De acordo com Mellor *et al.* (2005, p.15), modelos são conjuntos de elementos que descrevem alguma realidade física, abstrata ou hipotética. Um modelo é sempre capturado por um metamodelo particular, como por exemplo, modelos que utilizam a linguagem UML são sempre capturados pelo metamodelo UML e descrevem como os modelos UML são estruturados, os elementos que eles contêm e as propriedades de uma plataforma particular. Os metamodelos são simplesmente modelos das linguagens de modelagem, ele define a estrutura, a semântica e as restrições de grupos de modelos que compartilham sintaxe e semântica comuns.

Em MDA, um modelo é uma representação da parte de uma função, da estrutura ou do comportamento de um sistema. A especificação de um modelo é dita formal quando é baseada em uma linguagem que tenha sintaxe, semântica e regras de análise (MELLOR, 2005).

A sintaxe pode ser gráfica ou contextual, a semântica é definida nos termos dos objetos observados no mundo que está sendo descrito (por exemplo, as mensagens enviadas e as respostas, as mudanças de estados de um objeto, etc). No MDA, uma especificação que não seja formal neste sentido, não é um modelo. Um modelo de MDA precisa ser representado segundo uma linguagem bem definida, que tenha sintaxe e semântica da linguagem que está modelando, bem definidos para que seja adequado para a interpretação automática por computador. Uma especificação baseada em UML é um modelo cujas propriedades possam ser expressas graficamente, por meio dos diagramas, ou textualmente, por meio do XML (*Extensible Markup Language*) (MUKERJI, 2006).

Inicialmente desenvolve-se um modelo independente de plataforma PIM (*Platform Independent Model*), em que todas as funcionalidades do sistema e restrições de negócio são modeladas. Esse modelo torna-se mais fácil de ser validado, pois se abstrai o detalhe semântico dependente de plataforma. O próximo passo após a definição do PIM é transformá-lo em um modelo específico de plataforma, chamado PSM (*Platform Specific Model*), em que leva-se em consideração como cada plataforma pode implementar melhor cada funcionalidade desejada e especificada no PIM. Com o PSM pronto a geração de código pode ser realizada. A transformação do PIM para PSM garante a preservação do conhecimento do sistema e, conseqüentemente, sua portabilidade. A idéia é que essas transformações de PIM para PSM e de PSM para código fonte se tornem cada vez mais automáticas, aumentando assim os ganhos na produtividade (MUKERJI E MILLER, 2003).

Para o desenvolvimento de um padrão MDA, são assim reforçados os três passos seguintes (MUKERJI E MILLER, 2003; MELLOR *et al., 2*005):

Passo 1: Primeiro, deve-se construir o modelo com um alto nível de abstração, isto é, independente de qualquer tecnologia (PIM). A utilização de um PIM possui várias vantagens:

- É um processo de tradução simplificado.
- O desenvolvedor/arquiteto mantém o foco na lógica de negócio sem ter que se preocupar com os detalhes inerentes a implementação da aplicação.
- O PIM pode ser reaproveitado, pois ele não é limitado a nenhuma plataforma existente.

Passo 2: Depois de construído o Modelo Independente de Plataforma, o PIM é transformado em um ou mais modelo específico de plataforma (PSM). Um PSM é feito para especificar o sistema em termos de implementação que é feita em uma tecnologia específica, como por exemplo, um modelo de banco de dados ou um modelo de EJB (*Enterprise Java Beans*).

Passo 3-Final: O último passo é transformar o PSM em código. Como o PSM possui uma plataforma específica, esta transformação é feita de forma trivial. O passo complexo é o que transforma PIM em PSM.

O processo de transformação de um modelo PIM para o modelo PSM de um mesmo sistema está representado na Figura 2.1. Primeiramente se implementa um PIM ou PIM Marcado, o PIM Marcado é um modelo de marcação que define os nomes, tipos e valores default para cada marca (entrada adicional exigida para o mapeamento), após o PIM modelado, é transformado no modelo PSM e posteriormente gerado o código fonte.

Atualmente a linguagem específica de modelagem é a UML. A existência desse padrão produz uma linguagem bem definida que reduz a probabilidade de má interpretação dos modelos por meio dos vizualizadores (MELLOR *et al.*, 2005).

Uma plataforma deve ter um conjunto de tecnologias e subsistemas que fornecem uma especificação da funcionalidade, é definida como um ambiente de execução para um conjunto de modelos (MUKERJI E MILLER, 2003). Os exemplos de plataformas incluem a plataforma Java, CORBA, .NET, sistemas operacionais como Linux, Solaris e Windows, e plataformas específicas em tempo real.

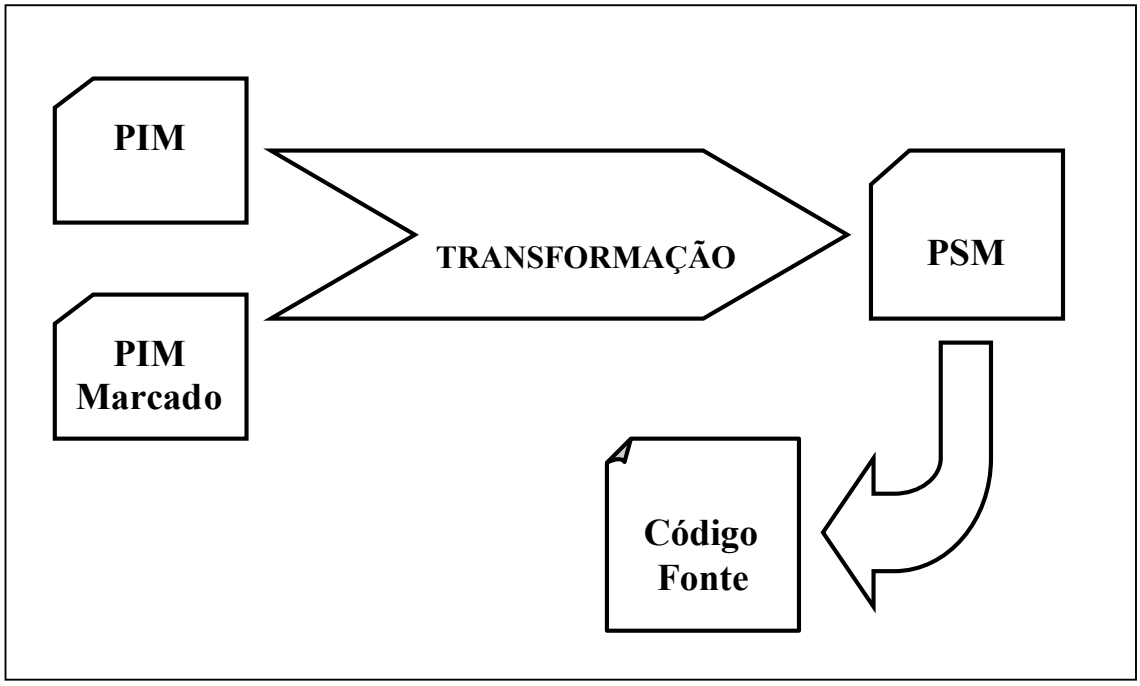

Figura 2.1. Processo de transformação PIM, PSM, Código fonte.

2.2. PIM (*Platform Independent Model)*

Todo projeto de desenvolvimento MDA inicia-se com a criação de um Modelo Independente de Plataforma (PIM - *Platform Independent Model*), expresso em UML. O PIM possui um alto grau de abstração e deve permitir representar o sistema de modo independente de qualquer tecnologia de Implementação. O PIM é um modelo declarativo formal da estrutura funcional do sistema (MUKERJI E MILLER, 2003), ou seja, tem foco na operação do sistema e, por isso, deve oferecer a melhor descrição possível para suportar o sistema em questão, mas sem considerar qual plataforma será utilizada ou como tal sistema será

construído, ou seja, é neutro tecnologicamente. É um modelo destinado a preservar a informação essencial a respeito do projeto da aplicação, sua arquitetura e infra-estrutura (BOOCH, 2004).

O PIM também constitui um modelo bastante rico em termos semânticos, pois pode ser representado gráfica ou textualmente em termos da UML e suas extensões (tais como a OCL - *Object Constraint Language* - que facilita a indicação de restrições), dando ao projetista, meios de expressar mais precisamente suas intenções e, com isso, reduzindo o trabalho das etapas posteriores. Outro aspecto importante é que o PIM poderá ser armazenado em um repositório através da MOF, possibilitando sua recuperação e processamentos posteriores.

Devido ao PIM possuir independência, o mesmo PIM pode ser utilizado para um conjunto de diferentes plataformas do mesmo tipo. Um exemplo disso seria um PIM construído para uma plataforma de computação distribuída (MELLOR *et al*,. 2005; MUKERJI E MILLER, 2003).

2.3. PSM (*Platform Specifc Model)*

Um Modelo Específico de Plataforma (PSM - *Platform Specific Model*) é gerado a partir do PIM. Este modelo, também expresso através da UML, deve representar o sistema em termos de construções apropriadas de uma tecnologia particular, ou seja, considerando a operação descrita pelo PIM e também detalhes específicos da implementação em termos da tecnologia selecionada. Um PSM combina as especificações do PIM com detalhes que descrevem como o sistema usa um tipo particular de plataforma. Um PIM pode ser utilizado para a geração de vários PSMs (MELLOR *et al*,. 2005; MUKERJI E MILLER, 2003). Isto significa que uma determinada plataforma deverá ser escolhida, implicando na seleção de mecanismo de mapeamento particular, para a transformação de um artefato PIM em um PSM.

Conforme Mukerji e Miller (2003), a transformação de um PIM em um PSM CORBA exigiria a escolha de elementos específicos definidos num UML *profile*, no qual as classes UML representariam as interfaces, tipos e outras construções do CORBA através de estereótipos (*Stereotypes*).

A operação de transformação de um PIM em um PSM é o passo crucial do processo de desenvolvimento MDA, pois representa o maior ganho oferecido, dado que é relativamente

comum que um mesmo sistema tenha que operar em diferentes plataformas, evitando a repetição dos esforços de desenvolvimento (MUKERJI E MILLER, 2003).

2.4. Transformações ou Mapeamentos

Um grande diferencial do MDA reside na forma de realização das transformações entre os modelos PIM, PSM e o código. Tradicionalmente as transformações entre modelos de um processo de *software* são realizadas manualmente. Diferentemente no MDA, os modelos deverão ser usados para geração automática da maior parte do sistema. No MDA todas as transformações devem ser realizadas automaticamente por ferramentas apropriadas, o que pode significar maior rapidez e flexibilidade na geração de aplicações de melhor qualidade, caracterizando assim os benefícios imediatos de sua aplicação (FOWLER *et al.*, 2005).

As transformações entre modelos ou mapeamento (*mappings*) são entendidas como o conjunto de regras e técnicas aplicadas em um modelo de modo que seja obtido um outro com as características desejadas. A MDA considera a existência de quatro tipos de transformações diferentes (MILLER; MUKERJI, 2001):

- PIM para PIM. Utilizada para o aperfeiçoamento ou simplificação dos modelos sem a necessidade de levar em conta aspectos dependentes de plataforma.
- PIM para PSM. Transformação “padrão” do modelo independente de plataforma para outro específico durante o ciclo de desenvolvimento típico de aplicações.
- PSM para PSM. Esta transformação permite a migração da solução entre plataformas diferentes, bem como o direcionamento de partes da implementação para tecnologias específicas, usadas por questões de interoperabilidade ou benefícios obtidos através do uso de certas plataformas.
- PSM para PIM. Quando é necessário obter-se uma representação neutra em termos de tecnologia de soluções específicas.

Um mapeamento entre modelos necessita de um ou um conjunto de modelos como entrada chamado “origens” e produz um modelo de saída chamado “destino”. O mapeamento possui regras, são regras descritas em nível de metamodelos (um modelo que define uma linguagem por expressar um modelo) e aplicáveis a todos os conjuntos de modelos de origem que obedecem ao metamodelo dado. Há casos, que há necessidade de atributos diferentes, dispõe-se de entradas de mapeamentos adicionais. Essas entradas recebem uma forma

chamada “marcas”, que são extensões para modelos e capturam informações exigidas para a transformação de modelos (MELLOR *et al.,* 2005).

As marcas podem ser usadas em duas instâncias, como entradas adicionais, usadas para antecipar decisões de projeto e como saídas, usadas como registro do processo de transformação de um modelo origem para um modelo destino. As marcas são definidas por um modelo que descreve a estrutura e a semântica de um conjunto de tipos de marcas. Uma função de mapeamento pode usar vários modelos de marcação para um único metamodelo de origem, tornando as marcas correspondentes reusáveis para diferentes mapeamentos correspondentes (MELLOR *et al.*, 2005).

Um modelo pronto pode ser modificado, supostamente pode-se adicionar código ao modelo, ou seja, editar um modelo gerado. Isso é a idéia de “elaboração de modelos”, uma vantagem do *framework* MDA, que possibilita modelos destinos serem desenvolvidos livremente no paradigma orientado a modelos. Os modelos destinos são criados e recriados a partir das funções de mapeamento, mas nem sempre se consegue produzir modelos completos ou atender todos os processos não funcionais de um desenvolvimento de *software* (MELLOR *et al.*, 2005).

Os principais objetivos do MDA são:

- Portabilidade. A portabilidade se encontra no modelo PIM. O modelo PIM pode ser transformado em vários PSMs de diversas plataformas, ou seja, toda a informação especificada o PIM é completamente portável.
- Interoperabilidade. A interoperabilidade que o MDA se refere é a comunicação entre os PSMs.
- Reusabilidade. No MDA, com a ajuda das ferramentas MDA, alterações feitas no PSM podem ser refletidas no seu PIM de origem, mantendo PIM, PSM e documentação consistentes até o término do projeto, o que não ocorre entre os modelos durante um processo tradicional de desenvolvimento de software.

2.5. UML (*Unified Modeling Language*)

Os estudos sobre a tecnologia de objetos iniciaram-se na década de 1980 com ênfase nas linguagens de programação. No final da mesma década começaram a surgir os métodos de análise e projeto. Todos os métodos eram muito similares, apesar da existência de diferentes notações para representar o mesmo conceito, o que na verdade, causava muita confusão entre

os técnicos e competição entre os metodologistas, o que provocou a “guerra dos métodos” (MELLOR *et al.,* 2005; FOWLER *et al.*, 2005; QUATRANI, 2003).

Em 1994 James Rumbaugh e Grady Booch iniciaram o trabalho de unificação dos métodos Booch e OMT (*Object Modeling Technique*) (QUATRANI, 2003).

O método de Grady Booch para orientação a objetos definiu que o sistema é analisado como uma série de visões diferentes, em que cada uma delas é descrita por uma série de modelos de diagramas. O método de Booch era bastante extenso, e alguns usuários achavam alguns símbolos muito difíceis de serem desenhados manualmente. O método também continha um processo pelo qual o sistema era analisado em macro e micro visões de desenvolvimento, baseado num processo interativo altamente incrementativo (BOOCH *et al,*. 2000).

A Técnica de Modelagens de Objetos (OMT) é uma método voltado para o teste de modelos, baseando-se em etapas de análise, projeto de sistema, projeto de Objetos e implementação (RUMBAUGHT, 1994).

