Universidade do Estado da Bahia

BRUNO MORAIS LEAL

INTEGRANDO MDA COM MÉTODOS ÁGEIS: VANTAGENS E DESVANTAGENS

Salvador

2014

Bruno Morais Leal

INTEGRANDO MDA COM MÉTODOS ÁGEIS: VANTAGENS E DESVANTAGENS

Trabalho de Conclusão de Curso submetida ao Colegiado de Sistema de Informação da Universidade do Estado da Bahia como parte dos requisitos necessários para obtenção do grau de Bacharel em Sistema de Informação.

Sistema de Informação
Desenvolvimento Dirigido a Modelos e Métodos Ágeis

Orientadora: Prof.ª Ana Patrícia F.Magalhães

Salvador
2014

Resumo

A busca por produtividade e qualidade é uma realidade no desenvolvimento de software. Neste contexto estão inseridas as abordagens de desenvolvimento ágil e desenvolvimento dirigido a modelos. O desenvolvimento ágil dá ênfase à fase de codificação do sistema. Desta forma, a especificação do sistema fica em segundo plano, o software é desenvolvido rapidamente em pequenas versões que são disponibilizadas ao cliente. Já o Desenvolvimento Dirigido a Modelos (DDM) considera os modelos os principais artefatos do desenvolvimento, artefatos indispensáveis para geração de código da aplicação.

A integração dessas duas abordagens a primeira vista parece inviável, porém abordagens MDD Ágil estão sendo propostas para agilizar o processo de desenvolvimento na abordagem MDD. Este trabalho analisa a viabilidade dessa integração apontando benefícios e problemas de se utilizar o MDD Ágil através de uma ferramenta que adeque os processos de desenvolvimento dirigido a modelos com as definições de práticas ágeis.

Para avaliar a integração foi desenvolvido um sistema de vendas de impressoras na abordagem MDD Ágil. A avaliação mostrou que é possível integrar processos pertencentes ao MDD com algumas práticas ágeis. Porém, para que haja sucesso na integração ainda é preciso desenvolver modelos capazes de gerar o código do sistema em perfeito estado.

Palavras Chaves: Desenvolvimento Ágil, Desenvolvimento Dirigido a Modelos, MDD Ágil

Abstract

The pursuit for productivity and quality is a reality in software development. In this context, are inserted approaches for agile development and direct development models. Agile development emphasizes the coding phase of the system. Thus, the system specification is left in background, the software is developed rapidly in smaller versions and are made available to the customer. By the side the Model Driven Development (MDD) considers models the main development artifacts and it's the key of generating the application code.

The integration of these two approaches at first glance seems unfeasible, however Agile MDD approaches are being proposed to speed up the development process in the MDD approach. This study examines the feasibility of this integration pointing benefits and problems of using Agile MDD through a tool that fits the models driven development processes with the definitions of agile practices.

To assess the integration, a printer selling system was developed based on the agile MDD approach. The assessment showed that it is possible to integrate processes belonging to MDD with some agile practices. However, to have a successful integration it's still a need to develop models capable of generating the code of the system in perfect condition.

Key Words: Agile Development; Models Driven Development; Agile MDD.

Agradecimentos

Agradeço principalmente aos meus pais Ednaldo de Oliveira Leal e Ana Suely Lopes Morais Leal que nunca mediram esforços para proporcionarem a minha felicidade, ao meu irmão por sempre estar ao meu lado me ajudando no que for preciso, aos meus familiares e amigos por me apoiarem sempre nas decisões difíceis da minha vida e a minha orientadora por sempre me guiar nas atividades que deram suporte a este Trabalho de Conclusão de Curso.

SUMÁRIO

1 INTRODUÇÃO 10
2 FUNDAMENTAÇÃO TEÓRICA 13
2.1 Desenvolvimento Dirigido a Modelos 13
2.1.1 Conceitos envolvidos no DDM 14
2.1.2 Model Driven Architecture 16
2.1.3 Ferramentas MDA 18
2.1.3.1 *Moderne* 19
2.1.3.2 *OptmalJ* 22
2.1.3.3 *AndroMDA* 24
2.1.3.4 *Enterprise Architect* 25
2.2 Métdos Ágeis 31
2.3 Framework de Práticas Ágeis 32
2.3.1 Estrutura do Framework de Práticas Ágeis 33
2.3.1.1 Atividade de Definição de Requisitos 33
2.3.1.2 Atividades de Atribuição de Requisitos as Iterações 34
2.3.1.3 Atividades do Projeto de Arquitetura do Sistema 35
2.3.1.4 Atividades de Desenvolvimento do Incremento do Sistema 36
2.3.1.5 Atividades de Validação do Incremento 38
2.3.1.6 Atividade de Integração do Incremento 39
2.3.1.7 Atividades de Validação do Sistema 39
2.3.1.8 Atividades de Entrega Final 39
2.4 Modelos Executáveis 40
2.5 MDA Ágil 41
2.6 Trabalhos Correlatos 42
2.6.1 Agile Model driven Development in Practice 42
2.6.2 Um Relato de Experiência no Desenvolvimento Ágil de Sistemas com MDA 43
2.6.3 Uso de MDA em um Framework para seleção de Práticas Ágeis 44
3 EXEMPLO DE USO 45
3.1 Cenário escolhido para exemplo de uso 45
3.2 Casos de Uso da aplicação 46
3.3 Primeira Iteração 48
3.4 Segunda Iteração 52
3.5 Terceira Iteração 56
3.6 Pesquisa para avaliação da abordagem 61
3.7 Considerações Finais 62
4 CONCLUSÃO 64
REFERÊNCIAS 66
ANEXO 1 68

Lista de Símbolos

AM – Agile Modeling

ASD – Adaptive Software Development

ATL – Atlas Transformation Language

CIM – Computation Independent Model

DDM – Desenvolvimento Dirigido a Modelos

EA – Enterprise Architect

FDD – Feature Driven Development

MDA – Model Driven Architecture

MDD – Model Driven Development

MDM – Model-to-Model

MOF – Meta-Object Facility

M2C – Model-to-Code

OMG – Object Management Group

PIM – Plataform Independent Model

PSM – Plataform Specific Model

SPEM – Software Process Engineering Metamodel Specification

TDM – Model Driven Test

UML – Unified Modeling Language

WCT – Work Case ToolKit

XP – Extreme Programming

Lista de Figuras

1. Visão geral da MDD 13
2. Exemplo de transformação de Refatoração 15
3. Exemplo de transformação model-to-model 15
4. Exemplo de transformação modelo para código 16
5. Visões da MDA 17
6. Visão Geral da Moderne 20
7. Processo Editor 21
8. Processo Executor 22
9. Etapa para transformação MDA da ferramenta 26
10. Transformações PSMs a partir do modelo PIM 27
11. Etapa para geração do código usando EA 28
12. Modelo específico da plataforma java gerado pela EA 28
13. Código do PSM gerado pela EA 29
14. Template de Transformação Enterprise Architect 30
15. Casos de Uso do Sistema 47
16. Modelo PIM Primeira Iteração 49
17. Modelo PSM Primeira Iteração 50
18. Código gerado pela Enterprise Architect 51
19. Modelo PIM Segunda Iteração 53
20. Modelo PSM Segunda Iteração 54
21. Código da Base de Dados usada na Aplicação 55
22. Diagrama de Estados da Venda 56
23. Modelo PIM Terceira Iteração 57
24. Código da Classe F_Venda gerado pela EA 58
25. Código do Diagrama de Estados gerado pela EA 59
26. Tela de Cadastro da Venda 61

Lista de Tabelas

1. Primeira Iteração da Aplicação Usando MDD Ágil 52

2. Segunda Iteração da Aplicação Usando MDD Ágil 55

3. Terceira Iteração da Aplicação Usando MDD Ágil 60

1 INTRODUÇÃO

Os avanços tecnológicos aumentaram a demanda por softwares e as empresas vêm buscando cada vez mais produtividade no desenvolvimento dos seus sistemas. A necessidade de criação de softwares de forma mais rápida sem comprometer a qualidade a um custo menor, intencionando-se à boa utilização dos recursos computacionais e novas tecnologias é uma grande preocupação das indústrias de software (BORGES, SOUZA, SCHULZE, MURY, 2011).

Nesse contexto, surgiram métodos de desenvolvimento de sistemas que possibilitam o aumento da produtividade, como o desenvolvimento dirigido a modelos (DDM) e os métodos ágeis.

O DDM consiste em um estilo de desenvolvimento de software onde os principais artefatos de software são modelos, a partir dos quais são gerados o código e outros artefatos de acordo com as práticas de desenvolvimento do sistema. Um modelo é uma descrição de um sistema (expressos em UML) a partir de uma determinada perspectiva, omitindo detalhes irrelevantes para que as características de interesse sejam vistas de forma mais clara (NEWSLETTER, 2006). O objetivo do DDM é reduzir a distância semântica existente entre o domínio do problema e o domínio da implementação/solução, utilizando modelos mais abstratos que protegem os desenvolvedores de software das complexidades inerentes as plataformas de implementação.

A principal vantagem do DDM é poder expressar modelos usando conceitos menos vinculados a detalhes de implementação (NETO, FORTES, 2012). Isso garante uma maior produtividade e portabilidade no desenvolvimento do software, onde artefatos e códigos são gerados a partir desses modelos independente da tecnologia adotada na aplicação. Além disso, os modelos produzidos melhoram a comunicação com os stakeholders envolvidos no projeto, omitindo detalhes na implementação que não são relevantes para a compreensão do comportamento lógico de um sistema (NEWSLETTER, 2006).

Um modelo ágil consiste em um modelo que atenda seu propósito e seja suficientemente preciso, consistente, detalhado e tão simples o quanto possível. O objetivo dos métodos ágeis

é minimizar o risco pelo desenvolvimento do software em curtos períodos, chamados de iteração, os quais gastam um tempo de um a quatro semanas para seu desenvolvimento. Cada iteração é como um projeto de software em miniatura de seu próprio, e inclui todas as tarefas necessárias para implantar o mini-incremento da nova funcionalidade: planejamento, análise de requisitos, projeto, codificação, teste e documentação. Um projeto de software ágil busca a capacidade de implantar uma nova versão do software ao fim de cada iteração.

Nesse cenário, surge o conceito de *Agile Model Driven Development*, que é a versão ágil do Desenvolvimento Orientado a Modelos, onde no início do projeto é investido algum tempo na modelagem do sistema voltada, particularmente, para explorar os requisitos fundamentais e identificar uma grande abordagem ao nível de arquitetura. A integração dessas duas metodologias de desenvolvimento ainda apresenta problemas na definição dos modelos necessários para que seja possível usar DDM para gerar o código de aplicação.

Este trabalho tem como objetivo analisar a viabilidade da integração do Desenvolvimento Dirigido a Modelos e Métodos Ágeis apontando os benefícios e problemas da integração. Para isso pretende-se identificar as práticas ágeis, analisar a abordagens Ágil, MDD e MDA Ágil com relação às práticas ágeis, implementar um exemplo de sistema utilizando técnicas MDD e práticas ágeis e elencar benefícios e problemas que o MDD pode trazer para os métodos ágeis.

Os métodos ágeis são muito utilizados porque o mercado de software demanda produtividade no seu desenvolvimento. Porém algumas vezes os sistemas não são implementados como o cliente recomendou. Além disso, a falta de documentação necessária causa empecilhos na manutenção do sistema (KEFFER, 2002). Isso pode ocasionar problemas durante o desenvolvimento do projeto como: renovação da equipe do projeto; o conhecimento do sistema se encontra no código do projeto; dificuldade no entendimento do domínio do problema entre os stakeholders; dificuldade em evoluir e manter o projeto.

Portanto, iniciativas estão sendo propostas para integrar a abordagem Ágil e MDD com o intuito de encontrar melhorias no desempenho da produção e qualidade do software. Este trabalho ajuda a entender como desenvolver uma aplicação usando conceitos do MDD Ágil, utilizando processos que compõe o desenvolvimento dirigido a modelos e práticas de um framework ágil.

Para elaboração do trabalho inicialmente foi realizada uma pesquisa bibliográfica para compreensão do assunto, trazendo os conceitos sobre as abordagens MDD, Métodos Ágeis, Framework de Práticas Ágeis, Modelos Executáveis e MDA Ágil. Em seguida foram analisadas as ferramentas que integram técnicas de desenvolvimento MDA para utilizar na integração das duas abordagens, definição de um estudo de caso para ser desenvolvida como exemplo, implementação desse estudo de caso usando técnicas da abordagem MDA com práticas ágeis e análise dos resultados da implementação, identificando benefícios e problemas da integração.

O resto do texto está organizado da seguinte maneira: o capítulo 2 apresenta a fundamentação teórica, reunindo conceitos das abordagens fundamentais para elaboração do trabalho (DDM, MDA, ferramentas MDA, Métodos Ágeis e Framework de Práticas Ágeis) e os trabalhos relacionados a este projeto; o capítulo 3 traz o exemplo de uso necessário para integrar as abordagens MDD e Métodos Ágeis, definindo o cenário escolhido para desenvolvimento da aplicação, os casos de uso da aplicação, as iterações necessárias para desenvolvimento do sistema e as considerações finais com os resultados obtidos após implementação; por fim, o capítulo 4 apresenta a conclusão do projeto.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos básicos das duas abordagens tratadas neste trabalho: Desenvolvimento Dirigido a Modelos e Métodos Ágeis.

2.1 Desenvolvimento Dirigido a Modelos

O desenvolvimento orientado a modelos, ou Model Driven Development (MDD), é um estilo de desenvolvimento de software em que os principais artefatos de software são modelos, a partir dos quais são gerados o código e outros artefatos de acordo com as boas práticas adotadas. A Figura 1 mostra uma visão geral de como funciona o desenvolvimento dirigido a modelos. Modelos mais abstratos são construídos representando uma visão do sistema. Estes modelos são então transformados em modelos menos abstratos até que o código da aplicação é gerado. A diferença entre os Modelos Profissionais (*Fachliche Modelle*) e os Modelos Técnicos (*Techische Modelle*) está na definição da tecnologia que será adotada na transformação. Detalhes dessa tecnologia são incorporados aos Modelos Técnicos e serão responsáveis para conhecer a plataforma alvo da aplicação. Na figura 1 são descritas ao lado esquerdo as etapas de montagem dos modelos (*Modellieren*), transformação (*Transformieren*) desses modelos em modelos completos para serem executados e geração (*Generieren*) do código a partir dos mesmos.

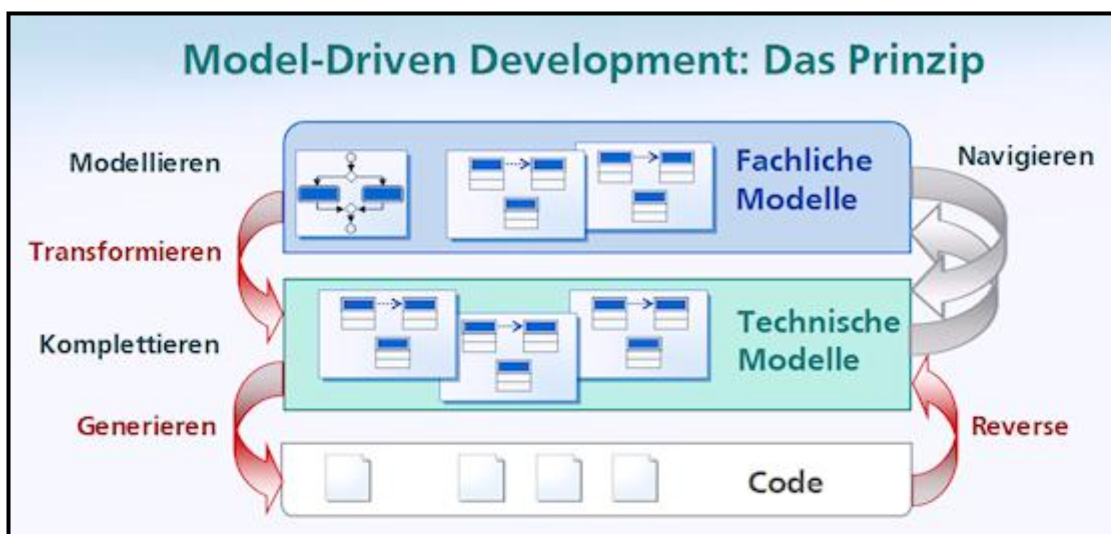


Figura 1: Visão geral da MDD

Fonte: Model Driven Development zum Mitmachen (2009)

O uso de MDD traz algumas vantagens no desenvolvimento de software. Entre elas as principais são produtividade de desenvolvimento, que se caracteriza pela automatização da geração de código a partir de modelos através do uso de ferramentas de transformação e portabilidade dos modelos, onde a partir de um mesmo modelo pode-se gerar código para diferentes plataformas.