Na análise se constrói um modelo de uma situação do mundo real, esta análise mostra propriedades relevantes, por exemplo, o que o sistema deverá fazer. Um modelo de análise deve ser bem compreendido pelos programadores, mas sem aplicações de programação. No projeto do Sistema toma-se decisões sobre a arquitetura do sistema, decidir quais características devem ser otimizadas. No Projeto dos Objetos é construído um modelo de projeto baseado no modelo de análise e contendo detalhes de implementação. Na implementação os modelos desenvolvidos durante o projeto dos objetos são implementados em uma linguagem de programação (RUMBAUGHT, 1994).

Em 1995 no OOPSLA (*Conference ob Object-Oriented Programming System, Languages and Applications*) , Booch e Rumbaught apresentaram a documentação da Versão 0.8 do método unificado (QUATRANI, 2003).

Durante 1996 eles trabalharam no método que passou a chamar de *Unified Modeling Language* (UML) e a *Object Management Group* (OMG) iniciou um esforço para padronização na área de métodos. Os esforços de Rumbaugh, Booch e Jacobson resultaram as versões 0.9 e 0.91 da UML em junho e outubro de 1996, respectivamente (QUATRANI, 2003).

A UML é a sucessora da linguagem de modelagem encontrada nos métodos Booch, OOSE/Jacobson, OMT e outros como Modelos de Entidades e Relacionamentos (BELL, 2003).

Cada um desses métodos era reconhecido mediante alguma forte característica. O OOSE (*Object-Oriented Software Engineering*) é orientado a Casos de Uso (*Use-Case*) e provê um excelente recurso para a análise de requisitos. O OMT foi expressivo para análise e sistemas centrados a dados. Em 1993 Booch foi relevante na fase de projeto e construção de sistemas voltados para Engenharia (QUATRANI, 2003).

Durante 1996 a *Rational Software Corporation* contou com a participação de outras instituições parceiras como: *Hewlett-Packard*, IBM, Microsoft, Oracle, Unisys, Platinum Technology, etc. Essa colaboração contribuiu para a produção da versão 1.0, uma versão bem definida, expressiva, poderosa e aplicável, a qual foi submetida a OMG para adoção(QUATRANI, 2003).

Em setembro de 1997 liberaram a Versão1.1 da UML e em dezembro a mesma foi aprovada como padrão pela OMG(UML, OMG).

- **Conceitos de UML**

UML (*Unified Modeling Language*) é um conjunto de notações gráficas, que ajuda na descrição e no projeto de sistemas de software, principalmente aqueles construídos no paradigma Orientado a Objetos(FOWLER, 2005).

De acordo com Fowler(2005), orientação a objeto é um conceito que está relacionado com a idéia de classificar, organizar e abstrair coisas. Em uma definição formal, "orientação a objetos significa organizar o mundo real como uma coleção de objetos que incorporam estruturas de dados e um conjunto de operações que manipulam estes dados", esses dados formam as classes. Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades ou atributos, o objeto terá. As classes possuem dois elementos importantes, a estrutura e o comportamento. A estrutura representa os atributos que descrevem a classe, o comportamento representa os serviços que a classe suporta.

Existem alguns conceitos básicos que estão vinculados ao conceito de orientação a objetos, **herança**, **encapsulamento** e **polimorfismo**(FOWLER, 2005).

A **herança** permite implementar a funcionalidade de uma classe, emprestando a estrutura e comportamento de classes de nível mais alto. É um mecanismo que permite que características comuns a diversas classes sejam agrupadas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas. Cada classe derivada ou

subclasse apresenta as características (estrutura e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela (BOOCH *et al*., 2000).

O **encapsulament**o “oculta informações”, ele define que cada objeto contém todos os detalhes de implementação necessários sobre como ele funciona e oculta os detalhes internos sobre como ele executa os serviços (BOOCH *et al*., 2000).

O **Polimorfismo** significa muitas formas, na orientação a objetos você pode enviar uma mesma mensagem para diferentes objetos e fazê-los responder da maneira correta. Você pode enviar a mensagem de dar marcha-ré para cada objeto semelhante a um carro e cada um vai se comportar de maneira diferente para atender a sua solicitação (BOOCH *et al*., 2000).

A UML é um padrão relativamente aberto, controlado pelo OMG (*Object Management Group*), um consórcio aberto de empresas. O OMG foi formado para estabelecer padrões que suportassem interoperabilidade, especificamente a de sistemas orientados a objetos(BOOCH *et al*., 2000).

De acordo com Fowler(2005), para criar o software de uma aplicação é necessário uma descrição do problema e dos seus requisitos. A análise, não é mais nada do que uma investigação do problema, e de como a solução é definida. Para desenvolver uma aplicação também é necessário ter descrições de alto nível e descrições detalhadas da solução lógica e de como ela atende os requisitos e as restrições. O projeto enfatiza uma solução lógica, ou seja, como o sistema atende os requisitos.

A essência da análise e do projeto orientado a objetos é enfatizar a consideração de um domínio de problema e uma solução lógica, segundo a perspectiva de objetos (coisas, acontecimentos ou entidades). Durante a análise orientada a objetos, há uma ênfase na descoberta e na descrição dos objetos ou conceitos do domínio do problema. Durante o projeto orientado a objetos existe uma ênfase na definição de elementos lógicos de software, os quais, em última instância, serão implementados em uma linguagem de programação orientado a objetos (LARMAN, 2002).

À medida que se trabalha com a UML, a programação fica cada vez mais mecânica, precisando ser automatizada. Muita das ferramentas *CASE* (*Computer Aided Software Engineering*) realizam alguma geração de código, o que automatiza a construção de uma parte significativa de um sistema, mas seus recursos são extremamente limitados e não permitem ao desenvolvedor uma geração de código de uma forma mais refinada, mas também chegando a um ponto em que todo o sistema pode ser especificado na UML, chegando a uma linguagem de programação. Nesse ambiente, os desenvolvedores desenham diagramas UML que são

compilados diretamente para código executável e a UML se torna código fonte (FOWLER *et al.*, 2005).

Geralmente as pessoas vêem a UML como sendo apenas os diagramas. Entretanto, os criadores da UML vêem os diagramas como secundários, a essência da UML é o metamodelo. Os diagramas são simplesmente apresentações do metamodelo. Os metamodelos são simplesmente modelos das linguagens de modelagem. Ele define a estrutura, a semântica e as restrições de grupos de modelos que compartilham sintaxe e semântica comuns (FOWLER, 2005).

Segundo Booch et al (2000), a linguagem UML possui diagramas diferentes para representar os aspectos estruturais, comportamentais e físicos de um *software.* São eles:

- **Diagrama de Classes**. Esses diagramas mostram as diferentes classes que compõem um sistema e como elas se relacionam umas com as outras. Os Diagramas de Classes são apontados normalmente como “estáticos” porque mostram as classes, em conjunto com os seus métodos e atributos.
- **Diagrama de Objetos**. O diagrama de objetos é uma variação do diagrama de classes e utiliza quase a mesma notação. A diferença é que o diagrama de objetos mostra os objetos que foram instanciados das classes. Diagramas de objetos também são usados como parte dos diagramas de colaboração. Exibe um conjunto de objetos e seus relacionamentos. São diagramas que abrangem a visão estática da estrutura ou do processo de um sistema, como ocorre nos diagramas de classes, mas sob perspectiva de casos reais de protótipos.
- **Diagrama de Casos de Uso**. Os Diagramas de Caso de Uso especificam o comportamento de um sistema ou de parte de um sistema e é uma descrição de um conjunto de seqüências de ações e seus relacionamentos. Abrangem a visão estática de casos de uso do sistema.
- **Diagrama de Interação.** Um diagrama de Interação exibe uma interação, consistindo de um conjunto de objetos e seus relacionamentos, incluindo as mensagens que podem ser trocadas entre eles. São utilizados para fazer a modelagem dos aspectos dinâmicos do sistema, envolvendo a modelagem de instâncias concretas ou protótipos de classes, interfaces e componentes juntamente com as mensagens que são trocadas por eles, em uma linha de tempo, mostrando os seus comportamentos.

- **Diagrama de Seqüências (Diagrama de Interação).** Um diagrama de seqüência mostra a colaboração dinâmica entre os vários objetos de um sistema. O mais importante aspecto deste diagrama é que a partir dele percebe-se a seqüência de mensagens enviadas entre os objetos. Ele mostra a interação entre os objetos, alguma coisa que acontecerá em um ponto específico da execução do sistema. O diagrama de seqüência consiste em um número de objetos mostrado em linhas verticais. O decorrer do tempo é visualizado observando-se o diagrama no sentido vertical de cima para baixo. As mensagens enviadas por cada objeto são simbolizadas por setas entre os objetos que se relacionam. Abrangem a visão dinâmica de um sistema.
- **Diagrama de Colaborações (Diagrama de Interação).** Um diagrama de colaboração mostra de maneira semelhante ao diagrama de seqüência, a colaboração dinâmica entre os objetos. Normalmente pode-se escolher entre utilizar o diagrama de colaboração ou o diagrama de seqüência. São usados na modelagem dos aspectos dinâmicos de um sistema.
- **Diagrama de Máquinas de Estados.** Este diagrama enfatiza o comportamento de um objeto ordenado por eventos. Os Diagramas de Máquinas Estados são empregados na modelagem dos aspectos dinâmicos de um sistema.
- **Diagrama de Atividades**. Os Diagramas de Atividades é um tipo especial de diagrama de Máquina de Estado, que exibem o fluxo de uma atividade para outra em pontos diferentes do fluxo de controle. São empregados na modelagem no aspecto dinâmico do sistema, envolvendo modelagem das etapas seqüenciais em um processo computacional.
- **Diagrama de Componentes.** Um Diagrama de componentes exibe as organizações e as dependências que existem em um conjunto de componentes diagramas de componentes. Os diagramas de componentes são empregados para a modelagem da visão estática de implementação de um sistema. Envolve a modelagem de itens físicos que residem em um nó, como executáveis, bibliotecas, tabelas, arquivos e documentos.
- **Diagrama de Implantação**. De acordo com Booch et al(2000), um Diagrama de Implantação exibe a configuração dos nós de processamento em tempo de execução e os componentes neles existentes. São empregados para a modelagem

> da visão estática da implantação de um sistema, geralmente envolve a modelagem da topologia do hardware em que o sistema é executado.

No MDA os diagramas que mais se usam são os diagramas de classes, os diagramas de seqüência, os diagramas de colaboração, usados para o desenvolvimento de modelos, metamodelos e os diagramas de Máquina de estados para o desenvolvimento da UML executável (MELLOR *et al.,* 2005).

A abordagem multi-visão da UML fornece aos desenvolvedores os benefícios da separação de áreas de interesse durante a modelagem de um software. Os diferentes diagramas servem a diferentes propósitos durante o processo de desenvolvimento (FOWLER, 2005).

A UML também modelos executáveis, modelos que possuem todos os requisitos necessários para produzir a funcionalidade desejada de um único domínio. Os modelos executáveis atuam exatamente como o código, fornece a habilidade de interagir diretamente com o domínio do cliente e é compilado por um compilador de modelos. São também independentes de plataforma de software, tornando-os dinâmicos em vários ambientes de desenvolvimento (MELLOR *et al.*, 2005).

Esses modelos executáveis são expressos pela UML executável. A UML executável é a parte da UML que define uma semântica de execuções para um subconjunto da UML. O subconjunto é computacionalmente completo, tornando-o diretamente executável. Todos os diagramas da UML são projeções de um modelo de semântica subjacente. Alguns modelos UML que não suportam execuções como do diagramas de casos de uso, podem ser usados para ajudar na construção de modelos UML executáveis (MELLOR *et al.*, 2005).

A UML é um importante padrão usado no *framework* MDA, pois é usado para a criação, edição e manutenção no desenvolvimento de modelos e metamodelos, mapeamentos e funções de mapeamentos. A UML também trabalha juntamente com o MOF (*Meta Object Facility*), para escrever funções de mapeamentos no qual se torna impossível trabalharem sem corporativismo. Existem duas razões para definir uma linguagem dentro da UML. A primeira é a comunicação entre os membros da equipe, no qual pode-se debater, por exemplo, a inclusão de tags em um determinado modelo, tornando todas as decisões em uma política de notações. A segunda é a comunicação com as máquinas, pois a UML define formalmente as linguagens e permitem as transformações entre modelos expressos nessas linguagens (BOOCH *et al.,* 2000; FOWLER *et al.,* 2004; MELLOR *et al.*, 2005).

- **Diagrama de Classes**

De acordo com Booch *et al.*(2000), o Diagrama de Classes é o diagrama mais comumente encontrado em modelagens de sistema Orientado a Objeto (OO). Ele é composto por classes, interfaces e seus relacionamentos. As classes contêm métodos e atributos e abrangem uma visão estática da estrutura do sistema. Graficamente, um diagrama de classes é uma coleção de vértices e arcos.

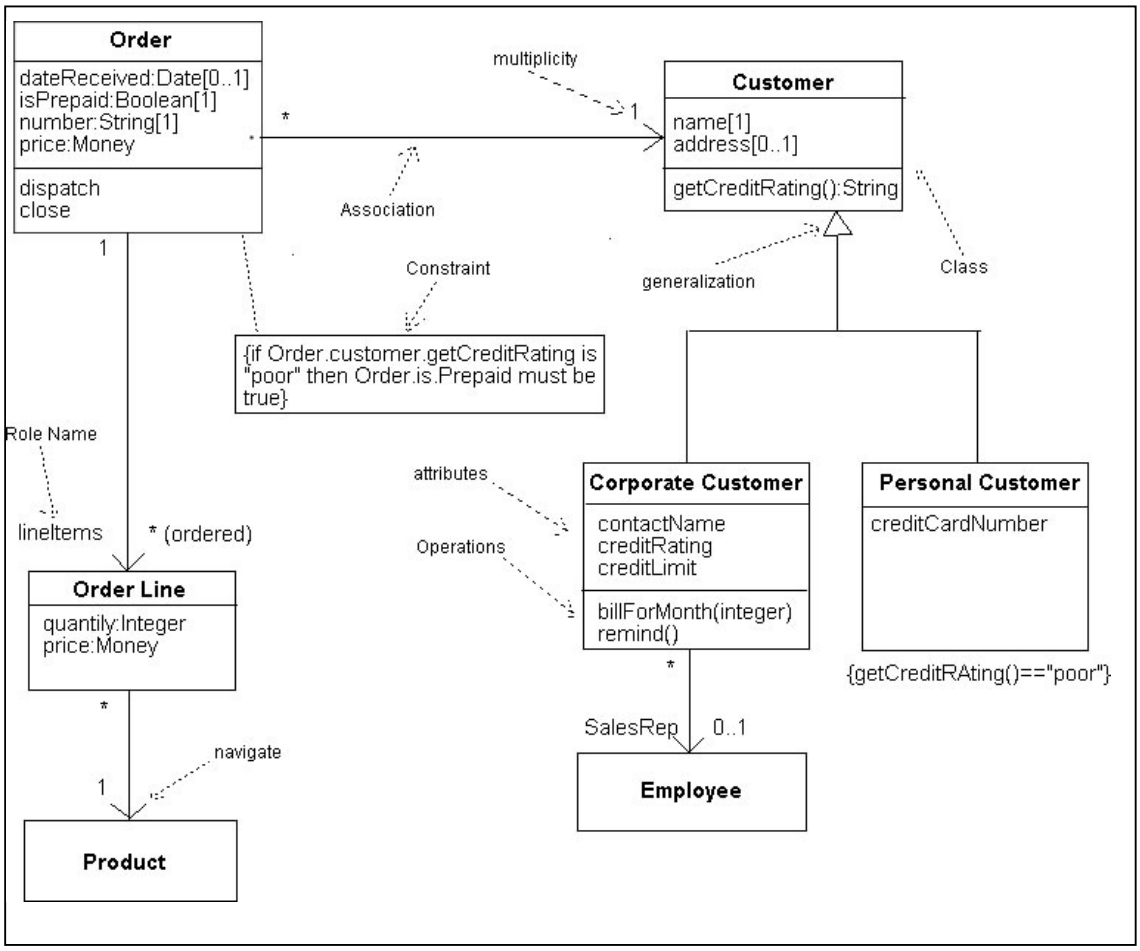

Figura 2.2 Um simples Diagrama de Classes (FOWLER *et al.*, 2004)

Os Diagramas de Classes são usados para fazer a modelagem da visão estática do projeto de um sistema. Na maioria dos casos, envolve a modelagem do vocabulário do sistema, a modelagem de colaborações ou a modelagem de esquemas. É também a base para um par de diagramas relacionados: os diagramas de componentes e is diagramas de implantação (BOOCH *et al.*, 2000).