2.1.1 Conceitos envolvidos no DDM

No DDM, um modelo é visto como uma descrição de um sistema a partir de uma determinada perspectiva, omitindo detalhes irrelevantes para que as características de interesse sejam vistas de forma mais clara. (NEWSLETTER, 2006).

Os modelos e modelagem em desenvolvimento de software têm sido utilizados como uma forma de auxiliar os desenvolvedores desde o início da programação (NEGRE, 2014). É possível a criação de vários tipos de modelos, dependendo somente do objetivo do desenvolvedor: para estabelecer um claro entendimento do problema a ser tratado; para comunicar uma visão compartilhada do problema; para analisar aspectos chaves da solução.

Um modelo é descrito de acordo com uma linguagem de modelagem. Por exemplo, a linguagem UML é uma linguagem de modelagem de sistemas de propósito geral. No entanto para trabalhar com DDM é preciso que os modelos sejam mais expressivos, permitindo geração de código ao final da cadeia. Desta forma, linguagens específicas de domínio (DSLs) são utilizadas como linguagem de modelagem.

Modelos abstratos são transformados em modelos menos abstratos através de uma cadeia de transformação. Uma transformação mapeia elementos de uma linguagem fonte em elementos de uma linguagem alvo. Por exemplo, modelos independentes da plataforma, como diagrama de classes, gerando modelos específicos da plataforma Java. Esses conceitos serão detalhados mais adiante, no decorrer do projeto.

A transformação entre os modelos é uma das atividades chave do Desenvolvimento Dirigido a Modelos. Pode haver transformação de modelos dentro do mesmo nível de abstração ou entre níveis de abstração diferentes, além de haver a possibilidade de ocorrer transformação dentro

de um mesmo domínio ou de diferentes domínios. A transformação é a criação de forma automática / semi-automática de um modelo (destino) a partir de outro modelo (origem) (NEGRE, 2014).

Existem três tipos de transformação de modelos mais comuns: transformações de refatoração, transformações model-to-model e transformações de modelo para código. A primeira permite a reorganização de modelos através de uma série de padrões definidos. O modelo de origem será revisado de acordo com o que foi definido gerando um novo modelo melhorado. A Figura 2 mostra um exemplo de transformação de refatoração (NEGRE, 2014).

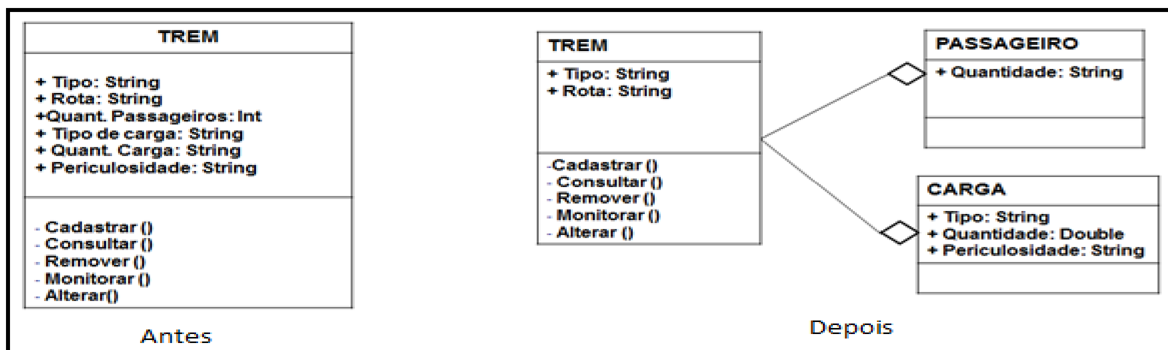


Figura 2: Exemplo de Transformação de Refatoração

Fonte: Model Driven Development – Transformação de Modelos (NEGRE, 2014)

A transformação model-to-model tem um ou mais modelos de origem que são transformados em um modelo ou grupo de modelos. A transformação pode ser realizada dentro de uma mesma notação ou para uma notação diferente do modelo de origem. A Figura 3 mostra um exemplo de transformação model-to-model (NEGRE, 2014).

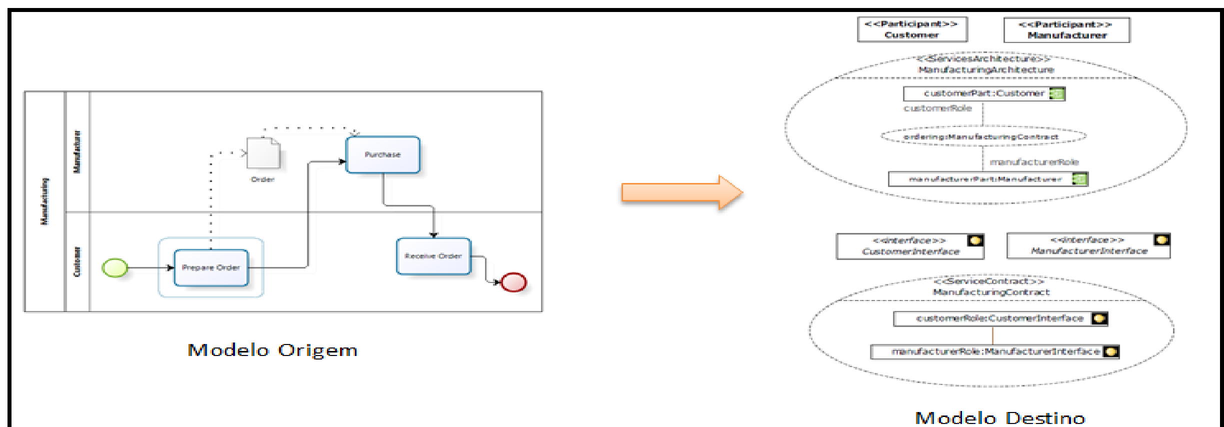


Figura 3: Exemplo de transformação model-to-model

Fonte: Model Driven Development – Transformação de Modelos (NEGRE, 2014)

A transformação de modelo para código permite transformar um modelo para o mais próximo possível de uma linguagem de programação (NEGRE, 2014). A seguir é mostrado um exemplo de transformação modelo para código.

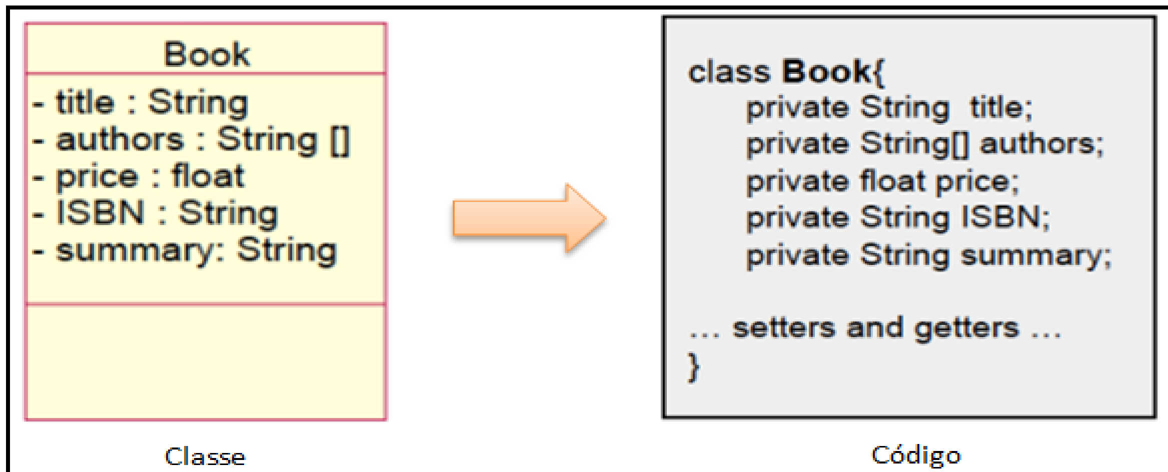


Figura 4: Exemplo de transformação modelo para código.

Fonte: Model Driven Development – Transformação de Modelos (NEGRE, 2014)

A adoção da abordagem DDM por uma empresa demanda a customização do seu processo de desenvolvimento para um processo DDM. Esta customização envolve a definição de DSLs e da cadeia de transformação de irá automatizar o processo.

2.1.2 Model Driven Architecture

A *Model Driven Architecture* é uma especificação do Desenvolvimento Dirigido a Modelos, proposta pela OMG (*Object Manegement Group*) (AGUIAR, 2012). A MDA enxerga o desenvolvimento dirigido a modelos compreendendo três visões: Modelo Independente da Computação (CIM – *Computational Independent Model*); Modelo Independente da Plataforma (PIM – *Plataform Independent Model*) e Modelo Específico da Plataforma (PSM – *Plataform Specific Model*). A figura 5 ilustra estas visões que podem ser executadas de acordo com os seguintes passos:

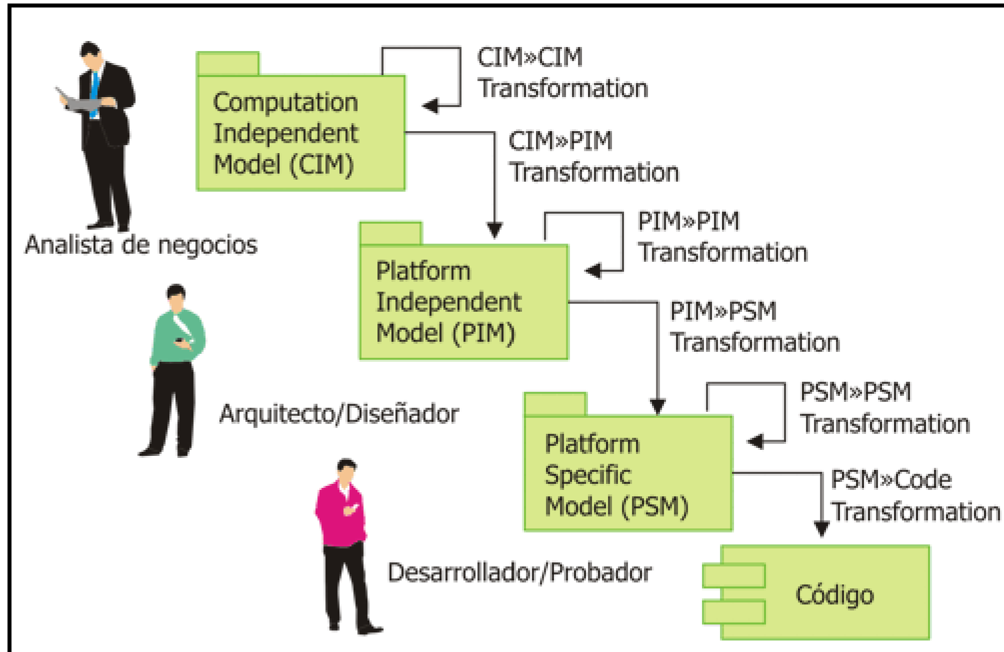


Figura 5: Visões da MDA

Fonte: Proyectos e procesos de software (2007)

- Primeiro passo é a geração de modelo *Computation Independent Model* (CIM), uma visão do sistema independente de computação. O CIM representa apenas requisitos do sistema e não mostra detalhes da sua estrutura.
- O segundo passo é a geração, a partir do CIM, do modelo chamado *Platform Independent Model* (PIM), definido como alto grau de abstração, independente de qualquer tipo de tecnologia. O PIM descreve o sistema de software de uma perspectiva que melhor represente o negócio sendo modelado.
- O terceiro passo é a geração de um ou mais *Plataform Specific Models* (PSMs). Cada PSM é gerado levando em conta detalhes específicos de uma determinada tecnologia a ser utilizada na implementação. Para cada PIM vários PSMs podem ser gerados.

Na MDA os modelos de software são tipicamente expressos em linguagem UML (*Unified Modeling Language*) enriquecida pelo uso de perfils. Perfil é o mecanismo de extensão da UML para definir DSLs. Ao criar um perfil restringe-se a UML para um domínio específico.. A UML fornece uma notação visual e semânticas subjacentes para os modelos de software. De igual modo, tem um formato padrão de serialização legível para a máquina, permitindo assim a automação. (NEWSLETTER, 2006).

Os principais benefícios que MDA proporciona no desenvolvimento do software são (NEWSLATTER, 2006), (SOUZA, 2012):

- Produtividade - Uma vez definidas as regras, as transformações se tornam automáticas, e apenas o modelo conceitual (PIM) deveria ser atualizado, enquanto que os demais modelos, inclusive a implementação, seriam automaticamente atualizados, reduzindo o tempo de desenvolvimento e aumentando a qualidade do artefato final.
- Manutenção e documentação - todo o trabalho de manutenção não é mais feito no código, mas sim no nível mais alto de abstração (o modelo PIM). Desta forma, a manutenção é facilitada e a documentação constantemente atualizada. E, no caso da manutenção ser realizada diretamente no PSM, deve ser possível atualizar automaticamente o PIM e manter os modelos consistentes.
- Interoperabilidade - é alcançada através de bridges, que realizam a comunicação entre PSMs e códigos de diferentes plataformas. Para obter interoperabilidade as ferramentas de transformação devem gerar não só os PSMs, mas também as bridges entre eles.
- Adaptabilidade - A adição ou modificação de uma função de negócio é linear, uma vez que já foi feito o investimento em automação. Quando se pretendem adicionar novas funções de negócio, desenvolvemos apenas o comportamento específico dessa capacidade. A restante informação necessária para gerar artefatos de implementação foi capturada nas transformações.

Apesar dos processos MDA trazerem esses benefícios, ainda não se encontram totalmente maduros. As linguagens (modelos) e ferramentas ainda precisam ser aperfeiçoadas, para que haja uma melhor automação nos processos. Segundo Almir Buarque (2009), especialistas em engenharia de software e a própria OMG apostam que MDD e MDA serão os processos mais utilizados pelas empresas nos próximos anos.

2.1.3 Ferramentas MDA

Para apoiar o desenvolvimento dirigido a modelos, tentativas de desenvolvimento de ferramentas vêm sendo feitas, além da adaptação de outras já existentes (AGUIAR, 2012). Algumas são ferramentas comerciais de grande porte, já conhecidas pela maioria dos desenvolvedores de software mundo afora, outras porém, podem ser resultados de esforços

individuais ou de grupos de pesquisas de construir meios apropriados para utilizar o MDA em seus processos de software (AGUIAR, 2012).