Os diagramas de classes são importantes não só para a visualização, a especificação e a documentação de modelos estruturais, mas também para a construção de sistemas executáveis por intermédio de engenharia de produção e reversa (BOOCH *et al.*, 2000).

Com a UML, utilizam-se os diagramas de classes para visualizar os aspectos estáticos, seus relacionamentos e para especificar detalhes da construção, conforme mostra a Figura 2.2.

- **Diagrama de Seqüência (Diagrama de Interação)**

De acordo com Booch *et al*. (2000), todos os Diagramas de Seqüência são tipos de diagramas de Iterações. Abrangem a visão dinâmica de um sistema.

Os Diagramas de Seqüência enfatizam a ordenação temporal, são utilizados para a documentação dos casos de uso e a representação das interações entre objetos. É também um diagrama de objetos que mostra a troca de mensagens entre eles e descrevem ao longo de uma linha de tempo a seqüência de comunicações entre objetos. Essas mensagens podem ser: síncronas e assíncronas (FOWLER *et al.,* 2005).

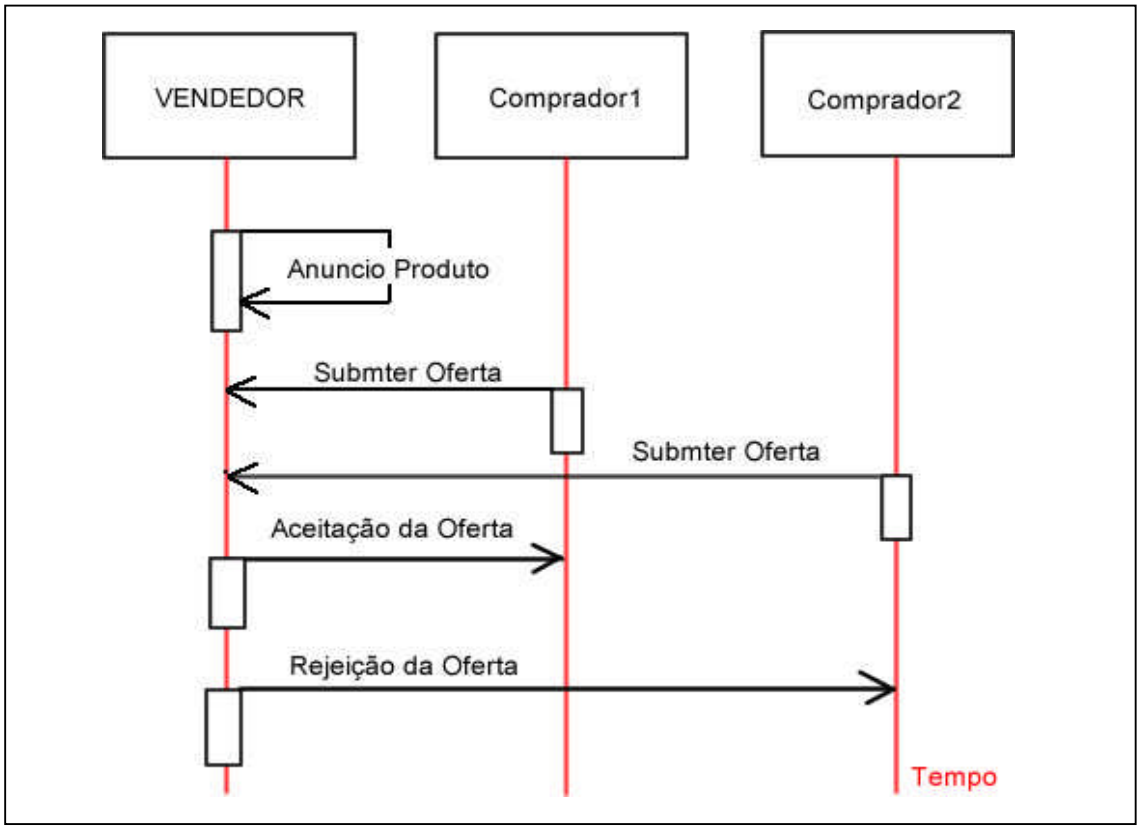

Figura 2.3 Diagrama de Seqüência (adaptada de FOWLER *et al.*, 2004).

Nas mensagens síncronas o emissor fica bloqueado até o receptor receber e tratar a mensagem, por exemplo, uma chamada de procedimento. Nas mensagens assíncronas o emissor continua a emitir mensagens, não há dependências, por exemplo, uma operação para apresentação de uma mensagem no monitor (FOWLER *et al.*, 2005).

Na Figura 2.3 é mostrada como os objetos e as mensagens se relacionam, como por exemplo, a troca de mensagens de vendedores e seus clientes.

- **Diagrama de Colaboração**

Um Diagrama de Colaboração é um diagrama de Interação que enfatiza a organização dos objetos que participam de uma interação. Nesses diagramas, primeiro se coloca os objetos que participam da interação como os vértices de um gráfico, em seguida representa-se os vínculos que conectam esses objetos como os arcos dos gráficos e junta-se esses vínculos com as mensagens que os objetos enviam e recebem, proporcionando a visualização dos fluxos de controle no contexto da organização estrutural dos objetos que colaboram (BOOCH *et al.,* 2000).

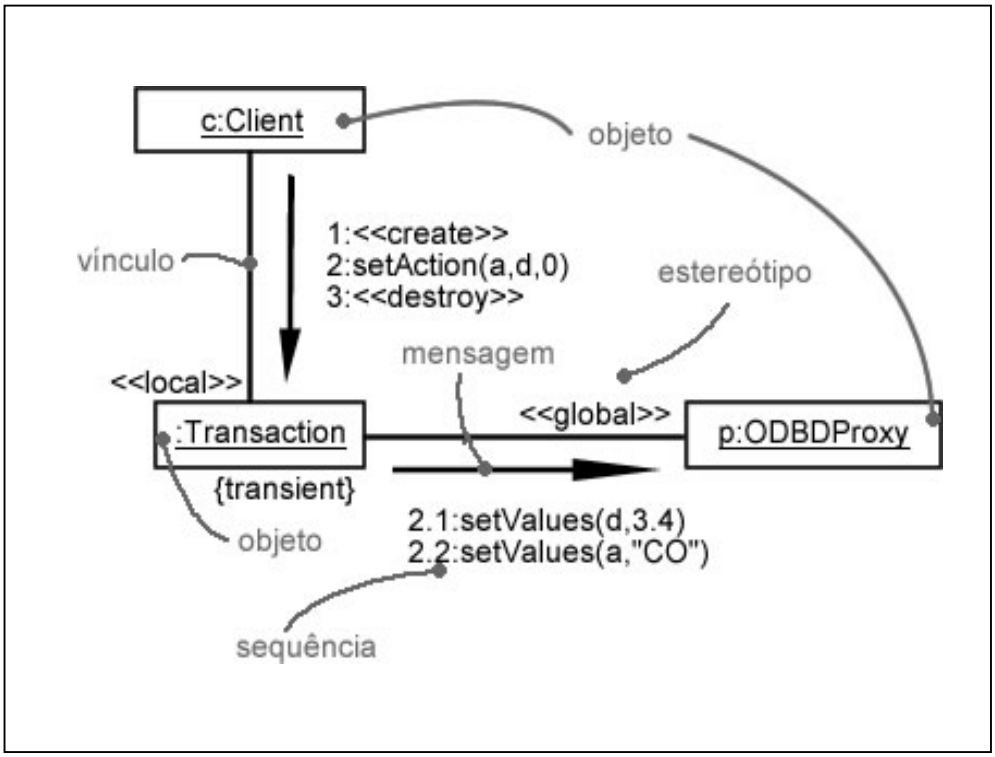

Figura 2.4 Diagrama de Colaboração (adaptada de BOOCH *et al.*, 2000).

São usados na modelagem dos aspectos dinâmicos de um sistema, por duas maneiras: Para a modelagem dos fluxos de controle por ordenação temporal, que enfatiza a passagem de mensagens com o tempo. E para a modelagem de fluxos de controle por organização, que enfatiza os relacionamentos estruturais entre as instâncias da interação(BOOCH *et al.*, 2000).

Na Figura 2.4 é mostrada um diagrama de colaboração coagindo com seus objetos, mensagens e vértices.

- **Diagrama de Máquina de Estados**

De acordo com Booch *et al.* (2000), Os diagramas de Máquinas de estados exibem uma máquina de estados, formados por estados, transições, eventos e atividades. Utilizadas para a modelagem de comportamentos de uma interface, classe ou colaboração, que enfatiza o comportamento de um objeto ordenado por eventos.

Os Diagramas de Máquinas Estados são empregados na modelagem dos aspectos dinâmicos de um sistema, principalmente na modelagem do comportamento de objetos reativos. Um objeto reativo possui um comportamento mais caracterizado por sua resposta a eventos ativados externamente ao seu contexto. Os diagramas de estados podem ser anexados a classes, a casos de uso ou a sistemas inteiros para visualizar, especificar, construir e documentar a dinâmica de um objeto individual (BOOCH *et al.*, 2000).

2.6. MOF (*Meta Object Facility*)

De acordo com Heaton (2005), MOF (*Meta Object Facility*), é um instrumento comum do ambiente CORBA (conjunto de serviços que várias aplicações podem compartilhar), para gerência de meta-informação (informações que estão contidas no corpo de um sistema). O MOF tem como objetivo o seu uso numa larga variedade de cenários, desde gerência de tipos de modelos, até desenvolvimento de softwares, gerência de informação e *datawarehousing*. O MOF pode ser usado como um repositório de meta-informação em sistemas distribuídos baseados em CORBA. É composto de interfaces padrões que podem ser

usadas para definir e manipular um conjunto de metamodelos interoperáveis e seus respectivos modelos. Sua versão corrente é a 2.0

A especificação do MOF é uma tecnologia adotada pela OMG. A especificação do MOF foi lançada em 1997, mesmo ano do lançamento da UML. A principal motivação inicial para a definição da especificação do MOF foi tornar possível a interoperabilidade de metamodelos utilizando a infra-estrutura de sistemas distribuídos do CORBA (ISO/IEC, 2005).

Do ponto de vista da modelagem, o MOF é usado para definir informações de modelos para um particular domínio de interesse. Essa definição é usada para guiar subseqüentes projetos de software e etapas de implementações para softwares com as informações dos modelos (HEATON, 2005).

A especificação MOF define um metamodelo. O propósito do modelo MOF é permitir a definição e a manipulação de metamodelos em vários domínios, com o foco inicial sendo em análise e projeto de metamodelos de objetos. A especificação MOF define um conjunto de interfaces CORBA IDL, que podem ser usadas para definir e manipular um conjunto de metamodelos interoperáveis e seus correspondentes modelos. Esses metamodelos interoperáveis, incluem o metamodelo da UML, o metamodelo MOF e futuras tecnologias adotadas pela OMG que serão especificadas usando metamodelos. A especificação MOF fornece a infra-estrutura para implementar o projeto e a reutilização de repositórios baseados em CORBA. Especifica regras de mapeamento preciso que permitem que interfaces CORBA para metamodelos sejam automaticamente geradas. Desta forma, dando consistência na manipulação de metadados em todas as fases do ciclo de desenvolvimento de uma aplicação distribuída (DSTC, 2001).

Metadado é uma abstração do dado, capaz, por exemplo, de indicar se uma determinada base de dados existe, quais são os atributos de uma tabela e suas características, tais como o tamanho e o formato (MELLOR *et al*., 2005).

A arquitetura de metadados do MOF é baseada em uma arquitetura tradicional de quatro camadas. O modelo MOF é orientado a objetos, suportando construtores de metamodelagem alinhados com os construtores da UML. O modelo MOF é descrito por ele mesmo, isto é o modelo MOF é formalmente definido usando os seus próprios construtores de meta-modelagem (DSTC, 2001).

Na Figura 2.5 são apresentadas as quatro camadas da arquitetura de metadados do MOF. No qual o nível M0 possui dados do aplicativo, por exemplo, as linhas de uma tabela de um banco de dados. O nível M1 possui o aplicativo, as classes de um sistema orientado para

objetos ou definições de tabelas de um banco de dados relacional. O nível M2 possui os metadados que capturam a linguagem da modelagem, os elementos da UML como a classe, atributos e Operação. O nível M3 é o nível em que o MOF se encaixa, possui os meta-metadados que descrevem as propriedades que o metadado pode exibir. O M3 também é o nível em que as linguagens de modelagens e os metamodelos operam (MELLOR *et al.*, 2005).