A seguir serão apresentados alguns exemplos de ferramentas de apoio ao MDA. Porém, será utilizada, a princípio, apenas uma ferramenta para realização deste trabalho. Essas ferramentas são de extrema importância para a avaliação da integração dos modelos com os frameworks de práticas ágeis, pois a partir delas podemos verificar se é possível adaptar os processos de construção do software aos frameworks de práticas ágeis.

2.1.3.1 Moderne

A Moderne (MACIEL, et.all, 2008) é uma ferramenta que a apoia a especificação e execução de processos de desenvolvimento de software que usam as abordagens MDA (*Model Driven Achitecture*) e TDM (*Model Driven Test*) para a construção e teste de software A especificação dos processos na Moderne é baseada em metamodelos que estendem os conceitos do SPEM para dar suporte a aspectos específicos de processos MDA e TDM. SPEM é um padrão proposto e mantido pelo (OMG). É um metamodelo baseado em MOF (*Meta Object Facility*) utilizada para especificar processos de software.

Os metamodelos da Moderne possibilitam representar explicitamente elementos específicos de um processo MDA, tal como as fases de modelação (CIM, PIM, PSM), perfis UML e regras de transformação. Nos processos TDM também é possível especificar artefatos do sistema tais como: casos de teste, relatório de ensaios, registros, assim como papéis e atividades utilizadas nesse tipo de processo.

De acordo com o padrão SPEM, o *MethodContent* é uma biblioteca de conceitos que podem ser reutilizados em diferentes processos e um *Process* representa um processo de software que podem utilizar diferentes elementos dessa biblioteca. Na Moderne, é possível usar bibliotecas independentes para modelar e testar seu software, cada uma possuindo seus próprios elementos (tarefas, workpoducts, papéis e disciplinas). Os elementos do processo de ensaio pertencem a uma biblioteca específica, chamada *MethodContentForTesting,* que pode ser modelada de forma independente no processo de desenvolvimento. Mais tarde, um processo instanciado pode usar elementos de ambas as bibliotecas, permitindo o desenvolvimento e teste de integração dos processos.

Um processo de software pode ser representado por diagramas ou como uma definição estrutural hierárquica. Na ferramenta, é possível gerar automaticamente diagramas de processo a partir da sua estrutura hierárquica. Um único processo pode ser controlado por diferentes usuários em máquinas distintas.

A Moderne possui dois módulos (Figura 6): o Editor de processos e o Executor, um que permite a especificação de processos e outro apoiando a sua execução. Os dois módulos se conectam através de um repositório (em Mysql) que armazena os processos especificados no Editor, executados posteriormente no Executor.

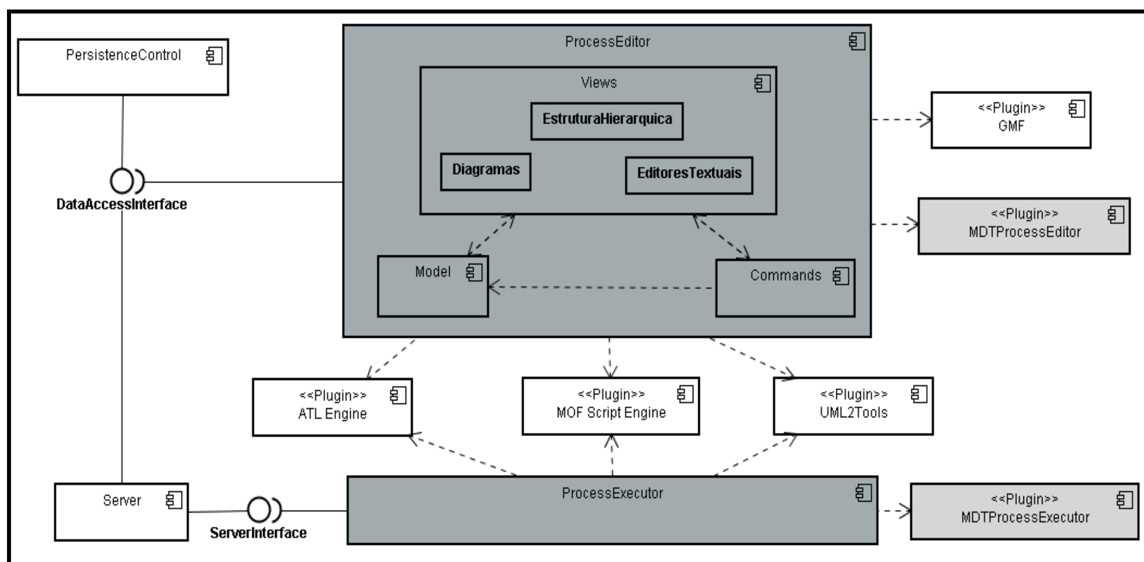


Figura 6: Visão Geral da Moderne.

Fonte: Moderne, Departamento de Ciência da Computação UFBA

O Editor de processos é um ambiente para criação e edição de processos de construção e teste de software. Ele permite a especificação dos processos através de uma estrutura hierárquica, editores aproveitados e estendidos da plataforma Eclipse, ou diagramas UML. O Editor possibilita, ainda, a criação de regras de transformações em ATL (Atlas Transformation Language) (MACIEL, 2012) ou MOFScript (transformação do modelo para o texto) (MACIEL, 2012) e de perfis UML, que serão utilizados posteriormente na execução do processo.

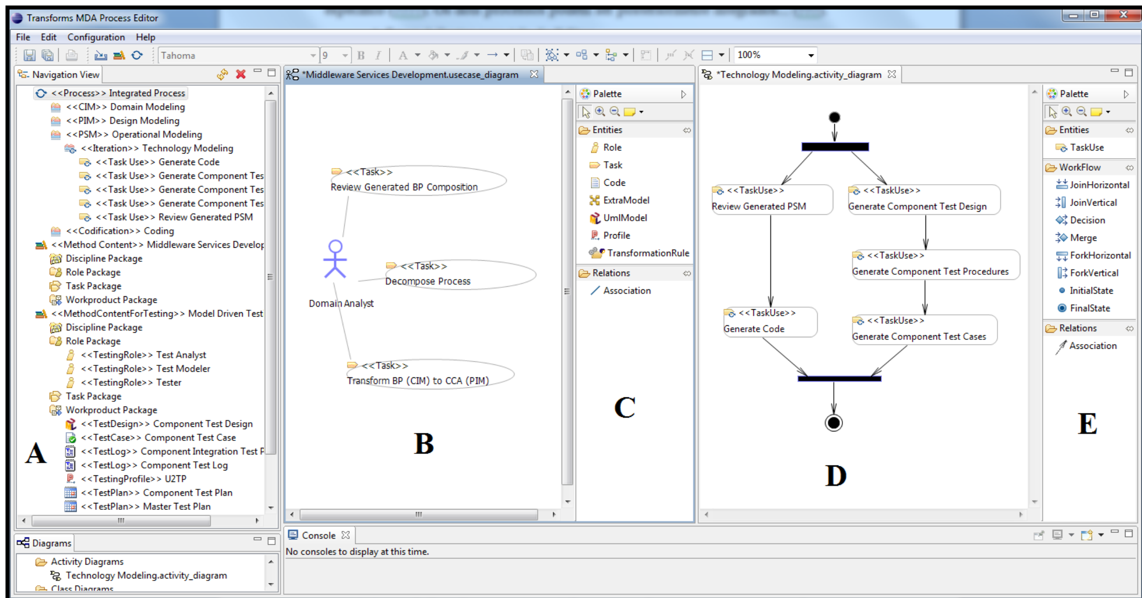


Figura 7: Processo Editor.

Fonte: Moderne, Departamento de Ciência da Computação UFBA

A Figura 7 ilustra o processo Editor dividido em cinco seções. O painel (A) apresenta um processo sob edição, chamada de processo integrado, e sua estrutura hierárquica, juntamente com seus elementos (tarefas, funções do projeto, etc.). Painéis (B) e (D) correspondem a áreas de modelagem visual, onde o usuário pode criar e editar diagramas UML. Painel (B) ilustra um diagrama de casos de uso, utilizado para atribuir responsabilidades aos vários papéis do processo. Painel (D), por outro lado, contém um diagrama de atividade, o que representa o fluxo de atividades de uma iteração de processo. Ela representa a sequência de atividades da Tecnologia de Iteração de Modelagem necessária para gerar o código a partir do modelo PSM e a aplicação de casos de teste de unidade, usando a linguagem ATL. Nos painéis (C) e (E) existem opções de ferramenta que suportam a criação de modelos de processos, instancias do SPEM/MDA e grandes metamodelos.

O Executor realiza a execução dos processos previamente especificados no Editor. Ele realiza o controle de acesso dos usuários aos artefatos e às tarefas do processo. Além disso, contém um ambiente de modelagem que permite aos usuários criar seus modelos dentro da própria ferramenta. Além disso, o Executor aplica automaticamente os perfis especificados no Editor e permite a execução de transformações dentro do seu ambiente.

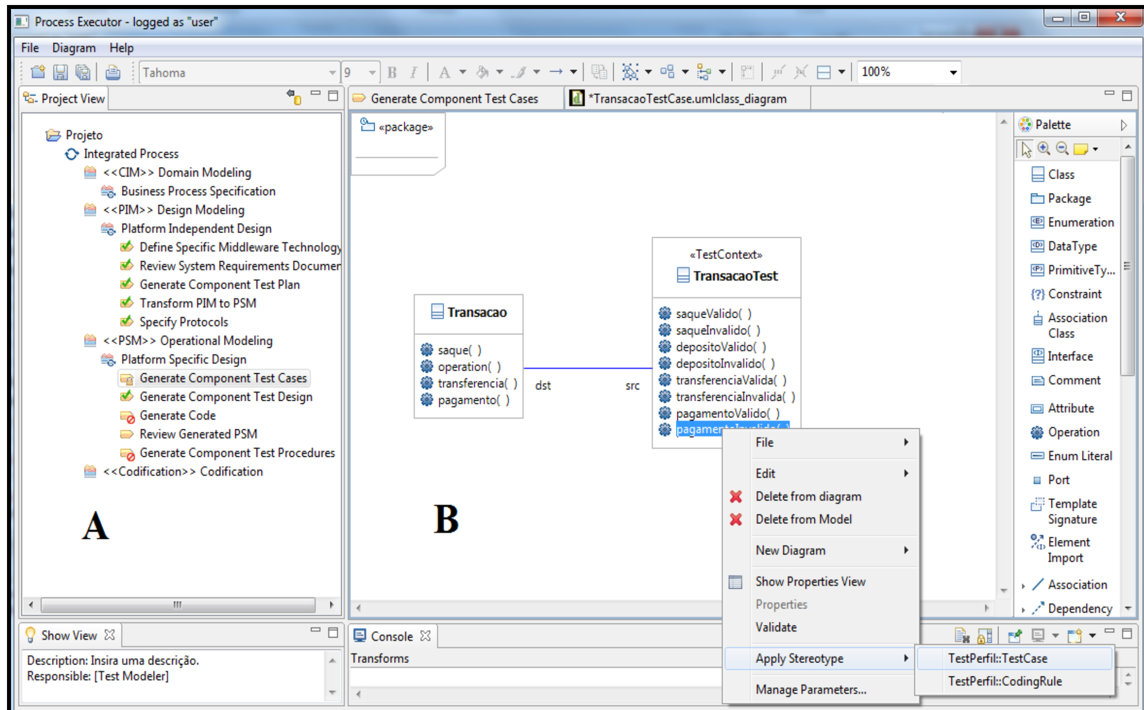


Figura 8: Processo Executor.

Fonte: Moderne, Departamento de Ciência da Computação UFBA

A figura 8 ilustra o processo executor dividido em duas seções. A parte (A) contém o processo, previamente especificado no Editor (Figura 7). Nesta área, o gerente de projeto pode atribuir funções a outros usuários, permitindo o controle de acesso às tarefas. Além disso, as tarefas têm um ícone que indica se eles estão sendo realizadas ( ), já terminaram ( ), que ainda não começou, mas está pronto para começar ( ), ou que não começou porque depende de outra tarefa que não tem terminou ainda ( ). A parte (B) consiste no ambiente de modelagem, onde o desenvolvedor pode criar os modelos de aplicação. Os perfis UML especificados no processo são automaticamente aplicados pelo executor com a criação dos diagramas.

2.1.3.2 OptimalJ

A OptimalJ utiliza um subconjunto da UML como linguagem de modelagem e o MOF (Meta-Object Facility) como meta-modelo (2006, Compuware apud Caliari, 2007). Ela divide o desenvolvimento de sistemas em três etapas, cada uma com seu modelo: Modelo de Domínio, Modelo de Aplicação e Modelo de Código.

Segundo a OptimalJ, o modelo de domínio define a funcionalidade e a estrutura, sem detalhes específicos de tecnologia. É dividido em três diagramas: um diagrama de classes para modelar a estrutura do sistema com as classes de domínio e suas associações; outro diagrama de classes para modelar a informação comportamental na forma de declarações de serviços de domínio e operações de domínio (fornecem os dados necessários aos componentes dos serviços de domínio para que implementem o comportamento desejado); e um diagrama de atividades, que modela o processo de um fluxo de trabalho na forma de atividades que podem invocar serviços de domínio, operações dos serviços de domínio, ou chamadas de processo para executar suas atividades.

O modelo seguinte é o Modelo de Aplicação, que descreve a funcionalidade, baseado em determinada tecnologia ou subsistema sendo divididos em dois módulos que representam unidades de implementação, todos representados por diagramas de classes. Os módulos incluem a camada de persistência, a de acesso aos dados, a de lógica de negócios e a camada de apresentação do sistema. A OptimalJ ainda provê um modelo de integração que possibilita a aplicação conectar-se a aplicações e componentes externos. O modelo de Aplicação adiciona não somente a tecnologia ao sistema, mas também a arquitetura. Quando o Modelo de Domínio é transformado em modelo de aplicação, uma arquitetura de camadas, como camada de apresentação e camada de negócios, é aplicada ao sistema.

Por fim, existe o modelo de código, que provê uma implementação do sistema descrito nos modelos anteriores. Contém o código fonte, descritores e scripts SQL. A OptimalJ automaticamente cria código implementando padrões de projeto, incluindo padrões de projeto J2EE, assim como padrões de projeto criados pelo usuário.

Para OptimalJ, os mapeamentos da transformação do Modelo de Domínio para o Modelo de Aplicação são feitos baseados em regras que mapeiam tipos do modelo fonte a tipos do modelo alvo. A ferramenta também disponibiliza algumas regras de mapeamento de tipos baseados em valores de instâncias dos modelos, em que os elementos são transformados de maneira diferente do mapeamento do seu tipo. Esses mapeamentos de tipos com valores de instâncias são disponibilizados como configurações que podem ser feitas em prioridades dos elementos do modelo.

Outro mecanismo de auxílio às transformações que a OptimalJ disponibiliza é a criação de novas meta-classes baseada no MOF para os modelos de domínio e aplicação. As novas meta-

classes são criadas para que possam ser utilizadas na modelagem, possibilitando a criação de outros mapeamentos.

A linguagem de mapeamento utilizada pela OptimalJ é Java. Devem ser construídos algoritmos para as regras do mapeamento em métodos Java, nos quais os parâmetros de entrada descrevem o tipo e/ou gabarito esperado do Modelo de Domínio. O retorno do método é resultado da transformação, isto é, um novo elemento do Modelo de Aplicação.

A OptimalJ permite que seja escolhida um estilo arquitetônico para o código da aplicação na criação do projeto. A escolha do estilo arquitetônico influi na transformação de PIM para PSM, determinando quais mapeamentos serão usados, para escolher os gabaritos que vão gerar o PSM. Alguns exemplos de estilo arquitetônico possíveis são: arquitetura em três camadas, arquitetura em duas camadas, arquitetura em duas camadas baseada em serviços, etc.