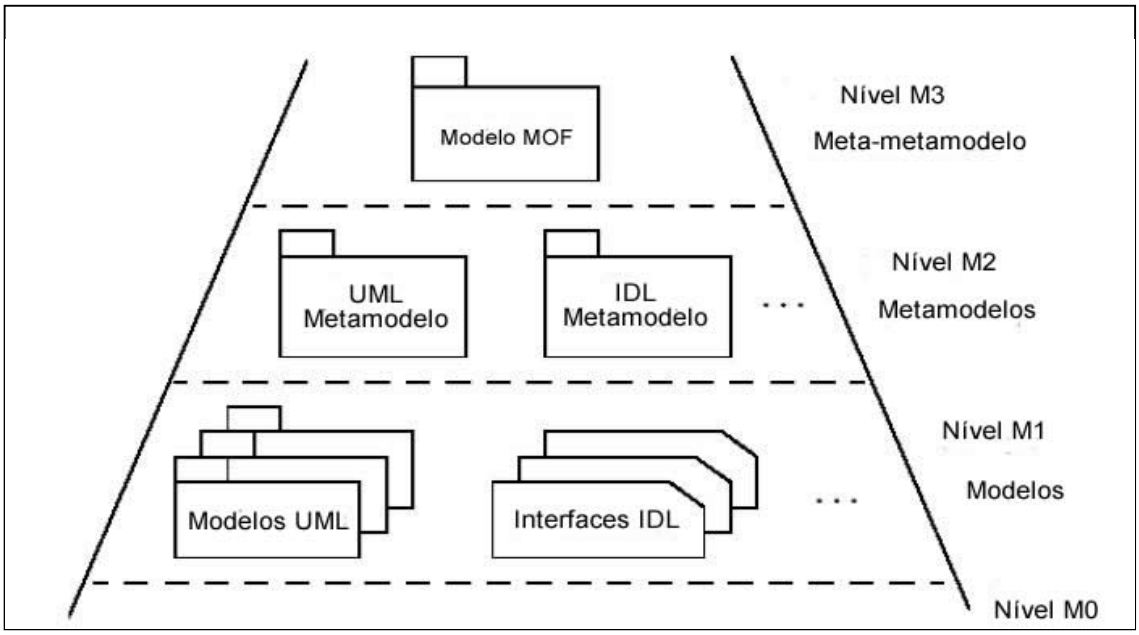

Figura 2.5. MOF Arquitetura de Metadados (adaptada de DSTC, 2001).

A metamodelagem do MOF é primeiramente sobre definição de modelos de informação para metadados. O MOF usa um framework de modelagem de objetos que é essencialmente um subconjunto do núcleo da UML. De acordo com a DSTC (2001), existem quatro construtores de modelagem: classes, associações, tipos de dados (*Data Type*) e pacotes.

A principal utilização do MOF é fornecer um ambiente de suporte para a definição e manipulação de metamodelos. Fornecer a infra-estrutura para permitir a interoperabilidade entre metamodelos servindo como um repositório padrão de metamodelos, através do mapeamento MOF-CORBA. A UML é um metamodelo que pode ser definido pelo MOF e também o CWM, um metamodelo padrão para integração de um ambiente de *DataWarehousing*, que também pode ser definido pelo metamodelo do MOF(HEATON, 2005).

No MDA, o MOF é uma importante ferramenta de metamodelagem, pois fornece acesso de dados padronizados, definindo padrões de suporte para serem usados por linguagens compatíveis com o MOF, por exemplo, o padrão XML a fim de fazer o intercambio e o acesso

de modelos compatíveis. Uma aplicação para MOF, por exemplo, é uma aplicação de funções de mapeamentos para modelos (MELLOR *et al.,* 2005).

2.7. CWM (*Common Warehouse Metamodel*)

De acordo com Chang (2001), *Common Warehouse Metamodel* (CWM) é um padrão de metadados cujo objetivo é permitir a integração de sistemas de *data warehouse*, *e-business* e sistemas de negócios inteligentes em ambientes heterogêneos e distribuídos, através de uma representação e de um formato de troca de metadados. O padrão CWM é parte dos esforços do grupo OMG (*Object Management Group)* no sentido de prover um *framework* arquitetural orientado a objeto e padronizado para aplicações distribuídas, de forma a suportar reusabilidade, portabilidade e interoperabilidade de componentes de software orientados a objetos em ambientes heterogêneos.

Proposto pelo grupo OMG em conjunto com organizações como IBM, *Oracle*, *Unisys*, *Hyperion Solutions* (*Essbase Software*), o padrão CWM foi adotado como um padrão OMG em junho de 2000 e é baseado nos seguintes padrões: UML, MOF (*Meta Object Facility*), uma metalinguagem e um padrão de repositório de metadados, e XMI (*XML Metadata Interchange*), um padrão baseado em XML para troca de metadados entre ferramentas e repositórios orientados a objetos (CHANG, 2001).

O padrão CWM é definido em UML e organiza os tipos de metadados por assunto:

- CWM *Foundation* provê os tipos de metadados para representação de conceitos e estruturas que são compartilhados por outros pacotes CWM;
- *Warehouse Deployment* provê os tipos de metadados para registrar como *hardware* e *software* são utilizados no D*atawarehouse*;
- *Relational* provê os tipos de metadados para descrever dados acessíveis através de uma interface relacional e segue o padrão SQL;
- *Record-Oriented* provê os tipos de metadados para descrição dos conceitos básicos de um registro e suas estruturas;
- *Multidimensional Database* (MDDB) corresponde uma representação genérica de um banco de dados multidimensional (MOLAP);
- XML provê os tipos de metadados para descrever fontes de dados em XML e é baseado na versão XML 1.0;

- *Transformation* provê os tipos de metadados para descrever transformações entre diferentes tipos de fontes de dados; OLAP (O*n-line Analytical Processing*) (OLAP, 2003) define um metamodelo dos construtores OLAP essenciais, presentes nas aplicações e ferramentas OLAP;
- *Warehouse Process* provê os tipos de metadados para documentar o fluxo de processos utilizados para executar as ransformações;
- *Warehouse Operation* contém classes para o registro das operações diárias de um processo de D*atawarehouse* (IBM, CHANG, 2001).

O modelo em CWM especifica a função da estrutura ou o comportamento de um sistema. Seu metamodelo é projetado para maximizar o reúso do modelo orientado a objeto (em conjunto com a UML). O CWM possui pacotes de metamodelos para controlar a complexidade e suportar o reúso dos modelos (HEATON, 2004).

2.8. Conceitos de Processos de Arquitetura Orientada por Modelos

De acordo com Mellor *et al.* (2005), um Processo MDA é uma extensão da identificação de modelos. A definição se dá no planejamento da passagem de funções de mapeamentos entre uma declaração do problema e de um sistema codificado, ou seja, são "saltos" de um ou mais modelos de origem para um único modelo destino. Quando se junta dois ou mais conjuntos de modelos de mapeamentos, forma-se uma cadeia de mapeamento.

O processo MDA determina somente as quais funções de mapeamento cada processo individual se aplica. Em um processo planejado, é muito importante saber quais modelos construir e quais podem ser reusados, quais funções de mapeamento construir e quais podem ser reusadas e como fazer uso do código do legado. Código do legado são os códigos que estão funcionando corretamente (MUKERJI E MILLER, 2003).

Os mapeamentos podem ser classificados pelo nível de abstração em que se enquadram: **mapeamento de salto curto e mapeamento de salto longo**.

Um mapeamento de salto curto não possui muito efeito no nível de abstração entre modelo de origem e o modelo destino, por exemplo, uma função de mapeamento que transforma um modelo de classe em um conjunto correspondente de classes de implementação, operações para métodos e atributos de um sistema. Os mapeamentos de salto curto também pode auxiliar para evitar redundâncias que ocorrem em mapeamentos de salto longo. **Um mapeamento de salto longo** tem uma modificação significativa no nível de

abstração, por exemplo, um mapeamento que aceite os elementos de um modelo e os mapeie diretamente para *bits* (MELLOR *et al.,*2005).

A construção de um processo MDA é dividida em duas técnicas: focar-se na pesquisa de modelos que existem em um único nível de abstração e focar-se no comprimento da cadeia de mapeamento e descobrir um comprimento ótimo para cada "salto" dentro da cadeia (MELLOR *et al.*, 2005).

Para uma base para a definição do processo MDA faz-se um gráfico de domínio. Um gráfico de domínio possui uma coleção de domínios de problema e funções de mapeamentos que compõem o sistema. A construção e manutenção desse gráfico fornecem um *framework* (ou seja, um arcabouço) para o trabalho contínuo sobre o projeto, ajuda a mostrar ao desenvolvedor quais modelos são completos, quais são as funções de mapeamento, quais são os metamodelos e quais são os modelos de marcação (MELLOR *et al,*. 2005).

Para um processo MDA deve-se realizar uma construção de um modelo de uma plataforma em uma linguagem de modelagem que podem ser transformadas em outras e mapear o conhecimento de formalizado de uma plataforma de destino a fim de criar um sistema (MELLOR *et al.,* 2005). Para formalizar esse conhecimento devemos abranger quatro atividades: coleta de requisitos, abstrair esse conhecimento para um conjunto de conceitos, expressar esses conceitos em um modelo e por fim testar esses modelos.

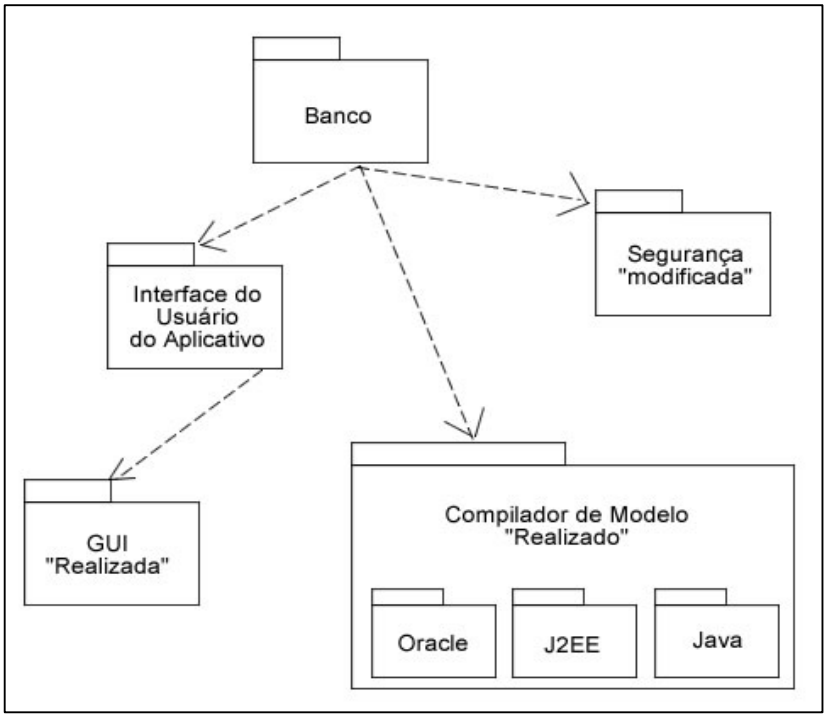

Figura 2.6. Exemplo de gráfico de domínio (MELLOR *et al.*, 2005).

Na Figura 2.6 é apresentado um exemplo de um gráfico de um processo MDA, no qual mostra um banco e sua importância com a segurança e uma interface de usuário, ambos totalmente independentes um do outro, tornando modelo portável.

2.9. Ferramentas que apóiam o MDA

Há décadas os desenvolvedores vêm estudando as formas para um desenvolvimento de software mais rápido e eficaz, o MDA traz algumas ferramentas que as apóiam, é o caso da ferramenta AndroMDA, OptimalJ, ArcStyler e Enterprise Architect.

2.9.1 A Ferramenta AndroMDA

A AndroMDA é uma ferramenta *open source* (de código aberto) baseado em MDA (*Model Driven Architecture).* Esta ferramenta utiliza modelos UML gerados sob a forma XML e com base em vários *plugins*, chamados de cartuchos (*cartridges*) para realizar a geração de componentes customizados (códigos fontes) para diversas plataformas, por exemplo, J2EE e .NET (ANDROMDA, 2006).

A Ferramenta AndroMDA vem com alguns *plugns* (cartuchos) prontos para geração de artefatos para ferramentas de desenvolvimento, como Hibernate, Spring, JSF, WebServices e outros, o qual realiza o mapeamento entre o modelo PIM e o modelo PSM O AndroMDA também possui um conjunto de ferramentas para a construção de seus próprios cartuchos ou customizar os já existentes, chamado de meta cartucho. Usando o meta cartucho pode-se construir um gerador de código customizado usando a ferramenta UML desejado (ANDROMDA, 2006).

Atualmente o AndroMDA possui as seguintes características:

- Design modular: todos os blocos principais de construção do AndroMDA são plugáveis e podem ser trocados para atingir as suas necessidades.
- Suporte para a maioria das ferramentas UML, como *MagicDraw, Poseidon*, *Enterprise Architect* e outras.
- Possui metamodelo UML 1.4 completo, podendo utilizar o seu próprio metamodelo em MOF, XMI e gerar código a partir de modelos baseados nele.

- Validação de entrada do modelo utilizando restrições OCL. Possui restrições pré-configuradas que protege o seu modelo contra os erros mais comuns de modelagem. Também possibilita a criação de restrições específicas para seu projeto.
- Transformações modelo para modelo ajudam a aumentar o nível de abstração.
- Pode gerar qualquer tipo de saída de texto usando *templates* (código fonte, scripts de banco de dados, páginas *web*, etc.)
- *Templates* são baseados em *template engines* bem conchecidos. Atualmente *Velocity* e *FreeMarker* são suportados.
- Cartuchos prontos para usar para as arquiteturas mais comuns (EJB, *Spring*, *Hibernate, Struts*, JSF, *Axis*, JBPM).

2.9.2. A Ferramenta OptimalJ

A OptimalJ, é uma ferramenta comercial de geração de aplicações J2EE que oferece um ganho de produtividade. Gera aplicações J2EE em ambiente Windows,Unix e Linux com ambiente de testes integrado. A aplicação pode ser desenvolvida independe de Banco de Dados, o que significa que pode ser desenvolvido em DB2, SQL Server, Oracle entre outros. A Ferramenta OptmalJ utiliza a metodologia MDA e proporciona uma diminuição do tempo do projeto em cerca de 35% do desenvolvimento (COMPUWARE).

Possui as principais características:

- Desenvolvimento rápido da aplicação através de uma abordagem visual, orientada para o modelo;
- Desenho e construção de aplicações;
- Geração de aplicações orientadas para o padrão J2EE;
- Editor de Regras Comerciais, com suporte a regras estáticas e dinâmicas;
- Sincronização ativa entre modelos e código;
- Integração da distribuição para teste e *debugging*.

2.9.3. A Ferramenta ArcStyler

A Ferramenta ArcStyler é um produto comercial baseado na metodologia MDA . Atualmente o ArcStyler está disponível em duas edições, uma edição da empresa ArcStyler e uma edição do arquiteto do ArcStyler. Na edição da empresa, o ArcStyler suporta o desenvolvimento em MDA com plataformas J2EE e .NET. Já na edição do arquiteto, o ArcStyler oferece ferramentas poderosas permitindo sua extensão, possui *plugins* (cartuchos), podendo ser customizáveis e flexíveis para um bom detalhamento da arquitetura em MDA. Também sustenta o padrão UML, e no contexto de modelação possui características de migração entre ferramentas e o reúso dos modelos (ARCSTYLER, 2005).

Suas principais características são:

- Geração de código para a arquitetura J2EE.
- É flexível a ponto de suportar outras arquiteturas, como .Net e CORBA.
- Possui infra-estrutura de testes automatizados. Os testes unitários e de integração são modelados em UML, em um alto nível de abstração.
- Geração de testes automatizados para a lógica de negócio através da linguagem OCL (*Object Constraint Language*).
- Suporte à geração do esquema do banco de dados da aplicação.
- Utiliza o Ant (APACHE) como ferramenta de automação de projeto.

2.9.4. A Ferramenta Enterprise Architect

A Ferramenta *Enterprise Architect* (E.A.) é uma ferramenta comercial para modelagem orientado a objetos e baseado na metodologia MDA. A ferramenta utiliza as técnicas de UML e MOF e cobre todas as exigências do desenvolvimento de software, desde ao estágio de análise à manutenção (SPARX SYSTEM, 2006).

A ferramenta ajuda a controlar a complexidade na sustentação de modelos grandes e com controle de versão. Também suporta a geração e a engenharia reversa do código fonte para muitas línguas populares como: C++, C#, Java, Delphi, VB.Net, Visual Basic, ActionScript e PHP. *Enterprise Architect* também suporta os modelos e diagramas da UML 2.0, podendo modelar processos de negócios, *web sites*, interfaces de usuário, *networks*, configurações de *hardware*, mensagens e etc (SPARX SYSTEM, 2006).

.

Possui diversas características:

- Compilação de um sistema complexo de modelo em UML de *softwares* e *hardwares.*
- Geração de códigos de engenharia reversa em, C++, C#, Delphi, Java, PHP, VB.Net, Visual Basic e Phyton. (Somente nas Edições: *Professional* e *Corporate Edition*)
- Geração de modelos *DataBase* e DDL Scripts. (Somente nas Edições: *Professional* e *Corporate Edition*)
- Controle de mudança, manutenção e testes de certificados.
- Distribuição de modelos, componentes e detalhes da implementação.
- Documentação nos formatos RFT e HTML.
- Saída de modelos no formato XML 1.0, XML 1.1, XML 1.2 e XML 2.1 compatível a outras ferramentas que importam e exportam XML.
- Importação de modelos no formato XML 1.0, XML 1.1, XML 1.2 e XML 2.1 outras ferramentas.
- Controle de Versão de projetos.
- Conexão com o SQL Server, MySQL ou Oracle e 10g DataBase(Somente na edição: *Corporate Edition*)
- Controle de pacotes baseados na exportação e importação de XML.
- Transformações de modelos no estilo MDA.

A principio a ferramenta escolhida possui uma interface completa e flexível. Na Figura 2.7 é mostrada a tela principal da Ferramenta *Enterprise Architect.* A tela principal da ferramenta possui 3 partes principais, *Toolbox* que contém as ferramentas para modelagem nos diversos tipos de diagramas (A), *Start Page* que contém o local em que são modelados os diagramas (B) e *Project View* que contém todos os detalhes do projeto que está sendo construído (C).

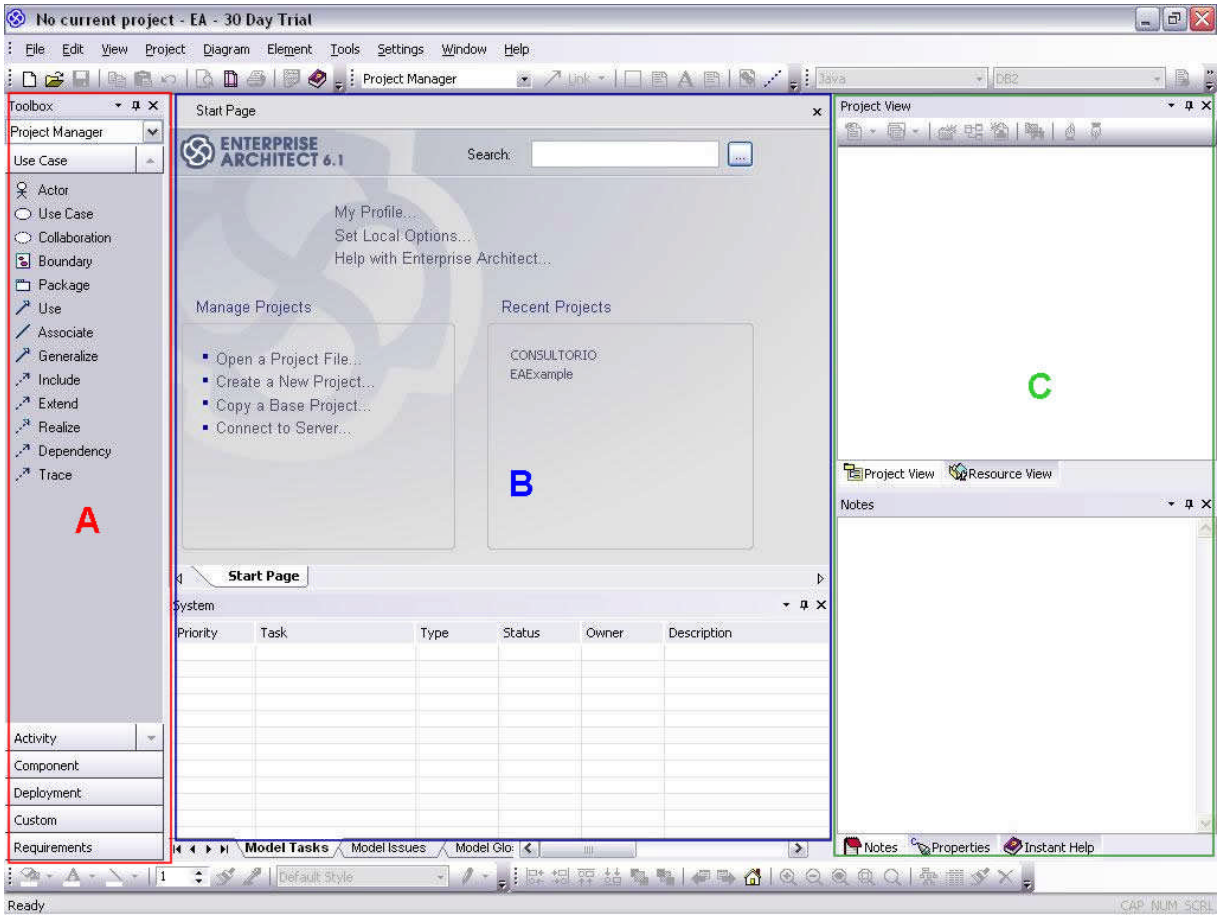

Figura 2.7 – Tela Principal da Ferramenta Enterprise Architect.

2.10. Quadro de Comparações

Com o objetivo de comparar as ferramentas pesquisadas, tem-se uma tabela (Quadro 2.1.) onde os seguintes parâmetros de comparação foram utilizados: **Gratuita**, que diz se a ferramenta é gratuita ou não. **Código aberto**, que diz se a ferramenta possui o código aberto ou não. **EJB**, **Struts**, **Hibernate** e **.Net**, que diz se a ferramenta possui suporte às plataformas mais usadas recentemente como EJB, Strutus, Hibernate e .Net. respectivamente. **Versão**, que diz a versão da ferramenta. **Suporte**, que diz se a ferramenta possui algum suporte por parte de seus desenvolvedores. **Extensível**, que diz se é possível customizar a ferramenta. Foi utilizado como padrão S para Sim e N para Não.

	AndroMDA	OptimalJ	ArcStyler	E.A.
Gratuita	**S**	**N**	**N**	**N**
Código Aberto	**S**	**N**	**N**	**N**
EJB	**S**	**S**	**S**	**S**
Struts	**S**	**S**	**N**	**N**
Hibernate	**S**	**S**	**N**	**N**
.NET	**S**	**N**	**S**	**S**
Versão	**3.2**	**3.3**	**5.5**	**6.5**
Suporte	**S**	**S**	**S**	**S**
Extensível	**S**	**S**	**N**	**S**
Preço (US $)	**Gratuito**	**24,240 (1 ano)**	**40,00(3 anos)**	**239,00**

Quadro 2.1 Comparação entre as ferramentas AndroMDA, OptimalJ e ArcStyler (ANDROMDA, 2006; ARCSTYLER, 2005; COMPUWARE, 2005; SPARX SYETEM, 2006).

2.11. Considerações Finais

Neste capítulo são descritos todos os conceitos para se entender o paradigma MDA, ressaltando a importância dos modelos e suas construções para um bom desenvolvimento de um projeto, e que a maior dificuldade são as abstrações entre as plataformas PIM e PSM. Também se ressalta a importância dos padrões como a UML, MOF e CWM, tendo elas a interoperabilidade entre elas e é muito importante que ferramentas construídas para esses padrões também se encaixem juntas dentro dessa arquitetura e por tanto, criar um ambiente completo de desenvolvimento orientado por modelos. Algumas ferramentas com geração de código foram citadas e descritas as principais características.

3. ESTUDO DE CASO

Neste capítulo apresenta-se o estudo de caso para apoiar o estudo do MDA, desenvolvendo um sistema de pequeno porte, na ferramenta escolhida. Na seção 3.2 aborda-se uma descrição do estudo de caso. Na Seção 3.3 tem-se o diagrama de casos de uso modelado a partir do documento de requisitos estudado. Na Seção 3.4 é mostrado como o modelo PIM foi modelado para o sistema. A Seção 3.5 traz a transformação do modelo PIM para PSM, foram feitos transformações para as plataformas em C# e Java. Na seção 3.6 aborda-se a geração do código fonte a partir do modelo PSM. Na Seção 3.7 abordam-se as considerações finais.

3.1. Descrição do Estudo de Caso

Visando a fazer uma análise utilizando ferramentas para o desenvolvimento de sistemas em nível comercial, que trazem diversas vantagens utilizando a tecnologia MDA (*Model Driven Architecture*), foi desenvolvido um sistema de pequeno porte voltado ao cadastro e manutenção de clientes para consultório dentário. Após pesquisas realizadas na Internet, optou-se por utilizar a ferramenta *Enterprise Architect* (EA), por ser uma ferramenta versátil, de fácil instalação, aborda diretamente a tecnologia MDA e a única disponível de graça entre as ferramentas de código fechado estudado.

O *software Enterprise Architect* é uma ferramenta comercial, disponível para compra nas edições, *Corporate Edition*, *Professional Edition*, *Desktop Edition* e *TRIAL.* Para o estudo de caso foi utilizada a versão *TRIAL*, disponível por trinta dias.

O sistema desenvolvido é um sistema de gerenciamento de cadastro de clientes para consultório dentário, gerenciamento de reservas e consultas, gerenciamento de pagamentos, gerenciamento de consultas e relatórios. Foi feita toda a análise de documento de requisitos funcionais e não funcionais para esse sistema, auxiliando a geração dos diagramas de casos de uso e diagrama de classes. Essa documentação está como anexo no final deste trabalho.

A principal meta do estudo de caso são o estudo e a construção do modelo PIM em que a Ferramenta EA faz uma transformação e gera elementos do PSM. Este modelo PSM por sua vez, deve ser a base para a geração automática de código.

3.2. Diagrama de Casos de Uso

Para o estudo do estudo de caso foram construídos diagramas de Classe e de Casos de Uso. Os diagramas de Casos de Uso não são os diagramas mais importantes para o MDA, os diagramas apenas especificam o comportamento do sistema, seus relacionamentos, para um auxílio posterior, na fase de construção dos diagramas de Classe.

Seguindo a visão geral do estudo de requisitos, foi implementado um diagrama de casos de uso na Ferramenta *Enterprise Architect.* O modelo de casos de uso implementado ficou como mostrada na Figura 3.1, gerido da própria ferramenta como documentação.

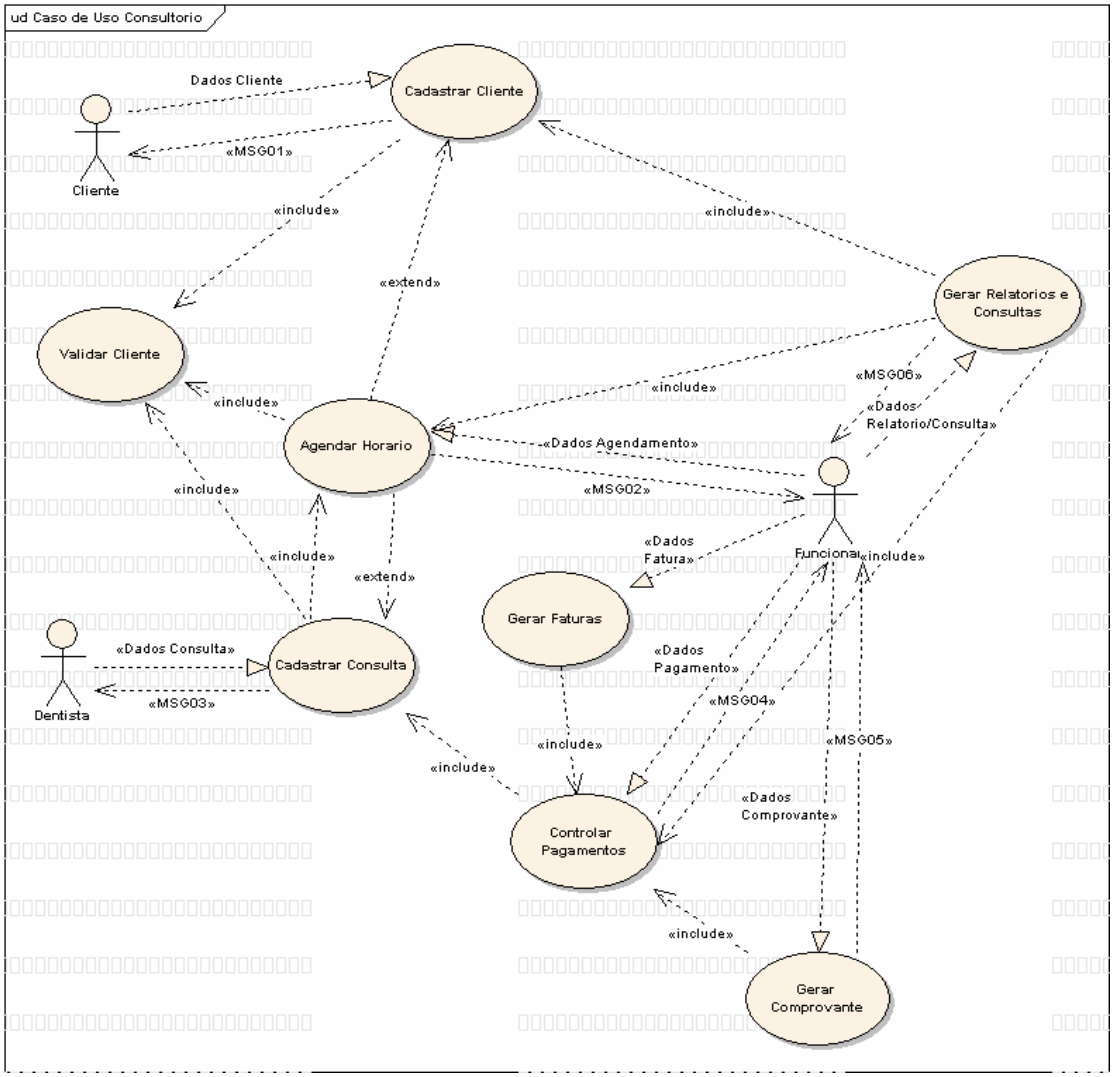

Figura 3.1 – Diagrama de Casos de Uso.

A figura 3.1 não possui nenhuma diferença entre as modelagens convencionais orientados a objetos. Assim devendo sempre modelar igualmente para o MDA.