2.1.3.3 AndroMDA

AndroMDA é um arcabouço de código aberto que implementa transformações da MDA (2006, AndroMDA apud Caliari, 2007). Ela é dividida em núcleo e plugins chamados de cartuchos, que são adicionados ao núcleo para realizar as transformações. Cartuchos são componentes que contém um mapeamento para determinada plataforma.

A linguagem de modelagem dessa ferramenta é a UML. Ela utiliza apenas um modelo, usado como entrada para as transformações e geração Direta de código. Esse modelo é chamado de PIM do sistema, e a AndroMDA especifica que ele não pode conter qualquer tipo de dado específico de plataforma nos seus elementos, apenas de tipos de dados genéricos. O modelo utiliza diagramas de casos de uso, diagramas de classes e diagramas de atividade da UML.

Para garantir que os tipos de dados corretos sejam usados, a AndroMDA desenvolveu um perfil UML que deve ser incluído como parte do modelo. Esse perfil UML contém todos os tipos de dados, estereótipos e valores anotados genéricos que devem ser utilizados com a ferramenta e todos os seus cartuchos. O modelo deve ser construído utilizando os tipos de dados genéricos definidos no perfil UML, e os elementos do modelo podem ser anotados com os estereótipos para que eles sejam transformados de maneira diferente.

Para gerar o código a partir do PIM do sistema, o modelo deve ser usado como entrada para a ferramenta e os cartuchos necessários devem ser escolhidos. Os cartuchos contêm o mapeamento para as plataformas, então a escolha dos cartuchos deve ser baseada na plataforma na qual se quer que o sistema seja construído. Desse modo, a escolha dos cartuchos define a plataforma do sistema. Mapeamentos na AndroMDA são feitos utilizando regras de mapeamento, as quais declaram os elementos do PIM do sistema esperados como entrada na regra e os elementos ou gabaritos alvos.

As regras de mapeamento da ferramenta definem entradas baseadas em três possibilidades: tipos dos elementos, valores de propriedades ou estereótipos, e saídas baseadas em gabaritos. Os tipos e os estereótipos estão presentes no perfil UML. Os elementos do modelo de entrada são analisados pelas regras e, ao encontrar elementos cujos tipos, valores de propriedades ou estereótipos sejam os requeridos por uma determinada regra, ela é aplicada. O gabarito definido na regra e os valores do elemento de entrada são passados para um gerador de gabaritos, que irá gerar o código seguindo o padrão descrito e utilizando os valores para preencher os valores necessários do gabarito. O gerador de gabaritos Velocity acompanha a ferramenta, mas pode ser substituída por outro gerador, conforme a necessidade. Os mapeamentos são descritos em XML e interpretados pela ferramenta utilizando a linguagem Java.

O primeiro passo para utilizar o AndroMDA é criar um modelo do sistema utilizando o perfil UML que ela dispõe. Em seguida, um projeto deve ser criado, escolhendo os cartuchos e um estilo arquitetônico para o código da aplicação. O estilo arquitetônico escolhido afeta a transformação de PIM para o código, determinando os mapeamentos e gabaritos que serão usados para gerar o código. Alguns exemplos de estilos arquitetônicos possíveis são: arquitetura em três camadas, arquitetura em duas camadas, arquitetura em duas camadas baseada em serviços, etc. Por fim, o modelo, as configurações e os cartuchos devem ser utilizados como entrada para a transformação de PIM para código da ferramenta. O resultado deve ser o código esperado.

2.1.3.4 Enterprise Architect

Uma das ferramentas mais conhecidas na especificação de modelos UML, a Enterprise Architect, também tentou se adaptar à nova arquitetura dirigida a modelos, oferecendo transformações diretas entre modelos de diferentes níveis de abstração (AGUIAR, 2012). Ela

disponibiliza uma versão gratuita de 30 dias para testes. A versão completa é uma das mais robustas encontradas no mercado.

A Enterprise Architect permite que sejam criados modelos independentes de plataforma, e que estes sejam mapeados para uma grande coleção de tecnologias (AGUIAR, 2012). A transformação MDA ocorre quando o modelo é selecionado e a função *do transform MDA* da ferramenta é acionada (Figura 9).

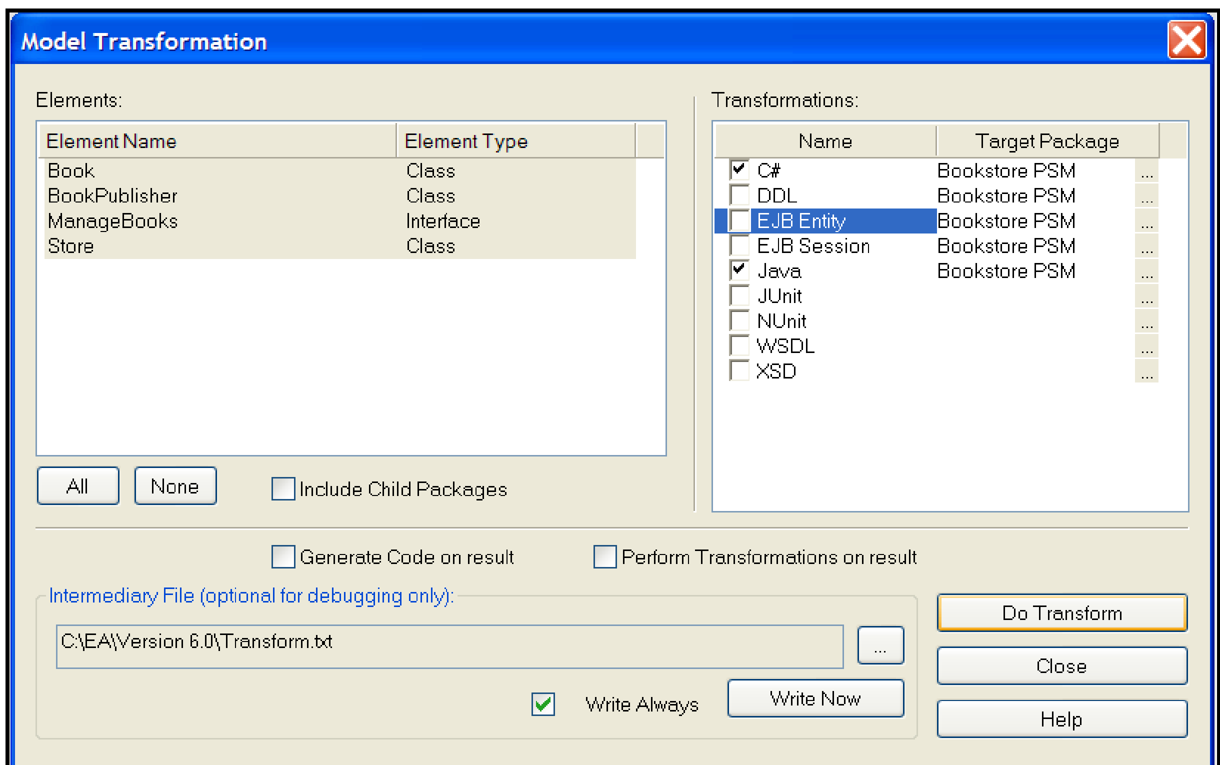


Figura 9: Etapa para transformação MDA da ferramenta

Fonte: Tutorial MDA in Pratice Enterprise Architect

A Figura 9 ilustra a etapa da transformação MDA na ferramenta, onde o usuário pode escolher qual a plataforma específica (C#, DDL, EJB Entity, EJB Session, Java) para qual o modelo será transformado.

Por exemplo, cria-se um diagrama UML, e nele especifica-se que o modelo não pertence a nenhuma linguagem. Então a partir dele pode-se gerar diagramas para diversas plataformas. Aspectos inerentes à plataforma são adicionados aos modelos e posteriormente seu código

fonte (AGUIAR, 2012) A figura a seguir mostra um exemplo de transformações que podem ser geradas pela ferramenta.

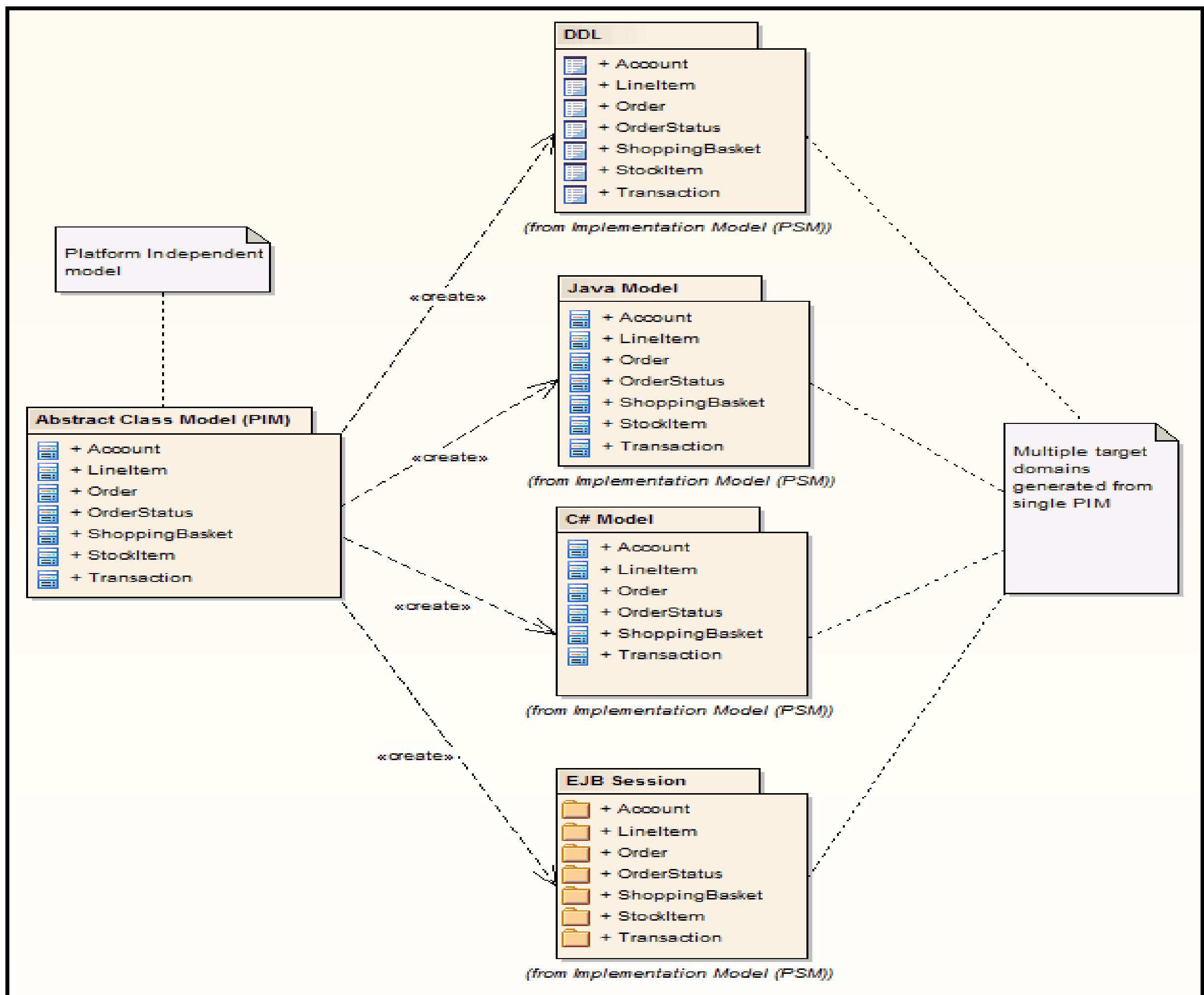


Figura 10: Transformações PSMs a partir do modelo PIM

Fonte: Tutorial MDA in Pratice Enterprise Architect

A Figura 10 ilustra as transformações de modelos específicos da plataforma (PSM) em diversas tecnologias a partir de um modelo independente da plataforma (PIM) criado. Após transformação do modelo PIM em modelo PSM é possível gerar código para uma plataforma especifica. A ferramenta dá a possibilidade de gerar código automaticamente a partir do modelo específico da plataforma. A geração do código é produzida selecionado o modelo PSM e acionando o comando *generate code* pertencente a ferramenta.

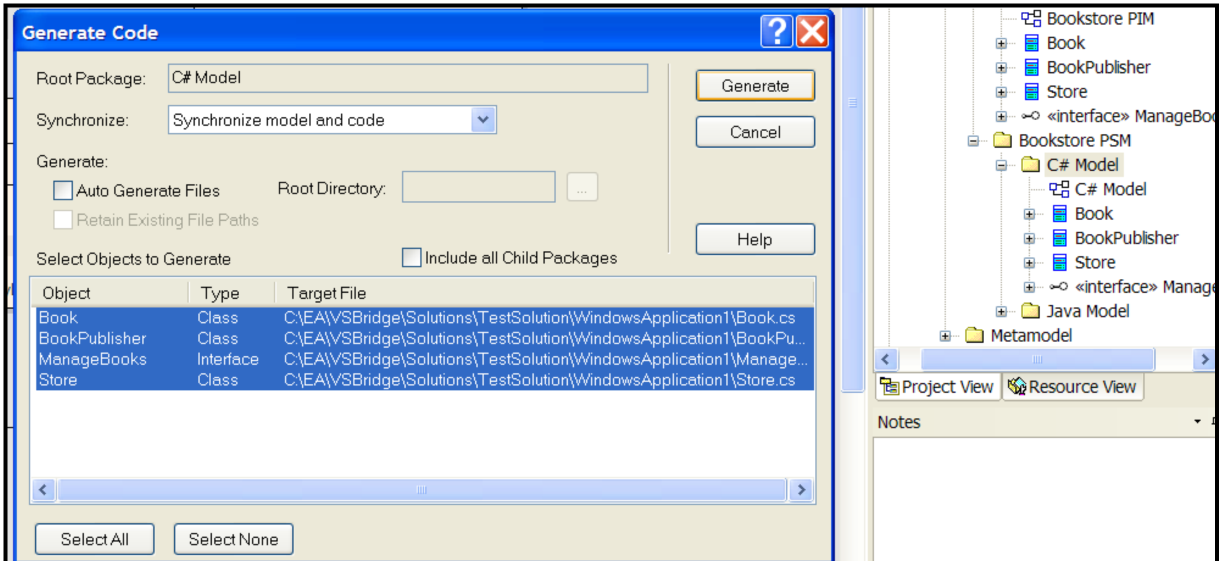


Figura 11: Etapa para geração de código utilizando EA

Fonte: Tutorial MDA in Pratice Enterprise Architect

A Figura 11 ilustra o processo de geração de código para uma plataforma *C#* a partir de um modelo PSM gerado anteriormente.

A seguir serão mostrados respectivamente um exemplo de modelo PSM (Figura 12) e o código gerado a partir desse modelo (Figura 13) usando a Enterprise Architect.