3.3. Construção do PIM

O próximo passo é modelar um modelo independente de plataforma (PIM). A Ferramenta EA Foi utilizada para a construção do diagrama de classes, esse diagrama de classes apresenta a modelagem do Modelo Independente de Plataforma.

Ao construir o PIM na Ferramenta EA, alguns pequenos cuidados na modelagem foram tomados:

- Foram usados tipos genéricos da ferramenta dentro dos modelos, como, *Integer, Long, Boolean* e *String*, para os atributos na modelagem do PIM. Estes tipos de dados são automaticamente mapeados durante a transformação do PIM para o PSM, por exemplo, é usado tipo genérico Integer da Ferramenta EA para os atributos do tipo inteiro na modelagem do PIM, após a transformação do PSM na plataforma C#, os atributos do tipo "Integer" do PIM irão ficar do tipo "int" no PSM.
- Para os atributos que precisem ter um resultado de retorno nas operações, os atributos do modelo de origem (PIM) foram colocados como do tipo "*public*", na transformação a Ferramenta EA automaticamente transforma o atributo "*public*" em "*private*", criando as operações previstas de get( ) e set( ). Para que não ocorra isso se deve marcar os atributos como "*private*" no modelo PIM.

Deve-se lembrar que o PIM é um modelo pouco detalhado, com alto nível de abstração e representado no padrão UML. Também é totalmente independente de plataforma e é a base para o modelo específico de plataforma (PSM). Nota-se também, que se segue exatamente a modelagem do padrão UML, não possuindo nenhuma diferença do processo convencional no desenvolvimento do modelo de diagrama de classes (FOWLER *et al.*, 2005, MELLOR *et al.,* 2005).

O modelo PIM do estudo de caso possui três classes, as classes CLIENTE, AGENDA, CONSULTA e PAGAMENTO, ambos possuem seus atributos e operações originados da fase de análise do sistema, na Figura 3.2 é mostrada exatamente como ficou o modelo independente de plataforma.

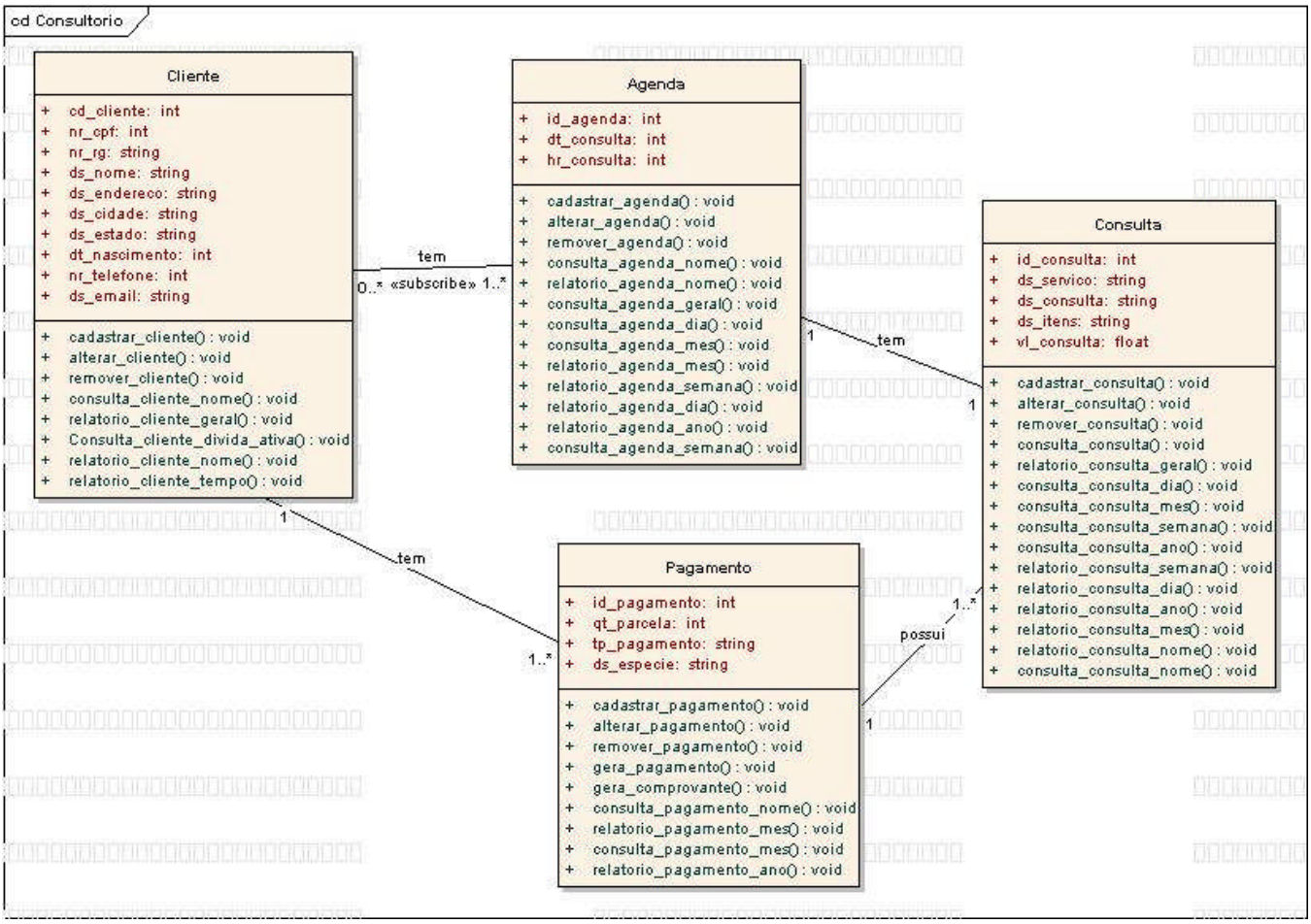

Figura 3.2 - Modelo PIM

Pode-se observar, que o PIM acima modelado, possui um nível de abstração alto, podendo representar o sistema de modo independente de qualquer tecnologia de implementação, possuindo a funcionalidade do sistema.

O MDA é baseado na construção de modelos formais, que sob muitos aspectos correspondem a uma importante documentação do sistema. O PIM é o modelo mais importante, pois também corresponde a uma documentação de alto nível de abstração, podendo ser visualizado, armazenado e processado automaticamente. O PIM também ser reutilizados para uma migração em uma outra plataforma ou, reutilizados para incorporação de alterações no sistema.

Ressalta-se ainda, que os modelos do MDA são modelados semelhantemente como os modelos convencionais, o grande diferencial em MDA para o desenvolvimento convencional, é a fase de transformação entre o PIM e o PSM (SIEGEL, 2002), em que, não são feitos manualmente, proporcionando uma diminuição no tempo de mapeamento.

3.4. Transformação do PIM para PSM

Após a definição do modelo independente de plataforma, foram aplicadas transformações ao PIM para gerarmos um novo modelo que leva em conta características da plataforma. Pode-se gerar mais de um PSM a partir de um PIM. Assim, podendo ter mais modelos, cada um compondo uma parte do sistema.

Com o modelo PIM correto, para fazer a transformação foram selecionados primeiramente os tipos de plataforma em que se deseja fazer as transformações. Foram escolhidas para o exemplo as plataformas em C# e Java. As transformações invocam automaticamente as características da geração do código para classes do PSM que correspondem às classes selecionadas do PIM, como mostrada na Figura 3.3.

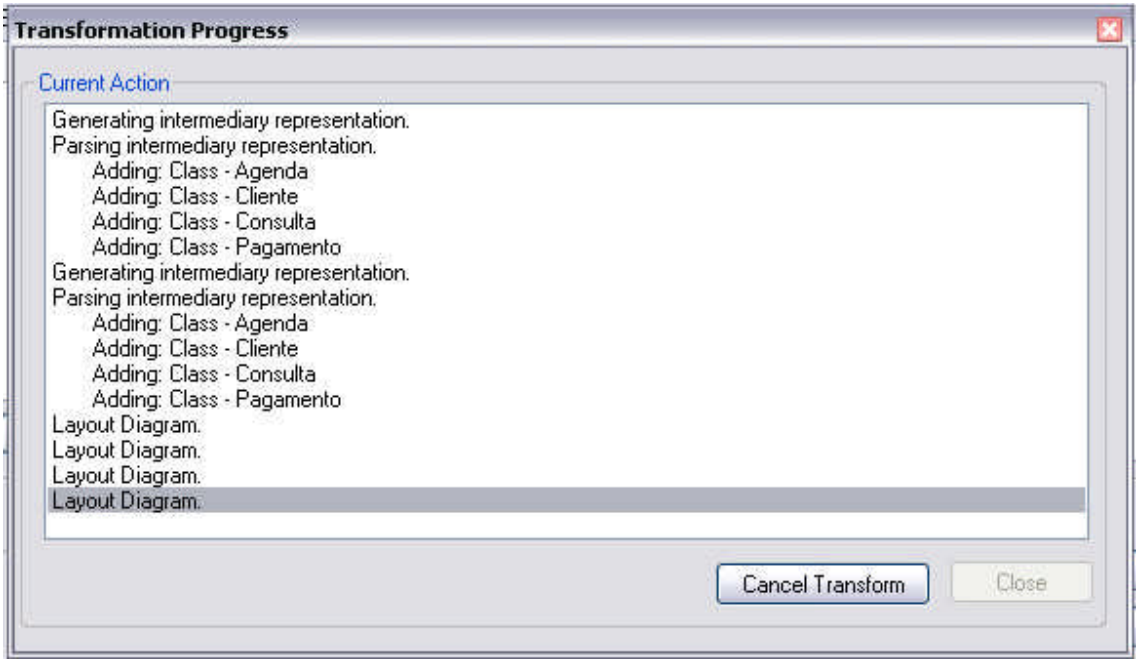

Figura 3.3 – Tela da transformação em Progresso (EA).

Com a execução da transformação a Ferramenta EA cria dois pacotes e os nomeiam como “C# Model” e “Java Model” onde estará o resultado das transformações. Os Diagramas PSM ficaram da seguinte forma, como apresentados na Figura 3.4 e na Figura 3.5 respectivamente.

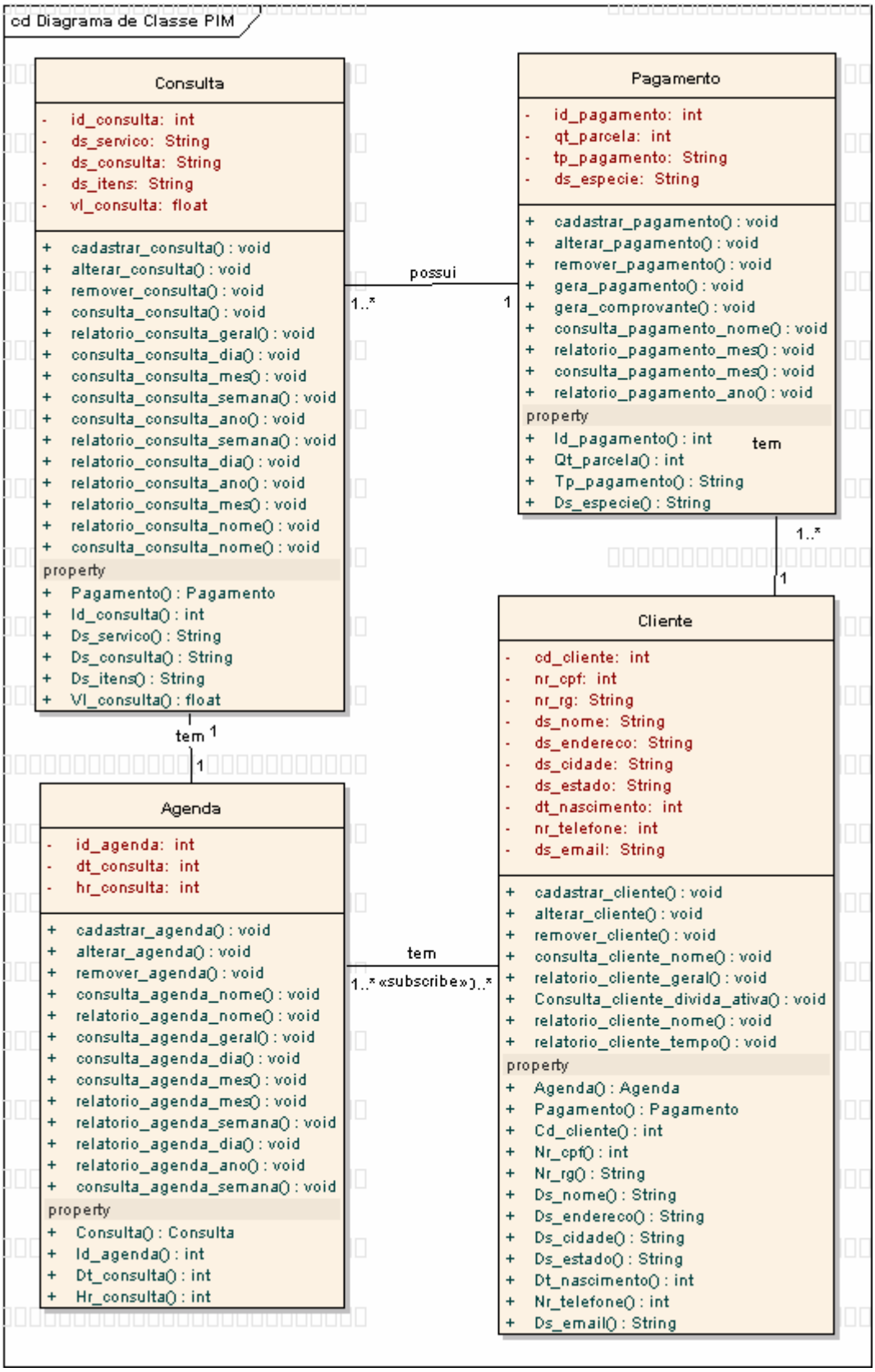

Figura 34 – Diagrama de Classes - Modelo PSM C#

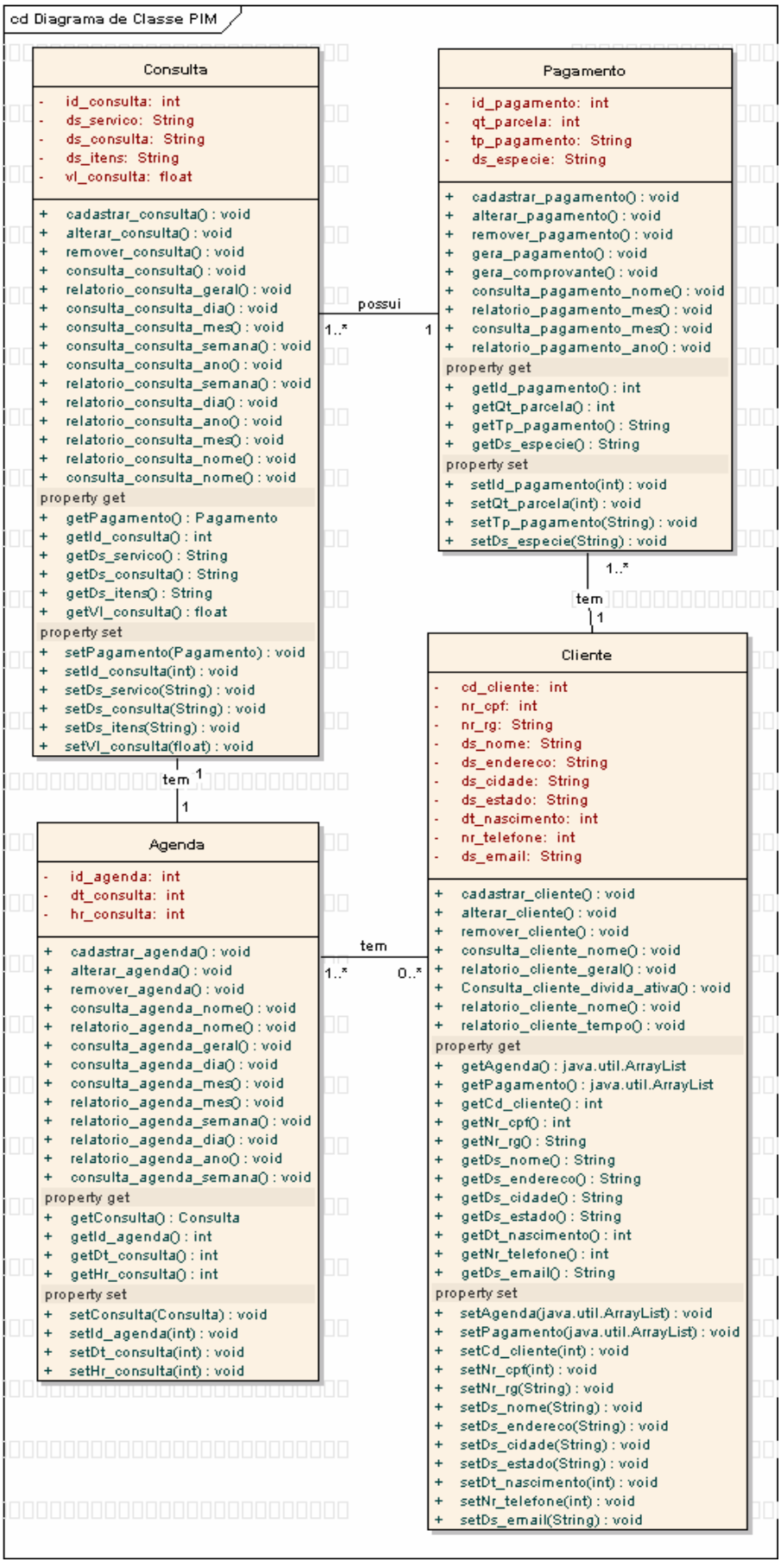

Figura 3.5– Diagrama de Classes - Modelo PSM Java

Nos PSM apresentados acima, nota-se que cada PSM possui suas características de sua plataforma. Podemos ver que em cada figura, foram criadas as operações previstas que cada plataforma necessita. Como por exemplo, foram criadas as operações de get( ) e set( ) para a plataforma Java no modelo PSM, Figura 3.5. Então, ambos elementos possuem características da plataforma no PSM.

Após as Transformações devemos notar os seguintes resultados nos diagramas PSM em C# e Java:

- A conversão dos tipos de dados genéricos para os tipos de dados em C# ou em Java.
- A conversão dos atributos públicos em atributos privados.
- A criação das propriedades em C#, incluindo as associações UML com o mapeamento apropriado das classes.
- A criação das propriedades de Java, incluindo as associações UML com o mapeamento apropriado das classes em Java.

Agora os elementos especificados no PSM estão prontos para a geração de código.

3.5. Geração de Código

Gerado os Modelos PSM C# e Java, pôde agora ser usada como o modelo da fonte para a transformação de código fonte na ferramenta EA, como mostrada na Figura 3.6. Para cada pacote que a ferramenta criou, foram gerados automaticamente os códigos para suas respectivas plataformas, o pacote C# Model" gerados para códigos em C# e o pacote "Java Model" para os códigos em Java.

Os códigos podem ser vistos ao final deste trabalho, em anexo. Agora esses códigos podem ser editados, compilados e construídos, usando o editor fornecido pelo EA, ou usando um editor externo como o Visual Studio para C# ou Eclipse para Java, a Sparx também fornece um extensão ao EA para a integração com o Visual Studio e o Eclipse.

A partir do código gerado, pode-se também efetuar a engenharia reversa e recuperar o modelo a partir do código. Removendo suas características específicas do projeto, a plataforma pode retornar para um modelo independente dela, e repassar o modelo para outras plataformas.

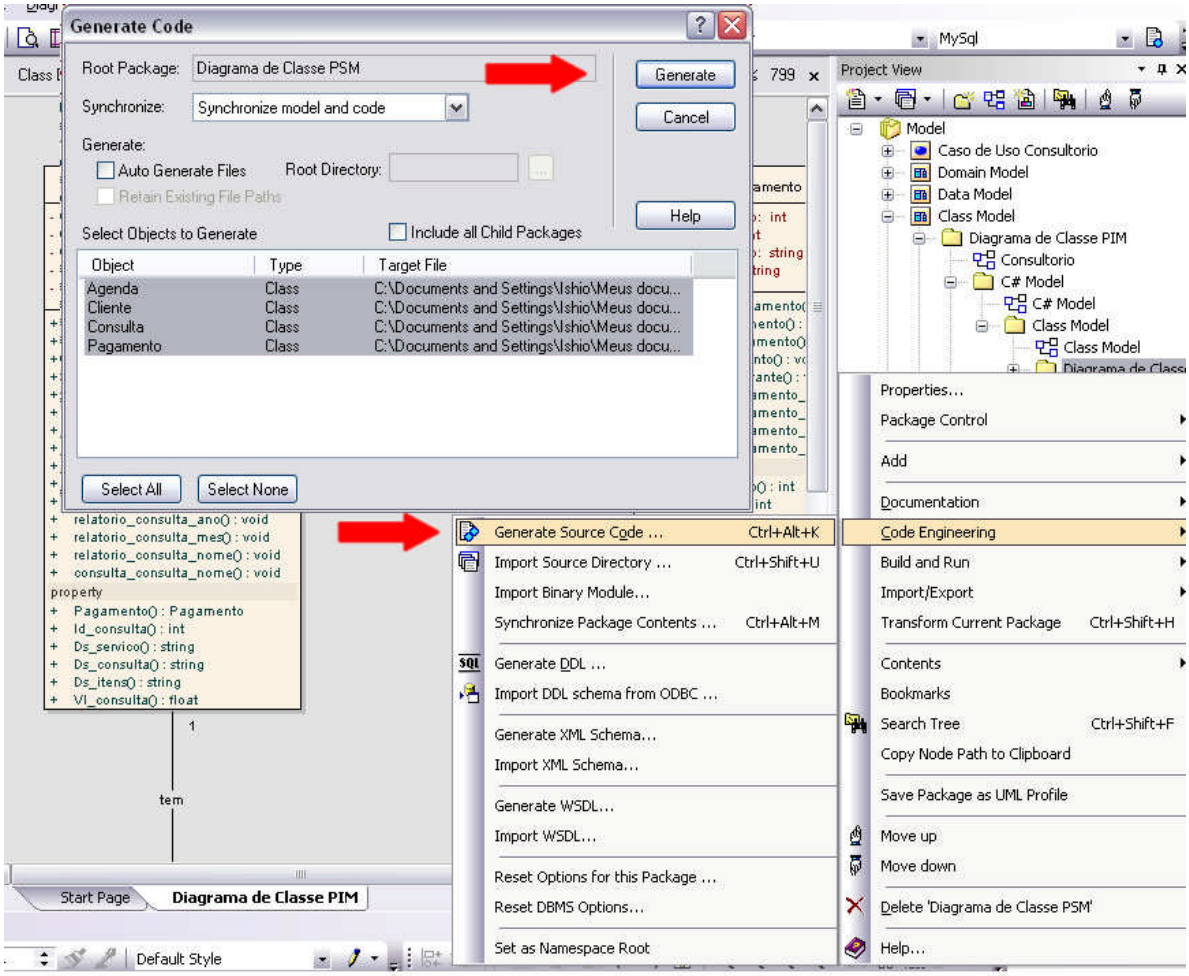

Figura 3.6 – Geração de Código (EA).

3.6. Considerações Finais

Durante o desenvolvimento do estudo de caso, foram encontradas algumas dificuldades impostas pelas limitações da ferramenta na versão TRIAL. Estas dificuldades estão descritas nesta seção.

O EA também suporta a gestão colaborativa de modelos, usando bases de dados. Na versão TRIAL não foi possível estabelecer esta conexão. No entanto uma base de dados como, SQL server, MySQL, Oracle9i, PostgreSQL, MSDE, Adaptive Server Anywhere, pode ser usado como repositório de modelos colaborativos.

Uma segunda dificuldade foi estudar as diversas configurações que o EA oferece e estabelecer qual funcionalidade oferece, onde muitas delas sem sucesso.

CONCLUSÃO

A metodologia de desenvolvimento MDA tem sido bem usada pelas indústrias de softwares e por pesquisadores de engenharia de softwares. O MDA é um *framework* promissor no desenvolvimento de software centrado na definição de modelos formais, cuja chave de compreensão e utilização é a importância dos modelos durante o processo de desenvolvimento de *software.* Dentro da arquitetura proposta pelo MDA o processo de desenvolvimento de *software* é dirigido completamente pelas atividades de modelagem do sistema em diferentes níveis de abstração semelhante ao desenvolvimento de *software* convencional. A diferença reside dos artefatos (PIM, PSM) criados durante o processo de desenvolvimento segundo o MDA, são os modelos formais que podem ser processados automaticamente, ou seja, a automação no processo de mapeamento entre os modelos independentes de plataforma (PIM) e os modelos de plataforma específica(PSM).

Também foi visto que conceitualmente o ciclo de desenvolvimento de sistemas através do MDA é relativamente semelhante a outros processos de desenvolvimento de software convencionais, ou seja, compreende fases de documento de requisitos, análise, projeto, codificação, depois posteriormente testes e implantação.

O código gerado, embora não constitua um modelo propriamente dito, também é um elemento importante nesta arquitetura, pois representa seu resultado concreto. Foi visto também que a geração de código é uma etapa relativamente simples dada a proximidade do PSM com a tecnologia particular em uso.

Uma das desvantagens vista é que após a geração do código ainda são necessários alguns ajustes e complementações, que devem ser feitos pelo programador ou por uma equipe de programação. Com o tempo espera-se que esses erros sejam quantitativamente menores nas ferramentas que estão surgindo no mercado.

A modelagem do PIM permite a adição de novas funcionalidades com relativa facilidade, mas por si só não garante que tais funcionalidades se articulem de modo coerente. Assim continua nas mãos dos projetistas avaliarem e garantir tal consistência, o que continua a exigir a alocação de profissionais experientes, e assim percebemos que o MDA não contribui de forma efetiva neste sentido.

Observou-se também que dentre as diversas ferramentas que adotam o MDA, inclusive o EA, não são capazes de realizar completamente as transformações entre PIM e PSM, ou mesmo efetuar a geração de código a partir do PSM, pelo menos não como idealizado pela proposta desta arquitetura.

Considerando este contraponto, concluiu-se que os principais benefícios do MDA são:

- PRODUTIVIDADE - No MDA o foco da atividade de desenvolvimento se situa na construção PIM, pois os PSMs deverão ser obtidos de forma automática nas transformações em alguma das ferramentas suportadas pelo MDA. Efetivamente há um ganho de qualidade e produtividade proporcionadas pelas atividades de modelagem.
- PORTABILIDADE – Com um mesmo PIM pode ser transformado em diferentes PSMs, possibilitando que um sistema possa ser construída em diferentes plataformas, então a construção do PIM é portável.
- REUSABILIDADE – Através de alterações no PIM do sistema é possível a geração de novos PSMs e códigos correspondente muito rapidamente, agilizando os procedimentos de manutenção do sistema. Com isso correções, adaptações ou a adição de novas funcionalidades tornam-se tarefas mais simples de serem realizadas.

TRABALHOS FUTUROS

Um dos trabalhos futuros que pode ser realizado é fazer um novo estudo de caso com ferramentas diferentes, utilizando o mesmo modelo independente de plataforma PIM. Com esses novos estudos de casos pode-se fazer uma análise das diferenças dos resultados obtidos nas transformações do PIM para o PSM, e as gerações de código fonte.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDROMDA. **Site official: "*What is AndroMDA*?" Artigo gerado em: 22 june, 2006.** Disponível em: http://www.andromda.org/index.php?option=com_content&task=blogcategory&id=0&Itemid=42 . Acesso em: 15, junho 2006.

APACHE. **"Ant – *Intruduction",*Site Ofiial da *Ant, 2004*.** Disponível em: http://ant.apache.org/manual/index.html. Acesso em: 24, julho 2006.

ARCSTYLER. **"ArchStyler *Overview*", *WhitePaper* : 220205. 2005.** Disponível em: http://www.omg.org/mda/mda_files/ArcStyler5_Whitepaper_220205.pdf. Acesso em: 21, junho 2006.

BELL, D.; IBM. **"UML *basics: An introduction to the Unified Modeling Language*". Article-23/05/03** Junho 2003. Disponível em: http://www.ibm.com/developerworks/rational/library/769.html. Acesso em: 2, março 2006.

BOLDT, J.; ***Meta Object Facility* (MOF) *Specification*. *Version* 1.3, *March* 2000.** Disponível em: http://www.omg.org/cgi-bin/doc?formal/00-04-03. Acesso em: 28 março, 2006.

BOOCH, G.; ***MDA: A motivated manifesto?*** Article/July 2004.Disponível em: http://www.sdmagazine.com/documents/s=9224/sdm0408a/sdm0408a.html. Acesso em 22/06/2006.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I.; **UML "Guia do usuário" (14ª EDIÇÃO).** Rio de Janeiro: Editora ELSEVIER, 2000.

CHANG, D.,T., ***Common Warehouse Metamodel* (CWM) *Specification, Version* 1.0, *February* 2001. Número do documento: ad/01-02-01**. Disponível em: http://www.omg.org/cgi-bin/doc?ad/2001-02-01. Acesso em: 07 junho, 2006.

COMPUWARE. **" MDA *tools study shows strenghths of OptimalJ", WhitePaper*: 12/05, *December* 2005.** Disponível em: http://www.compuware.com/dl/mdatools.pdf. Acesso em: 21, junho 2006.

CORBA; HEATON, L.; ***Common Object Request Broker Architecture - for embedded.* May 2006. Número do Documento: ptc/06-05-01.** Disponível em: http://www.omg.org/cgi-bin/doc?ptc/2006-05-01. Acesso em: 11 junho, 2006.

DSTC (*Cooperative Research Centre for Enterprise Distributed Systems Technology). **MetaObjectFacility*(MOF) *Specification. Version* 1.4, *October* 2001.** Disponível em: http://www.dstc.edu.au/Research/Projects/MOF/rtf1.4/ptc-2001-10-04.pdf. Acesso em: 28 março, 2006.