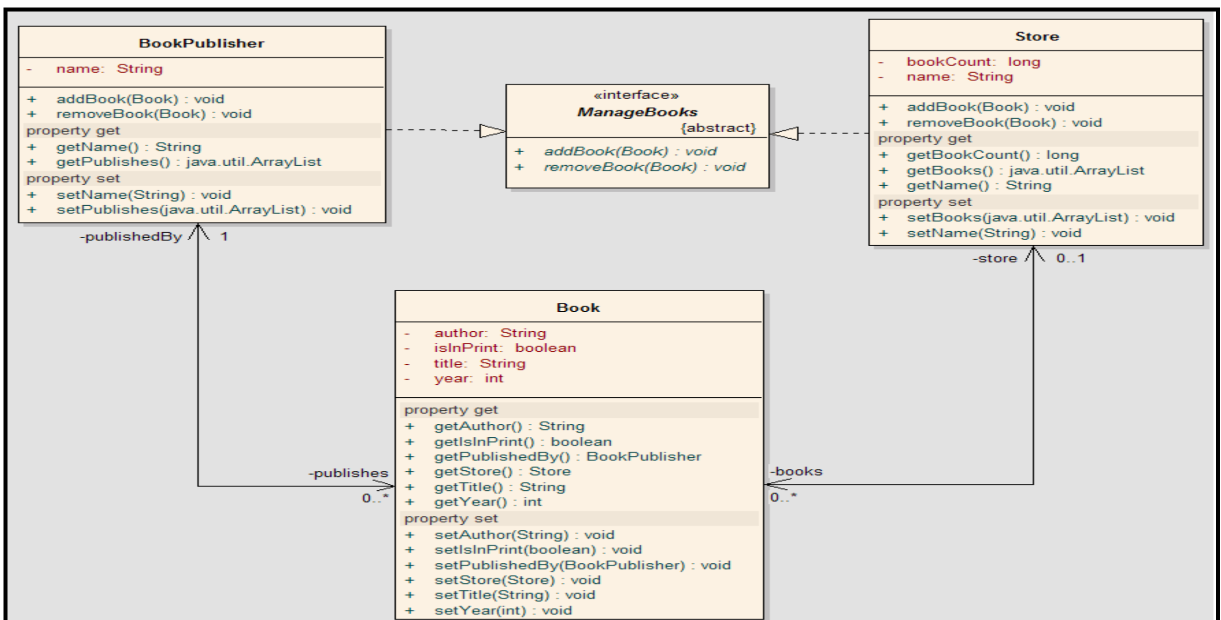


Figura 12: Modelo específico da plataforma *Java* gerado pela EA

Fonte: Tutorial MDA in Pratice Enterprise Architect

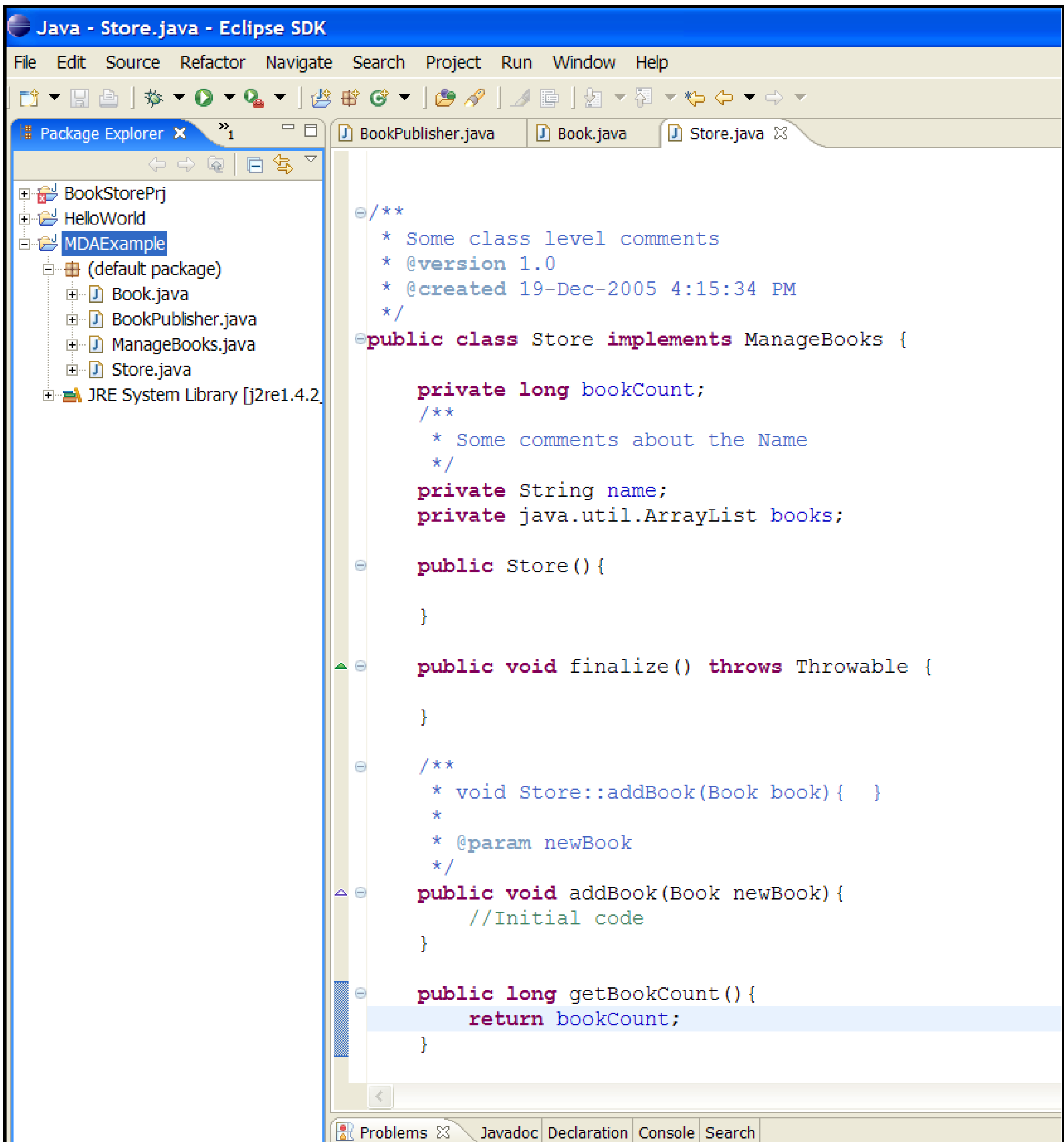


Figura 13: Código do PSM gerado pela EA

Fonte: Tutorial MDA in Pratice Enterprise Architect

A ferramenta permite que o usuário não só edite os modelos de transformação já embutidos na própria ferramenta, como também adicione novos templates de transformação para geração de novos PSMs. A edição e construção desses modelos são feitas utilizando linguagens de transformação de modelos (Figura 14). A seguir (Figura 14) é mostrado um exemplo do

template de transformação do tipo *Class* adotada pela Enteprise Architect. Observa-se que é possível escrever o comportamento da transformação através das linguagens definidas nos templates de transformação. A adição de um novo template de transformação para uma determinada plataforma é realizada através do comando *Add New Stereotyped Override* pertencente à ferramenta.

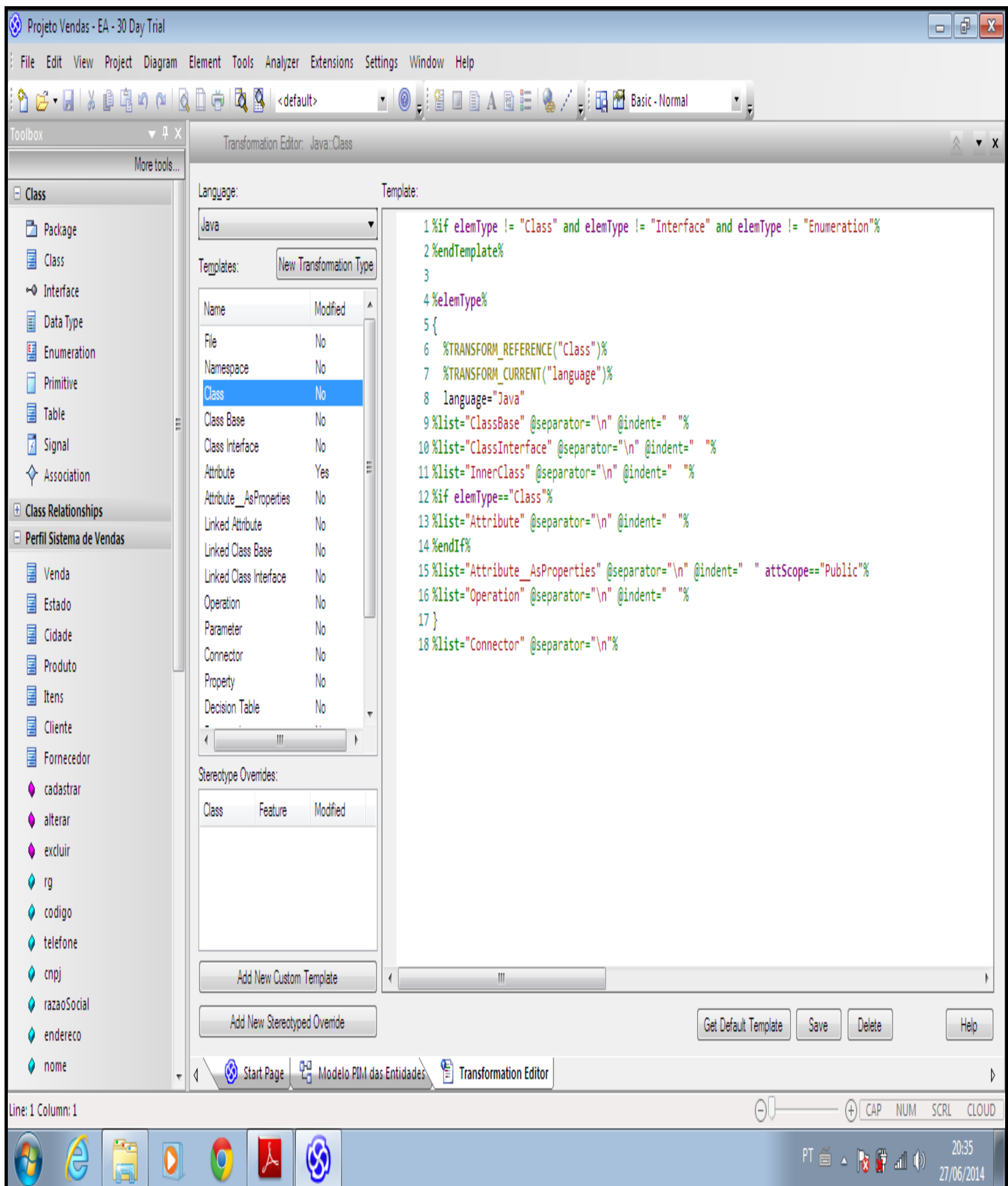


Figura 14: Template de Transformação Enterprise Architect

2.2 Métodos Ágeis

O surgimento dos chamados métodos ágeis se deu em resposta aos métodos existentes que não supriam as necessidades dos desenvolvedores de software (ALVAREZ, 2010). São diversos os métodos existentes, todos baseados em uma proposta de desenvolvimento iterativo, foco na comunicação interativa e na redução do esforço empregado, possuindo suas próprias peculiaridades.

O termo "Metodologias Ágeis" tornou-se popular em 2001 quando dezessete especialistas em processos de desenvolvimento de software representando os métodos Scrum [Schwaber e Beedle (2002)], Extreme Programming (XP) [Beck (1999)] e outros, estabeleceram princípios comuns compartilhados por todos esses métodos (SOARES, 2012). A importância dos métodos ágeis é reconhecida pela existência da comunidade *Aggile Alliance*, responsável pelo constante desenvolvimento de novos métodos e aperfeiçoamento dos existentes (ALVAREZ, 2010). A Aliança Ágil formulou um manifesto contendo um conjunto de princípios que definem critérios para os processos de desenvolvimento ágil de software chamado de Manifesto Ágil (2004, AMBLER apud AGUIAR, 2012).

Os conceitos chave do "Manifesto Ágil" são (SOARES, 2012):

- Indivíduos e interações ao invés de processos e ferramentas.
- Software executável ao invés de documentação.
- Colaboração do cliente ao invés de negociação de contratos.
- Respostas rápidas a mudanças ao invés de seguir planos.

O "Manifesto Ágil" não rejeita os processos e ferramentas, a documentação, a negociação de contratos ou o planejamento, mas simplesmente mostra que eles têm importância secundária quando comparado com os indivíduos e interações, com o software estar executável, com a colaboração do cliente e as respostas rápidas a mudanças e alterações (SOARES 2012).

Os membros da Aliança Ágil refinaram as filosofias presentes em seu manifesto em uma coleção de doze princípios para que o enfoque do desenvolvimento ágil fosse compreendido pelas pessoas (AGUIAR, 2012). São estes:

- A prioridade é satisfazer ao cliente através de entregas de software de valor contínuas e frequentes.

- Entregar software sem funcionamento com frequência de algumas semanas ou meses, sempre na menor escala de tempo.
- Ter o software funcionando é a melhor medida de progresso.
- Receber bem as mudanças de requisitos, mesmo em uma fase avançada, dando aos clientes vantagens competitivas.
- As equipes de negócio e de desenvolvimento devem trabalhar juntas diariamente durante todo o projeto.
- Manter uma equipe motivada fornecendo ambiente, apoio e confiança necessário para a realização do trabalho.
- A maneira mais eficiente da informação circular dentro da equipe é através de uma conversa face a face.
- As melhores arquiteturas, requisitos e projetos provêm de equipes organizadas.
- Atenção contínua a excelência técnica e um bom projeto aumentam a agilidade.
- Processos ágeis promovem o desenvolvimento sustentável. Todos envolvidos devem ser capazes de manter um ritmo de desenvolvimento constante.
- Simplicidade é essencial.
- Em intervalos regulares, a equipe deve refletir sobre como se tornarem mais eficazes e então se ajustar e adaptar seu comportamento.

A comunidade de desenvolvimento de software vem demonstrando grande interesse nos métodos ágeis (AGUIAR, 2012). Como muitas técnicas estabelecidas hoje em dia envolvendo métodos ágeis seguem os mesmos princípios, vários destes métodos apresentam práticas ágeis comuns (ALVAREZ, 2010). Para facilitar a escolha do método ágil, apresentarei o framework proposto por FAGUNDES (2005, *apud* AGUIAR, 2012, p.21).

2.3 Framework de práticas ágeis

O framework criado por FAGUNDES (2005, *apud* AGUIAR, 2012, P.22) apresenta as atividades dos métodos ágeis: Extreme Programming (XP), Scrum, Feature Driven Development (FDD), Adaptive Software Development (ASD) e Agile Modeling (AM) e pode ser utilizado durante o desenvolvimento de softwares, onde as atividades sugeridas por cada um dos métodos poderão ser selecionadas pela equipe de desenvolvimento de acordo com suas necessidades e experiências.

O autor que propôs o framework fez uma pesquisa sobre cada um dos métodos e analisou suas atividades, fazendo um estudo comparativo entre elas, podendo assim agrupar as atividades semelhantes, o que possibilitou a definição da estrutura do framework proposto em seu trabalho (AGUIAR, 2012).

Em um desenvolvimento de um projeto usando métodos ágeis, todos os membros envolvidos no projeto devem conhecer o método proposto para desenvolver o trabalho. Isto é de extrema importância para não comprometer o andamento do projeto. Uma das vantagens da utilização do framework proposto pelo autor é que, através dele, a equipe pode selecionar as atividades de alguns métodos para serem utilizadas em seu processo de software, sem a necessidade de se conhecer nenhum dos métodos específicos de maneira profunda (AGUIAR, 2012).

Portanto, o framework desenvolvido demonstra ser de fácil aplicação por equipes que possuam conhecimento sobre as práticas contidas nele, proporcionando aos interessados na utilização de um processo ágil uma boa oportunidade de fazê-lo de maneira simples, sem exigir grandes especializações (AGUIAR, 2012).

Para melhor entendimento do framework proposto, a seguir serão mostrados dados sobre como ele dispõe as informações sobre as atividades desenvolvidas durante o desenvolvimento de um sistema utilizando cada um dos métodos ágeis citados anteriormente.

2.3.1 Estrutura do Framework de Práticas Ágeis

Considerando que os métodos ágeis possuem algumas atividades semelhantes, o framework apresenta as atividades propostas pelos métodos agrupadas de acordo com possíveis semelhanças. As informações são apresentadas com a seguinte ordem: nome e descrição da atividade, quais as práticas e papéis incorporados à mesma e suas origens. Com essa estrutura o framework possibilita detectar quais atividades dentre as várias existentes nos métodos se assemelham para então escolher a que irá se adaptar melhor no desenvolvimento do sistema (Alvarez, 2010).

2.3.1.1 Atividade de Definição de Requisitos

User Stories – Para o levantamento das funcionalidades do sistema e prioridades é necessário que o cliente escreva as *user stories* (estórias dos usuários). Para cada funcionalidade uma

user story deverá ser escrita. A prática Cliente Presente é necessária para esta etapa, sendo os papéis envolvidos o Cliente e Equipe de Desenvolvimento. Origem da atividade: XP.

Lista de Requisitos – Para um melhor entendimento de todos os requisitos do sistema, um documento é formalizado contendo a descrição de cada requisito, separados por prioridade de desenvolvimento e apresentando informações como: descrição do requisito, tempo estimado de desenvolvimento e responsável pelo desenvolvimento. Origem da atividade: Scrum e FDD.

Organização da Documentação – Com os documentos sendo gerados, é necessário manter uma organização que pode ser realizada através da criação de outros documentos gerais: Documento de Requisitos, que denota tudo o que foi gerado durante a definição dos requisitos, Visão Geral Executiva apresentando uma estimativa de custos, benefícios, riscos e estimativas de pessoal e caso não seja gerado o documento Visão Geral Executiva, um documento chamado Visão Geral do Projeto é criado possuindo um resumo de informações como visão do sistema, mostrando principais contatos dos usuários, tecnologias e possíveis ferramentas usadas no desenvolvimento do sistema. Como não existe responsável por essa etapa ela poderá ser executada por qualquer integrante da equipe de desenvolvimento. Origem da atividade: AM.

Documentação Detalhada dos Requisitos – Quando a equipe de desenvolvimento não utiliza as *user stories* para documentação dos requisitos do sistema, outros meios podem ser utilizados como Casos de Uso e Diagramas de Classe da UML, ou qualquer outro meio utilizado para documentar requisitos. Papéis envolvidos: Equipe de Desenvolvimento, Cliente ou Especialista no Domínio. Origem da atividade: FDD.

2.3.1.2 Atividades de Atribuição de Requisitos às Iterações.

Planejamento das Iterações – Com os requisitos sendo definidos é necessário a distribuição dos mesmos às iterações seguindo as prioridades, dependências e riscos denotados (Considerando os requisitos já agrupados e priorizados). Uma reunião é sugerida antes do início de cada iteração, sendo assim a prática adotada nessa etapa é chamada Reuniões de Planejamento da Iteração. Outra prática que poderá ser adotada é o Cliente Presente, caso o cliente também participe do planejamento. Os papéis presentes na atividade são Equipe de Desenvolvimento e Cliente (caso participe do desenvolvimento como dito anteriormente). Origem da atividade: XP, Scrum, FDD, ASD.

Duração das Iterações – O tempo das iterações é um fator determinante para atingir um dos princípios da utilização dos métodos ágeis: "A prioridade é satisfazer ao cliente através de entregas de software de valor contínuas e freqüentes". Sendo assim, as iterações devem ter seu tempo reduzido e sempre ao ser finalizada, uma parte do sistema deve ser disponibilizada ao cliente. Os métodos ágeis propostos nesse framework utilizam um tempo de iteração que varia de 1 a 8 semanas, o que possibilita a equipe de desenvolvimento escolher a duração que melhor se adéqüe ao seu projeto. As práticas dessa etapa são Entregas Freqüentes e Contínuas e Prazos Pré-Fixados, os papéis envolvidos são Equipe de Desenvolvimento e Gerente de Projeto. Origem da atividade: XP, Scrum, FDD e ASD.

Distribuição dos Requisitos aos Responsáveis – Após o levantamento dos requisitos estes são distribuídos entre os responsáveis pelo desenvolvimento, tendo cada requisito ou conjunto de requisitos um programador responsável ou um grupo de programadores, com no máximo 6 integrantes e 1 responsável geral. A prática utilizada nesta etapa é a Propriedade Individual da Classe, e os papéis envolvidos são Programadores e Responsáveis pelas equipes de programadores. Origem da atividade: FDD.

Atualização da Documentação – Nesta etapa podem-se atualizar os diversos documentos gerados nas etapas anteriores como Definição dos Requisitos e Visão Geral do Sistema, caso haja necessidade de tais atualizações em campos como custos estimados e custos de pessoal. Origem da atividade: AM.

2.3.1.3 Atividades do Projeto de Arquitetura do Sistema

Projeto Geral do Sistema – Com os requisitos levantados até o momento, durante esta atividade é elaborado um projeto geral do sistema, pode ser realizado paralelamente com a atividade de Definição dos Requisitos e deve ser constantemente atualizado durante o processo de desenvolvimento. Para elaborar o projeto geral de sistema é indicado o uso de Diagramas de Classe e Diagramas de Sequência da UML. A prática sugerida nesta etapa é chamada Projeto Simples e propõe que um projeto simples além de facilitar sua compreensão facilita também sua manutenção. Os papéis relacionados a atividade são: equipe de desenvolvimento e gerente de projeto. Origem da atividade: XP, Scrum, FDD e AM.

Projeto Detalhado do Sistema – Considerando os conjuntos de requisitos que serão desenvolvidos em cada iteração, pode-se detalhar o projeto do sistema sempre que a equipe de

desenvolvimento achar necessário. Para isso, no início de cada iteração é proposto a realização dessa atividade. Novamente como na atividade descrita anteriormente, a prática Projeto Simples deve ser adotada. Tanto a equipe de desenvolvimento – que será responsável pela execução – quanto o gerente do projeto podem decidir se esta etapa será realizada ou não. Origem da atividade: FDD e AM.

Documentação Dos Projetos Gerados – Após as atividades tendo sido desenvolvidas é aconselhável a geração de um documento chamado Documentação do Sistema que irá agregar além da documentação gerada nas atividades anteriores, uma visão geral do projeto da arquitetura técnica e da arquitetura do negócio do sistema. Origem da atividade: AM.

2.3.1.4 Atividades de Desenvolvimento do Incremento do Sistema

Implementação dos Requisitos Durante Cada Iteração – Esta etapa se refere à geração de código de cada requisito que faz parte da iteração atual. A prática Padrões De Codificação, que denota padrões para a estrutura do código gerado é indicada para uma melhor organização, fazendo com que o código seja mais fácil para compreensão e manutenção, como também tenha um melhor controle de versões. Papéis presentes na atividade: Programadores. Origem da atividade: XP, Scrum, FDD e ASD.

Escrita dos Testes de Unidade e Aceitação – Para efeito de testes, a escrita de testes de unidade e aceitação podem ser realizadas previamente à atividade de implementação ou mesmo paralelamente, mas só podem ser executados após o código ter sido implementado. Testes de unidade escritos previamente fazem com que o programador revise o que realmente será codificado. Os testes de aceitação podem ser escritos pelos próprios clientes ou pelos programadores, porém sempre com a pergunta chave “O que precisaria ser conferido para se ter certeza de que um determinado requisito está OK?”. A prática Cliente Presente deve ser adotada sempre que o responsável por escrever os testes de aceitação for o cliente. Os papéis presentes na atividade são: Programadores, Clientes. Origem da atividade: XP.

Desenvolvimento Coletivo de Código – Com esta atividade os requisitos não possuem responsáveis diretamente selecionados para o desenvolvimento, toda a equipe se torna responsável e necessita saber o que será desenvolvido. A prática adotada por essa atividade é a Propriedade Coletiva, ou seja, todos são responsáveis por todo o sistema e qualquer um que

perceba uma oportunidade de agregar algo novo ao sistema deve fazê-lo. Papéis presentes na atividade: Programadores. Origem da atividade: XP

Desenvolvimento em Duplas – Ao contrário da atividade apresentada acima, o desenvolvimento em duplas sugere que a codificação seja feita por duplas de programadores, sendo um programador responsável por procurar a melhor maneira de implementar o método corrente, e o outro programador responsável por analisar se a estratégia adotada irá funcionar e se existe algum teste que ainda não foi utilizado e/ou se existe forma do código ser apresentado de forma mais simples. A prática denotada por esta atividade é chamada Programação em Pares. Considerando que é necessário o uso de pessoal a mais, o gerente de projeto deverá analisar se é viável a utilização desta atividade considerando maiores custos. Origem da atividade: XP

Realização de Refactoring – Quando necessário, a reconstrução do código é sugerida nesta etapa, fazendo com que o mesmo fique mais compreensível e reutilizável. Porém é necessário dar atenção ao tempo de desenvolvimento, analisando se o Refactoring é realmente necessário. Papéis envolvidos na atividade: Programadores. Prática adotada: Refactoring. Origem da atividade: XP.

Realização de Reuniões Diárias – Para manter todos os envolvidos no projeto informados sobre dificuldades e progresso do projeto, esta atividade propõe a realização de reuniões diárias com duração de 15 a 30 minutos e com os seguintes questionamentos em pauta: O que foi finalizado desde a última reunião? Quais as dificuldades encontradas durante o trabalho? Quais atividades pretendem-se realizar até a próxima reunião? Os participantes dessas reuniões são os Programadores e o Gerente de Projeto, e a prática adotada é chamada de Reuniões Diárias. Origem da atividade: XP, Scrum.

Desenvolvimento Simultâneo – Levando em conta o número de programadores disponíveis, o desenvolvimento dos conjuntos de requisitos que fazem parte de diferentes iterações pode ser feito simultaneamente. Os encarregados de tomar essa decisão são Gerente de Projeto em conjunto com a Equipe de Desenvolvimento. Origem da atividade: ASD.

Integração Paralela ao Desenvolvimento – O código gerado poderá ser integrado ao código já implementado em três ocasiões diferentes, após o desenvolvimento do incremento, após sua validação ou paralelamente à geração do código. Sendo a escolha a integração ocorrendo

paralelamente à geração de código, esta deve ocorrer diariamente. Prática proposta pela atividade: Integração Contínua. Papel envolvido: Programadores. Origem da atividade: XP.

Documentação do Desenvolvimento – Etapa em que são gerados documentos como Documentação de Operações, indicando dependência entre subsistemas, e Decisões de Projeto, contendo decisões críticas em relação ao projeto tomadas durante o desenvolvimento etc. Outro aspecto importante em qualquer sistema, a documentação do código deve ser levado em conta. Origem da atividade: AM.

2.3.1.5 Atividades de Validação do Incremento

Integração do Incremento Antes da Validação – Existindo uma dependência entre a atividade de validação e integração do incremento, a integração deve ser realizada antes da atividade de validação. Os envolvidos nessa etapa são os Programadores. Origem da atividade: XP.

Execução dos Testes de Unidade e Testes de Aceitação – Com a implementação do código chega o momento de realizar os testes de unidade e testes de aceitação. Caso eles não tenham sido escritos ainda, a atividade Escrita dos Testes de Unidade e Aceitação deve ser realizada para então dar procedimento a execução dos testes. Os programadores são encarregados dos testes de unidade, já em relação aos testes de aceitação, estes poderão ser executados pelos próprios clientes ou alguém diferente de programadores. Depois de realizados todos os testes e detectados os erros, novos testes deverão ser rodados para então confirmar que o incremento possui qualidade aceitável e atende as necessidades do cliente. As práticas propostas nessa atividade são: Testes e Cliente Presente, caso ele seja o responsável pela execução dos testes de aceitação. Os papéis envolvidos: Clientes, Testadores, Programadores. Origem da atividade: XP, FDD e ASD.

Inspeção do Código – Para detectar defeitos e confirmar que o código foi gerado com fácil entendimento, é feito uma inspeção de código pelos programadores, porém, é sugerido que cada programador inspecione o código de outro programador e nunca o seu próprio. A prática adotada nessa atividade é Inspeção tendo como participantes da atividade apenas os Programadores. Origem da atividade: FDD e ASD.

2.3.1.6 Atividade de Integração do Incremento

Integração do Incremento Resultante – Quando não tenha sido necessário integrar o incremento ou a atividade Integração Paralela ao Desenvolvimento não tenha sido adotada pela equipe de desenvolvimento para a realização das validações, a Integração do Incremento Resultante pode ser realizada no final de cada iteração e após todas as validações e inspeções. Envolvidos na atividade: Programadores. Origem da atividade: Scrum e FDD.

2.3.1.7 Atividades de Validação do Sistema

Reunião de Revisão da Iteração – Ao final de cada iteração é proposta a realização de uma reunião para que seja feita a avaliação da iteração pela equipe de desenvolvimento, com isso são identificados problemas e possíveis soluções para tais. Como primeiro passo, apenas equipe de desenvolvimento e gerentes de projeto participam dessas reuniões, posteriormente os clientes são envolvidos também para que seja apresentado o incremento e realizado testes de integração. A prática adotada por essa atividade é chamada Reunião de Revisão da Iteração. Origem da atividade: Scrum e FDD.

Colocar Sistema em Operação – Perto da finalização do desenvolvimento, esta atividade propõe que o sistema seja implantado dentro do ambiente em que será utilizado pelo cliente para então ocorrer a sua validação como um todo. Para isso, é aconselhável o acompanhamento da equipe de desenvolvimento junto ao cliente e que a ideia de que o sistema ainda em fase de teste seja entendido por todos, principalmente por parte do cliente, para que o mesmo não tome o sistema que está rodando em seu ambiente de uso como finalizado. Os papéis envolvidos nessa atividade: Clientes, Gerente de Projeto, Programadores. Origem da atividade: XP, Scrum

2.3.1.8 Atividades de Entrega Final

Entrega do Sistema ao Cliente – Ao não haver mais requisitos a serem desenvolvidos e com o cliente estando satisfeito com o sistema gerado, a atividade de Entrega do Sistema ao Cliente é executada. Durante essa atividade, uma reunião poderá ocorrer para oficializar a entrega final do sistema com todos os envolvidos no projeto. Envolvidos na atividade: todos no projeto. Origem da Atividade: XP, Scrum, FDD e ASD.

Geração de uma breve documentação – Quando durante o desenvolvimento a geração de uma documentação não tenha sido executada, essa atividade propõe que após a entrega do sistema ao cliente seja então gerada uma breve documentação detalhando as funcionalidades do sistema. Envolvidos na atividade: Programadores e Gerente de projeto. Origem da atividade: XP e Scrum.

Refinamento da Documentação Gerada – Etapa final com relação a geração de documentação do sistema, é recomendado a criação de documentos de Suporte que inclui um material de treinamento para a equipe de suporte e uma lista de contatos entre a equipe de manutenção. Outra documentação importante a ser criada é a documentação de Usuário, possuindo um manual de referência, guia de uso e guia de suporte. Os papéis envolvidos nessa atividade são os Programadores e Gerente de Projeto. Origem da atividade: AM.

Algumas atividades incluídas no framework são dependentes da prática de outras atividades, sendo essas possíveis de execução somente em conjunto com outras atividades ou após sua execução. Por outro lado, existem também atividades propostas que são realizadas somente se uma ou um conjunto de atividades não tenha sido executada, e por último existem as atividades opcionais que não possuem nenhum requisito em relação às outras atividades (ALVAREZ, 2010). A tabela das atividades excludentes, bem como outras práticas ágeis pertencentes a cada atividade do framework poderá ser consultada em (AGUIAR, 2012, p.22 à p.29).