FOWLER, M.; KOBRYN, C.; BOOCH, G.; JACOBSON, I.; RUMBAUGH, J.; **UML *DISTILLED "A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGVAGE"*(3ª *EDITION*).** Boston: Pearson Education, Inc, 2004.

FOWLER, M.; KOBRYN, C.; BOOCH, G.; JACOBSON, I.; RUMBAUGH, J.; **UML ESSENCIAL "Um breve guia para a linguagem padrão de modelagem de objetos" (3ª EDIÇÃO).** Porto Alegre: Editora BOOKMAN, 2005.

HEATON, L; CWM ***Metadata Interchange Patterns Specification, March* 2004. número do documento: formal/04-03-25.** Disponível em: http://www.omg.org/cgi-bin/doc?formal/04-03-25. Acesso em: 07 junho, 2006.

HEATON, L.; **MOF2 Versioning *Final AdoptedSpecification*, Número do documento: ptc/05-08-01**, ***August* 2005.** Disponível em: http://www.omg.org/cgi-bin/doc?ptc/2005-08-01. Acesso em: 01 junho, 2006.

IBM DATABASE TECHNOLOGY INSTITUTE; CHANG, D, T, **CWM:*Model Driven Architecture*, FORUM/TOKYO, *March* 2001.** Disponivel em: http://www.cwmforum.org/cwmMDA_foilset.pdf. Acesso em: 07 junho, 2006.

ISO/IEC; **Meta Object Facility (MOF) *Specification*, Número do documento: ISO/IEC 19502, *July* 2005.** Disponível em: http://www.omg.org/cgi-bin/doc?formal/05-05-05. Acesso em: 01 junho 2006

LARMAN, C.; **Utilizando UML e padrões : uma introdução à análise e ao projeto orientado a objetos**. Porto Alegre: Editora BOOKMAN, 2002.

MDA (*Model Drinven Architecture*)**. *Model Driven Architecture - WhitePaper* Número do documento: omg/00-11-05**. Disponível em: http://www.omg.org/cgi-bin/doc?omg/00-11-05. Acesso em: 3 abril, 2006.

MELLOR, S., J.; SCOTT, K.; UHL, A.; WEISE, D.; **MDA Destilada "Princípios de arquitetura Orientada por Modelos" (1ª EDIÇÃO)**. Rio de Janeiro: Editora Ciência Moderna Ltda., 2005.

MUKERJI, J.; MILLER, J.; ***MDA Guide version 1.0.1***. **Número documento ormsc/2003-06-01.** Disponível em: http://www.omg.org/cgi-bin/doc?omg/03-06-01 . Acesso em: 3 abril, 2006.

MUKERJI, J.; ***Architecture Board ORMSC1.* Número do documento ormsc /2001-07-01**. Disponível em: http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01. Acesso em: 3 abril, 2006.

OLAP. **ORACLE - *Developer's Guide to the OLAP API*. Número do documento: B10335-02. *December* 2003.** Disponível em: http://www.stanford.edu/dept/itss/docs/oracle/10g/olap.101/b10335.pdf. Acesso em: 22, junho 2006.

OMG (*Object Management Group*), **Site official da OMG.** Disponível em: http://www.omg.org/. Acesso em: 28 março, 2006.

QUATRANI, T.; ***Introduction to the Unified Modeling Language.*** RATOPNAL/06-11-2003. Disponível em: ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/intro_rdn.pdf. Acesso em: 2 março, 2006.

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.; LORENSEN, W.; **Modelagem e Projetos Baseados em Objetos (1ª EDIÇÃO).** Rio de Jeneiro: Editora Campus, 1994.

SIEGEL, J; OMG. ***Developing in OMG's Model-Driven Architecture*.** WhitePaper. Dezember/2001. Disponível em: ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf. acesso em 26 agosto, 2006.

SPARX SYSTEM; **Enterprise Architect – Version 6.5 – User Guide.** SPARX/September 2006. Disponível em: http://www.sparxsystems.com.au/bin/EAUserGuide.pdf. Acesso em 15 Setembro, 2006.

UML; OMG, **Site official da UML.** Disponível em: http://www.uml.org. Acesso em: 2 março, 2006.

ANEXOS

A REQUISITOS FUNCIONAIS E NÃO FUNCIONAIS

- **Requisitos Funcionais**

A1. Controle de Cadastros

1. O sistema deve permitir a inclusão, alteração e exclusão de clientes do consultório, contendo os seguintes atributos: código do cliente, nome, documentos de identificação (RG e CPF), endereço, cidade, estado, telefone, e-mail e data de nascimento.

2. O sistema deve permitir a inclusão, alteração e exclusão das opções de pagamento dos serviços prestados pelo consultório, que são os seguintes: 1) à vista (em dinheiro ou cheque); 2) faturado em “X” parcelas de acordo com o contratante e o cliente.

3. O sistema deve permitir a inclusão, alteração e exclusão de serviços prestados.

A2. Controle de Reservas

4. O sistema deve permitir a reserva de horários para consultas e também possibilitar a exclusão das mesmas. Cada reserva possui os seguintes atributos: identificação do cliente, data e hora da consulta e previamente o término da consulta.

A3. Controle de Consultas de Pacientes

5. Após a reserva o sistema deve permitir fazer um controle da consulta, contendo os dados dos serviços prestados, nome do cliente, data e hora. Além disso, devem-se gerar as faturas, de acordo com a forma de pagamento solicitada pelo cliente.

A4. Controle de Pagamentos

6. O sistema deve permitir a quitação de uma fatura paga pelo cliente, contendo as seguintes informações: numero da fatura, data de vencimento, data de pagamento, valor total pago.

A5. Controle de Relatórios

7. O sistema deve permitir a impressão de uma listagem de todos os clientes do consultório que não fazem consultas há seis (6) meses atrás, contendo o código do cliente, nome, ultima data de consulta.

8. O sistema deve permitir a impressão de todas as consultas agendadas no mês corrente, contendo o código do cliente, nome, data e hora da consulta.

9. O sistema deve permitir a impressão do comprovante de pagamento das faturas pagas pelo cliente, contendo o nome do cliente, data de pagamento, valor total pago.

10. O sistema deve permitir a impressão para o cliente da próxima consulta agendada para o mesmo, contendo o nome do cliente, data e hora da consulta.

11. O sistema deve permitir a consulta de clientes contendo todos os dados do mesmo e também mostrar se o cliente possui algum débito pendente.

12. O sistema deve permitir a consulta de horários disponíveis e horários já marcados, caso o horário esteja marcado o sistema deverá dar as informações da reserva da consulta, contendo o código do cliente, nome, data e hora da consulta.

- **Requisitos não Funcionais**

A6. Confiabilidade

13. O sistema deve fornecer realizações de *backups* dos arquivos do sistema.

14. O sistema deve possuir uma senha de acesso para a identificação de usuário.

A7. Portabilidade

15. O sistema deve ser executado em computadores Pentium 500mHz ou superior, com sistema operacional Windows 98 ou acima.

B. CÓDIGOS FONTES

Um exemplo do Código em C# após a geração automática:

Classe Agenda

```
using Class Model.Diagrama de Classe PSM;
namespace Class Model.Diagrama de Classe PSM {
	public class Agenda {
		private int id_agenda;
		private int dt_consulta;
		private int hr_consulta;
		private Consulta m_Consulta;

		public Agenda(){
		}
		~Agenda(){
		}
		public virtual void Dispose(){
		}
		public Consulta Consulta{
			get{
				return m_Consulta;
			}
			set{
				m_Consulta = value;
			}
		}
		public int Id_agenda{
			get{
				return id_agenda;
			}
			set{
				id_agenda = value;
			}
		}
		public int Dt_consulta{
			get{
				return dt_consulta;
			}
			set{
				dt_consulta = value;
			}
		}
		public int Hr_consulta{
			get{
				return hr_consulta;
			}
			set{
```

```
                    hr_consulta = value;
                }
            }
            public void cadastrar_agenda(){
            }
            public void alterar_agenda(){
            }

            public void remover_agenda(){
            }
            public void consulta_agenda_nome(){
            }
            public void relatorio_agenda_nome(){
            }
            public void consulta_agenda_geral(){
            }
            public void consulta_agenda_dia(){
            }
            public void consulta_agenda_mes(){
            }
            public void relatorio_agenda_mes(){
            }
            public void relatorio_agenda_semana(){
            }
            public void relatorio_agenda_dia(){
            }
            public void relatorio_agenda_ano(){
            }
            public void consulta_agenda_semana(){
            }
        }//end Agenda
    }//end namespace Diagrama de Classe PSMusing Class Model.Diagrama de Classe PSM;
```

Classe Cliente

```
using Class Model.Diagrama de Classe PSM;
namespace Class Model.Diagrama de Classe PSM {
    public class Cliente {
        private readonly int cd_cliente;
        private readonly int nr_cpf;
        private string nr_rg;
        private string ds_nome;
        private string ds_endereco;
        private string ds_cidade;
        private string ds_estado;
        private int dt_nascimento;
        private int nr_telefone;
        private string ds_email;
```

```
private Agenda m_Agenda;
private Pagamento m_Pagamento;

public Cliente(){
}
~Cliente(){
}
public virtual void Dispose(){
}
public Agenda Agenda{
        get{
                return m_Agenda;
        }
        set{
                m_Agenda = value;
        }
}
public Pagamento Pagamento{
        get{
                return m_Pagamento;
        }
        set{
                m_Pagamento = value;
        }
}
public int Cd_cliente{
        get{
                return cd_cliente;
        }
        set{
                cd_cliente = value;
        }
}
public int Nr_cpf{
        get{
                return nr_cpf;
        }
        set{
                nr_cpf = value;
        }
}
public string Nr_rg{
        get{
                return nr_rg;
        }
        set{
                nr_rg = value;
        }
}
public string Ds_nome{
```

```
        get{
                return ds_nome;
        }
        set{
                ds_nome = value;
        }
}
public string Ds_endereco{
        get{
                return ds_endereco;
        }
        set{
                ds_endereco = value;
        }
}
public string Ds_cidade{
        get{
                return ds_cidade;
        }
        set{
                ds_cidade = value;
        }
}
public string Ds_estado{
        get{
                return ds_estado;
        }
        set{
                ds_estado = value;
        }
}
public int Dt_nascimento{
        get{
                return dt_nascimento;
        }
        set{
                dt_nascimento = value;
        }
}
public int Nr_telefone{
        get{
                return nr_telefone;
        }
        set{
                nr_telefone = value;
        }
}
public string Ds_email{
        get{
                return ds_email;
```

```
                }
                set{
                        ds_email = value;
                }
        }
        public void cadastrar_cliente(){
        }
        public void alterar_cliente(){
        }
        public void remover_cliente(){
        }
        public void consulta_cliente_nome(){
        }
        public void relatorio_cliente_geral(){
        }
        public void Consulta_cliente_divida_ativa(){
        }
        public void relatorio_cliente_nome(){
        }
        public void relatorio_cliente_tempo(){
        }
    }//end Cliente
}//end namespace Diagrama de Classe PSM
```

Um exemplo do Código em Java após a geração automática:

Classe Agenda

```
package Class Model.Diagrama de Classe PSM;
/**
 * @author Thiago
 * @version 1.0
 * @created 05-nov-2006 14:56:48
 */
public class Agenda {
    private int id_agenda;
    private int dt_consulta;
    private int hr_consulta;
    private Consulta m_Consulta;

    public Agenda(){
    }
    public void finalize() throws Throwable {
    }
    public Consulta getConsulta(){
        return m_Consulta;
    }
    public void setConsulta(Consulta newVal){
        m_Consulta = newVal;
    }
```

```
	public int getId_agenda(){
		return id_agenda;
	}
	public void setId_agenda(int newVal){
		id_agenda = newVal;
	}
	public int getDt_consulta(){
		return dt_consulta;
	}
	public void setDt_consulta(int newVal){
		dt_consulta = newVal;
	}
	public int getHr_consulta(){
		return hr_consulta;
	}
	public void setHr_consulta(int newVal){
		hr_consulta = newVal;
	}
	public void cadastrar_agenda(){
	}
	public void alterar_agenda(){
	}
	public void remover_agenda(){
	}
	public void consulta_agenda_nome(){
	}
	public void relatorio_agenda_nome(){
	}
	public void consulta_agenda_geral(){
	}
	public void consulta_agenda_dia(){
	}
	public void consulta_agenda_mes(){
	}
	public void relatorio_agenda_mes(){
	}
	public void relatorio_agenda_semana(){
	}
	public void relatorio_agenda_dia(){
	}
	public void relatorio_agenda_ano(){
	}
	public void consulta_agenda_semana(){
	}
}
```

Classe Cliente

```
public class Cliente {
	private final int cd_cliente;
	private final int nr_cpf;
	private String nr_rg;
	private String ds_nome;
	private String ds_endereco;
	private String ds_cidade;
	private String ds_estado;
	private int dt_nascimento;
	private int nr_telefone;
	private String ds_email;
	private java.util.ArrayList m_Agenda;
	private java.util.ArrayList m_Pagamento;

	public Cliente(){
	}
	public void finalize() throws Throwable {
	}
	public java.util.ArrayList getAgenda(){
		return m_Agenda;
	}
	public void setAgenda(java.util.ArrayList newVal){
		m_Agenda = newVal;
	}
	public java.util.ArrayList getPagamento(){
		return m_Pagamento;
	}
	public void setPagamento(java.util.ArrayList newVal){
		m_Pagamento = newVal;
	}
	public int getCd_cliente(){
		return cd_cliente;
	}
	public void setCd_cliente(int newVal){
		cd_cliente = newVal;
	}
	public int getNr_cpf(){
		return nr_cpf;
	}
	public void setNr_cpf(int newVal){
		nr_cpf = newVal;
	}
	public String getNr_rg(){
		return nr_rg;
	}
	public void setNr_rg(string newVal){
		nr_rg = newVal;
```

```
}
public String getDs_nome(){
        return ds_nome;
}
public void setDs_nome(string newVal){
        ds_nome = newVal;
}
public String getDs_endereco(){
        return ds_endereco;
}
public void setDs_endereco(string newVal){
        ds_endereco = newVal;
}
public String getDs_cidade(){
        return ds_cidade;
}
public void setDs_cidade(string newVal){
        ds_cidade = newVal;
}

public String getDs_estado(){
        return ds_estado;
}
public void setDs_estado(string newVal){
        ds_estado = newVal;
}
public int getDt_nascimento(){
        return dt_nascimento;
}
public void setDt_nascimento(int newVal){
        dt_nascimento = newVal;
}
public int getNr_telefone(){
        return nr_telefone;
}
public void setNr_telefone(int newVal){
        nr_telefone = newVal;
}
public String getDs_email(){
        return ds_email;
}
public void setDs_email(string newVal){
        ds_email = newVal;
}
public void cadastrar_cliente(){
}
public void alterar_cliente(){
}
public void remover_cliente(){
}
```

```
        public void consulta_cliente_nome(){
        }
        public void relatorio_cliente_geral(){
        }
        public void Consulta_cliente_divida_ativa(){
        }
        public void relatorio_cliente_nome(){
        }
        public void relatorio_cliente_tempo(){
        }
}
```