2.4 Modelos Executáveis

Os modelos executáveis possuem todos os requisitos necessários para produzir a funcionalidade desejada de um único domínio. Estes modelos não são nem croquis, nem *blueprints* (projetos), mas modelos executáveis, como o nome próprio sugere. Eles permitem fornecer um sistema rodando com pequenos incrementos em comunicação direta com o cliente (Mellor, 2005).

Modelos executáveis atuam exatamente como o código, de certa forma, embora eles forneçam também a habilidade de interagir diretamente com o domínio do cliente, o que é algo que o código não faz direito. Eles não são exatamente o mesmo como o código porque precisam ser configurados juntos com outros modelos para produzir um sistema. Isso geralmente é feito por um compilador de modelos.

Assim como as linguagens de programação conferem independência de plataforma de hardware, os modelos executáveis conferem independência de plataforma de software, o que torna os modelos executáveis portáveis através de vários ambientes de desenvolvimento. Uma maneira de expressar modelos executáveis envolve o uso da UML executável (MELOR, SCOTT, UHL, WEISE, 2005) um perfil da UML que define uma semântica de execução para um subconjunto aerodinâmico e cuidadosamente selecionado da UML. O subconjunto é completo computacionalmente, portanto um modelo da UML executável pode ser diretamente executado.

Todos os diagramas (por exemplo, diagramas de classe, diagramas de máquinas de estados e especificações de procedimentos) são projeções de um modelo de semântica subjacente. Modelos UML que não suportam execução tais como diagramas de caso de uso podem ser usados livremente para auxiliar a construção dos modelos UML executáveis. A execução direta fornecida pela UML executável habilita um processo ágil, no qual um modelo pode ser construído e executado em parceria com o cliente.

2.5 MDA ÁGIL

A MDA ágil é baseada na notação de que o código e os modelos executáveis são os mesmos operacionalmente (MELLOR, SCOTT, UHL, WEISE 2005). Um modelo executável, por ser executável, pode ser construído, executado, testado e modificado em curtos ciclos incrementais e iterativos. Desse modo, os princípios da Aliança Ágil podem ser aplicados igualmente a modelos.

Muitos dos princípios da XP e da Aliança Ágil envolvem relacionamentos de processo e de cliente e seu gerenciamento, não o código. Como tal, os princípios de processos ágeis para a construção do código se aplicam tão bem à construção de modelos executáveis. Para esses princípios que especificam realmente o “o código” ou o “software”, um modelo executável, sob essa definição, é o código (MELLOR, SCOTT, UHL, WEISE 2005).

Para alcançar este estado de felicidade, os modelos devem ser completos o bastante para que possam ser executados de modo independente. Cada modelo em um determinado conjunto obedece necessariamente ao mesmo metamodelo, porque todos os modelos são iguais – não existem modelos de “análise” ou de “projeto”. Os modelos são interligados, em vez de transformados, e todos eles são então mapeados em um único modelo combinado que é

subsequentemente traduzido em código de acordo com uma única arquitetura do sistema (MELLOR, SCOTT, UHL, WEISE 2005).

Muitas das boas ideias da Aliança Ágil e da XP (tais como Desenvolvimento Sustentável, Estimativa para Melhorar e especialmente Cliente no Site) são igualmente positivas no contexto de modelos, e pode-se simplesmente substituir a palavra “código” por “modelo executável”. A MDA ágil utiliza modelos executáveis, não códigos, como artefatos principais. Modelos executáveis são mais abstratos do que o código, o que significa que destacam as comunicações com os clientes e melhoram a habilidade de interagir com o cliente (MELLOR, SCOTT, UHL, WEISE 2005).

2.6 TRABALHOS CORRELATOS

Com a expansão das técnicas MDD e MDA, e a vasta adoção dos métodos ágeis pela indústria de software (AGUIAR, 2012), naturalmente foram surgindo esforços para conciliar essas duas formas de produzir tecnologia.

Numa tentativa de encontrar trabalhos feitos nessa área, pesquisas foram realizadas em sites de busca como o Google, utilizando o Google acadêmico como base, e em bibliotecas de algumas universidades. Os trabalhos encontrados utilizam palavras chaves como MDD, métodos ágeis, engenharia de software, frameworks ágeis e MDA, o que facilitou bastante a busca. Outras pesquisas foram mais específicas, utilizando o nome da obra e links de artigos abordados nas referências dos outros trabalhos.

Foram encontrados artigos e monografias abordando o uso das técnicas MDA, MDD e métodos ágeis, bem como tentativas de conciliar o uso dessas técnicas. Alguns somente apresentavam uma abordagem teórica sobre o assunto.

A seguir será apresentado os trabalhos que se assemelham com o objetivo proposto por este que está sendo desenvolvido.

2.6.1 Agile Model-Driven Development in Practice

Trabalho realizado por ZHANG e PATEL no ano de 2010 teve o objetivo adicionar práticas de *Extreme Programming* (XP) ao MDD, além de mostrar como o MDD pode ser combinado com processo ágil chamado *System-Level Agile.*

“Particularmente, vamos mostrar como combinar um processo ágil chamado (SLAP) com um processo MDD para acelerar o desenvolvimento, melhorar a qualidade do produto e encurtar o tempo de entrega do software” (ZHANG, PATEL, 2010).

SLAP é um processo adotado pela Motorola baseado no método ágil Scrum e que inclui certas práticas de *Extreme Programming*. ZHANG E PATEL propõe uma relação entre as práticas ágeis do SLAP com processos MDD para o surgimento de um único processo chamado Agile MDD. Neste processo criam práticas como “modelagem paralela”, “modelagem iterativa e incremental” entre outras, a fim de miscigenar MDA com desenvolvimento ágil e verificar como seus conceitos podem ser utilizados juntos de maneira adequada (AGUIAR, 2012).

Os modelos utilizados pelos autores do trabalho são feitos em UML e transformados em código C++ por um compilador UML. O objetivo do trabalho de ZHANG e PATEL foi realizado com sucesso e foi aplicado em um sistema de telecomunicações em tempo real.

2.6.2 Um Relato de Experiência no Desenvolvimento Ágil de Sistemas com a MDA

Trabalho realizado por (BASSO, PILLA, 2010) descreve o uso de MDD com métodos ágeis em um ambiente de uma empresa privada utilizando uma ferramenta própria chamada Work Case ToolKit (WCT). O trabalho menciona como o processo de software e a ferramenta foram sofrendo modificações para adaptar-se ao contexto MDA (AGUIAR, 2012).

O relato dá grande visibilidade à ferramenta e destaca os perfis dos utilizadores e a importância deles dentro do processo de maneira geral. O trabalho relaciona diretamente a qualidade das definições de transformação com a qualidade do produto final e classifica as transformações de acordo com os modelos produzidos (AGUIAR, 2012). No final, os resultados da trabalho são apresentados onde afirmam que ferramentas MDA podem ser eficazes também no contexto dos processos ágeis (BASSO, PILLA, 2010), porém gasta-se um tempo no processo de aprendizagem da ferramenta.

2.6.3 Uso de MDA em um Framework para seleção de Práticas Ágeis

Trabalho desenvolvido por (AGUIAR, 2012) teve como objetivo analisar o uso de técnicas MDA em processos ágeis, definidos a partir do framework de práticas ágeis.

Para realização do seu trabalho, o autor utiliza frameworks proposto por (FAGUNDES, 2005) como Extreme Programming (XP) e Scrum, e ferramentas MDA como Enterprise Architect, Moderne, MagicDraw e AlphaSimple.

AGUIAR explica que a utilização desses frameworks se deu pelo fato de que sua utilização não precisa de um grande embasamento teórico na área. Ele realiza combinações entre os modelos gerados pelas ferramentas MDA e os processos ágeis gerados pelo framework de práticas ágeis a partir de aplicações feitas por ele.

"A questão fundamental para conciliar o Framework de Práticas Ágeis apresentado no capítulo dois com MDA é a adequação da ferramenta utilizada aos processos produzidos por ele e aos conceitos cruciais envolvendo MDA e métodos ágeis. Só assim MDA poderá ser utilizada com o Framework de Práticas Ágeis com total sucesso" (AGUIAR, 2012).

O autor termina seu trabalho chegando a conclusão de que a utilização de uma única ferramenta que contemple MDA em sua plenitude é inviável e que para que haja sucesso na integração entre frameworks de práticas ágeis e processos MDA seria necessário a análise e utilização de mais ferramentas, afim de encontrar uma combinação ideal entre funcionalidades e custo, que permita a exploração de desenvolvimento dirigido a modelos de forma adequada.

3 EXEMPLO DE USO

Este capítulo apresenta o exemplo de uso realizado para mostrar a integração entre MDA e Métodos Ágeis. O exemplo foi desenvolvido na ferramenta Enterprise Architect e consiste na construção de um sistema de vendas e impressoras.

A questão fundamental para conciliar um Framework de Práticas Ágeis a Arquitetura Dirigida à Modelos é adequar funções que a MDA oferece aos princípios estipulados pelo MDA Ágil.

Utiliza-se MDA Ágil para ganhar retorno direto de clientes sobre o produto em desenvolvimento. Construir casos de teste, escrever modelos executáveis, e compilar os modelos usando um compilador de modelos, fornecendo fragmentos do sistema de uma forma iterativa e incremental. De uma maneira análoga aos métodos ágeis, é o mesmo que dizer que deseja construir funcionalidades, escrever código, compilar o código usando um compilador de linguagens, de modo incremental, onde será substituída uma linguagem (código), por uma outra em um nível mais alto de abstração: um modelo executável e traduzido.

As seções a seguir detalham respectivamente: o cenário escolhido para o exemplo de uso; os casos de uso da aplicação; o desenvolvimento da primeira iteração da aplicação; o desenvolvimento da segunda iteração e o desenvolvimento da terceira iteração do sistema.

3.1 Cenário escolhido para exemplo de uso

A aplicação desenvolvida para mostrar o uso de MDA Ágil simula um sistema de vendas de impressoras, onde o foco principal é a parte do cadastro da venda. O escopo do sistema foi elaborado da seguinte maneira:

O cliente solicita um produto. O vendedor verifica se o cliente já é cadastrado no sistema. Caso não esteja cadastrado, cadastra-o com código, nome, RG e telefone. O vendedor pode verificar o estado e a cidade que esse cliente se localiza. Para tanto, o sistema deverá armazenar o código, nome e sigla do estado, bem como o código e o nome da cidade cujo cliente estará associado. O sistema também deverá armazenar os fornecedores e produtos ligados a estes fornecedores. Cada fornecedor possui um código, razão social, CNPJ endereço e telefone. Os produtos são cadastrados com a marca da impressora, modelo, preço de compra, preço de venda e quantidade total que será armazenada em estoque.

O sistema deverá permitir que o vendedor visualize a cidade de um determinado fornecedor. A venda será realizada pelo vendedor e o sistema deverá armazenar o nome do cliente, qual produto foi vendido, a quantidade que o cliente solicitou, a data da compra e o valor total da compra. A cada venda realizada a quantidade total do produto em estoque é atualizada de forma automática.

3.2 Casos de Uso da Aplicação

Como já foi dito anteriormente, modelos UML que não suportam execução tais como diagramas de caso de uso podem ser usados livremente para auxiliar a construção de modelos UML executáveis. Fazendo uma analogia com os métodos ágeis, os casos de uso servem para ajudar a descrever as funcionalidades que o sistema irá apresentar. Utilizando a metodologia ágil *Scrum* como exemplo, os casos de uso auxiliaria a construção do *Product Backlog* do projeto.

Com relação ao Framework de Práticas Ágeis proposto por Fagundes, a construção do modelo de casos de uso ajudaria em duas atividades pertencentes a este framework, a atividade de definição dos requisitos (que propõe os procedimentos de levantamento de requisitos, modelagem geral e documentação inicial) e a atividade do desenvolvimento do incremento do sistema (usada com a própria construção do casos de uso).

A Figura 15 descreve o modelo de Caso de Uso gerado como uma documentação pela ferramenta Enterprise Architect utilizado para desenvolver a aplicação:

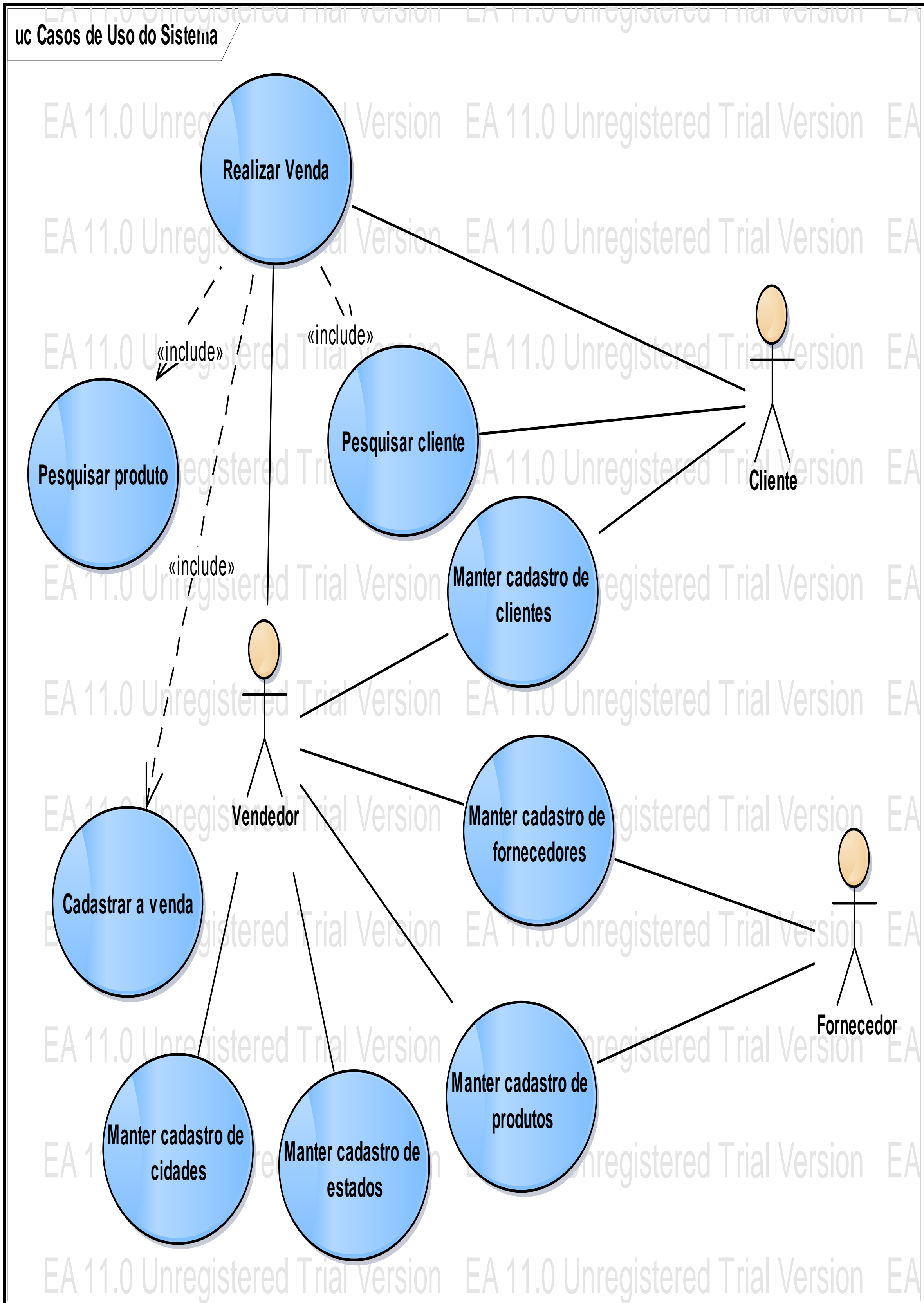


Figura 15: Casos de Uso do Sistema

Três atores interagem com o sistema, o vendedor, o cliente e o fornecedor. O vendedor é responsável por manter os dados dos clientes, fornecedores e seus produtos no sistema, realizando o processo de validação dos mesmos. Além disso, o vendedor é responsável por vender o produto ao cliente.

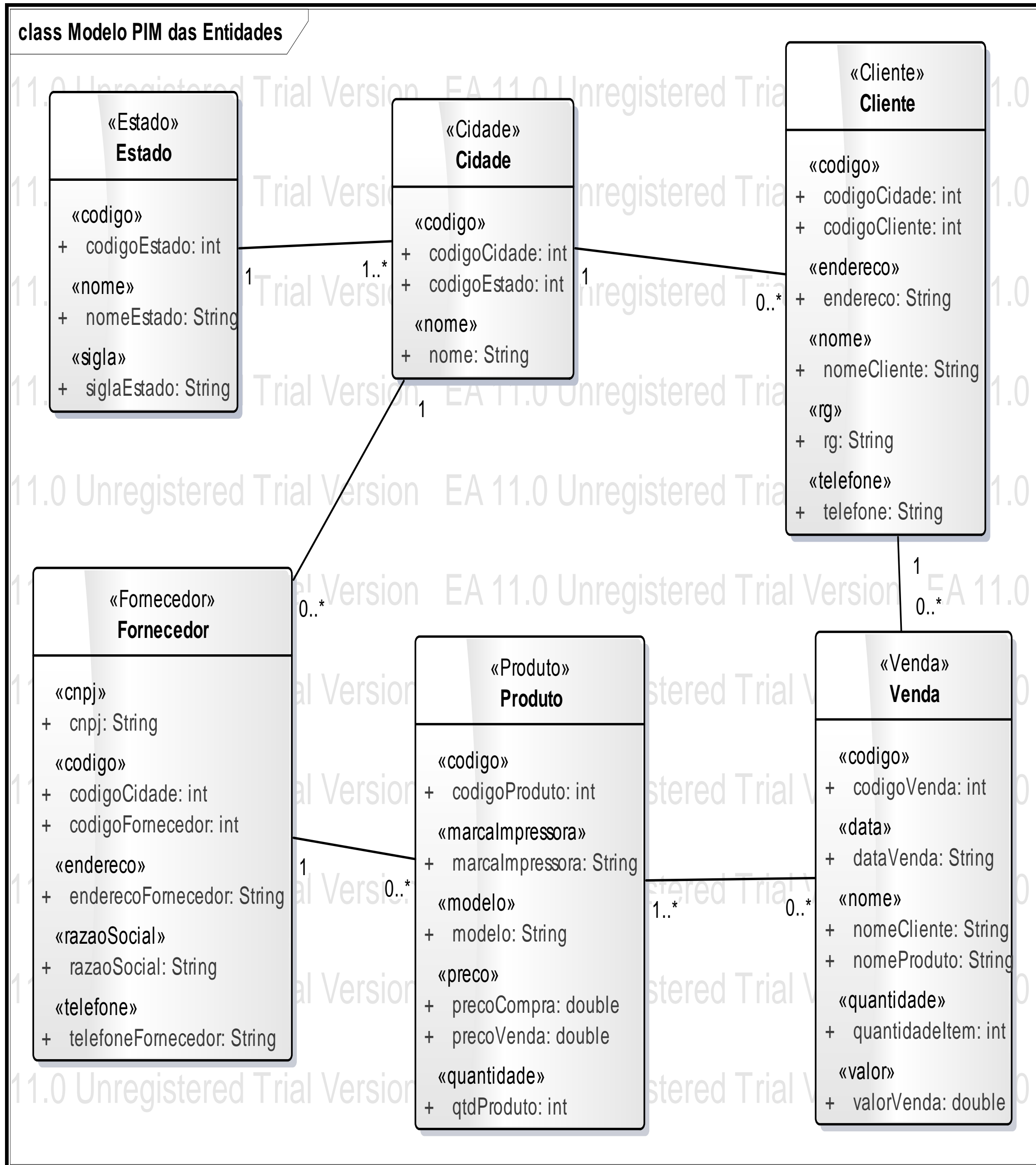
3.3 Primeira Iteração

O objetivo dessa primeira iteração é identificar quais serão as entidades que irão compor o sistema.

Para que o projeto se enquadre nos princípios da abordagem MDA Ágil é preciso construir modelos UML que sejam executáveis. Como foi dito anteriormente, MDA Ágil baseia-se na notação de que o código e os modelos executáveis são os mesmos operacionalmente. Por isso um modelo executável pode ser construído, executado, testado e modificado em curtos ciclos incrementais e iterativos, uma das características dos princípios da Aliança Ágil.

Durante esta etapa do projeto foi utilizada os processos que compõe uma estrutura MDA, com a criação de modelos independentes da plataforma (modelo PIM do sistema), bem como a realização das transformações desses modelos para uma plataforma de arquitetura específica do projeto (modelo PSM). A possibilidade das transformações entre os modelos fornecida de forma automática pela ferramenta Enterprise Architect mantém a construção do meu projeto num modo iterativo e incremental, onde pode-se selecionar e editar os modelos que serão transformados.

Foi utilizado como base para construção do modelo PIM o diagrama de classes, que segundo Mellor (2005) é um tipo de modelo usado na construção de modelos executáveis. De forma similar ao modelo MVC, foi construído um primeiro modelo PIM com as classes que irão compor o pacote modelo da aplicação (no caso do meu projeto dei o nome de pacote entidade) e que será transformado em um modelo PSM específico da plataforma Java. As figuras 16 e 17 mostram respectivamente os modelos PIM e PSM dessa primeira etapa construídos para desenvolver a aplicação. Na Figura 16 é apresentado o diagrama de classes desenvolvido no PIM. Observe que 6 classes foram definidas até este momento, com o nome da classe, seus atributos e as associações entre as classes.



EA 11.0 Unregistered Trial Version

Figura 16: Modelo PIM Primeira Iteração

A Figura 17 mostra o modelo PSM gerado pela ferramenta a partir do PIM construído. Para adequar a plataforma java, a Enterprise Architect cria *getters* e *setters* para as propriedades das classes envolvidas no PIM. Apenas os atributos criados a partir das relações entre as classes têm esses métodos gerados.

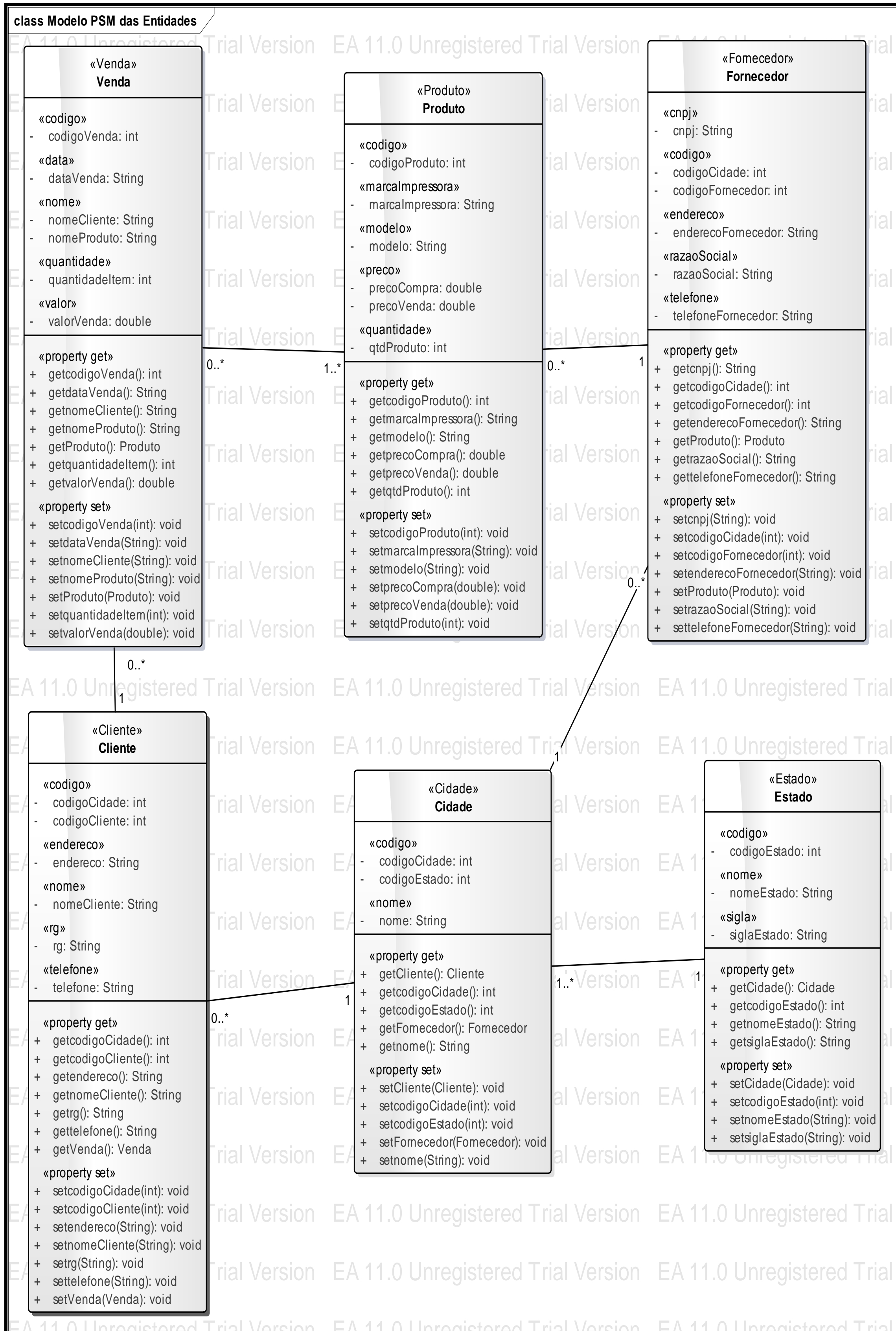


Figura 17: Modelo PSM Primeira Iteração

Após criar os modelos e realizar as transformações necessárias, pode-se gerar o primeiro incremento do sistema, ou seja, geração do código, de forma automatizada, a partir do modelo PSM. Essa iteração se assemelha com a prática ágil *Projeto da Iteração* incluído na Atividade *Desenvolvimento do Incremento do Sistema* pertencente ao Framework de Práticas Ágeis citado por (AGUIAR, 2012, p.24). A Figura 18 mostra um exemplo do código do primeiro incremento gerado automaticamente pela ferramenta a partir do modelo PSM. Para a classe estado foi gerado o código com os atributos usando os tipos de dados do Java e a propriedade *getter* e *setter* de cada atributo.

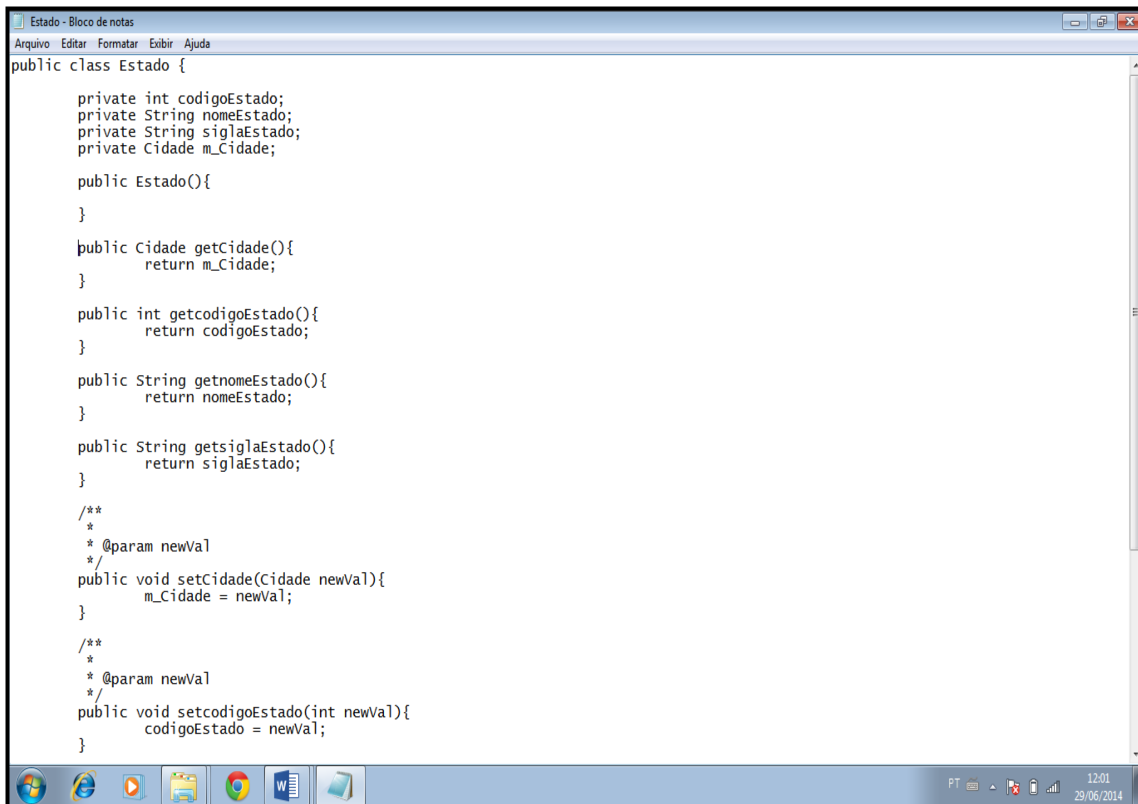


Figura 18: Código gerado pela Enterprise Architect

A seguir é mostrado uma tabela contendo um resumo dos processos e atividades envolvidas nessa primeira iteração para desenvolvimento da aplicação usando MDD Ágil. Por exemplo, a prática ágil Projeto da Iteração foi realizada na abordagem MDD na visão PIM. Desta forma, considerando a abordagem MDD Ágil, foi construído o primeiro modelo executável (o diagrama de classes apresentado na figura 16)

PRÁTICAS ÁGEIS	MDD	MDD ÁGIL
Projeto da Iteração	Construção do modelo PIM	Construção do primeiro modelo executável
Primeira Iteração	Transformação do modelo em PSM para plataforma JAVA	Transformação do modelo executável (geração do PSM)
Incremento	Geração do código a partir do PSM	Execução do modelo para gerar código
Inspeção do código	Verificação do código gerado pelo modelo	

Tabela 1: Primeira iteração da aplicação usando MDD ÁGIL

3.4 Segunda Iteração

O objetivo dessa iteração é construir o modelo que irá compor a base de dados do sistema. As práticas para o desenvolvimento desta etapa são semelhantes a etapa anterior. Entretanto, a uma ressalva que precisa ser mencionada – a prática de reuso dos modelos. A ferramenta Enterprise dá a possibilidade de você reusar um modelo PIM já construído anteriormente em outro pacote do projeto para ser transformado em um novo PSM numa plataforma diferente.

O modelo reutilizado sofrerá modificações até que seu PSM esteja pronto para atender os requisitos propostos da aplicação. Esse processo caracteriza-se num processo iterativo e incremental, concordante com os preceitos dos métodos ágeis, onde são proporcionados pelas transformações entre os modelos (processos MDA) até que o código alvo esteja correto. O modelo PSM terá uma nova plataforma alvo que irá servir de auxílio para a construção da base de dados da aplicação. A Figura 19 mostra o modelo PIM que será usado na construção da base de dados do sistema, construído a partir do modelo PIM proposto da primeira iteração:

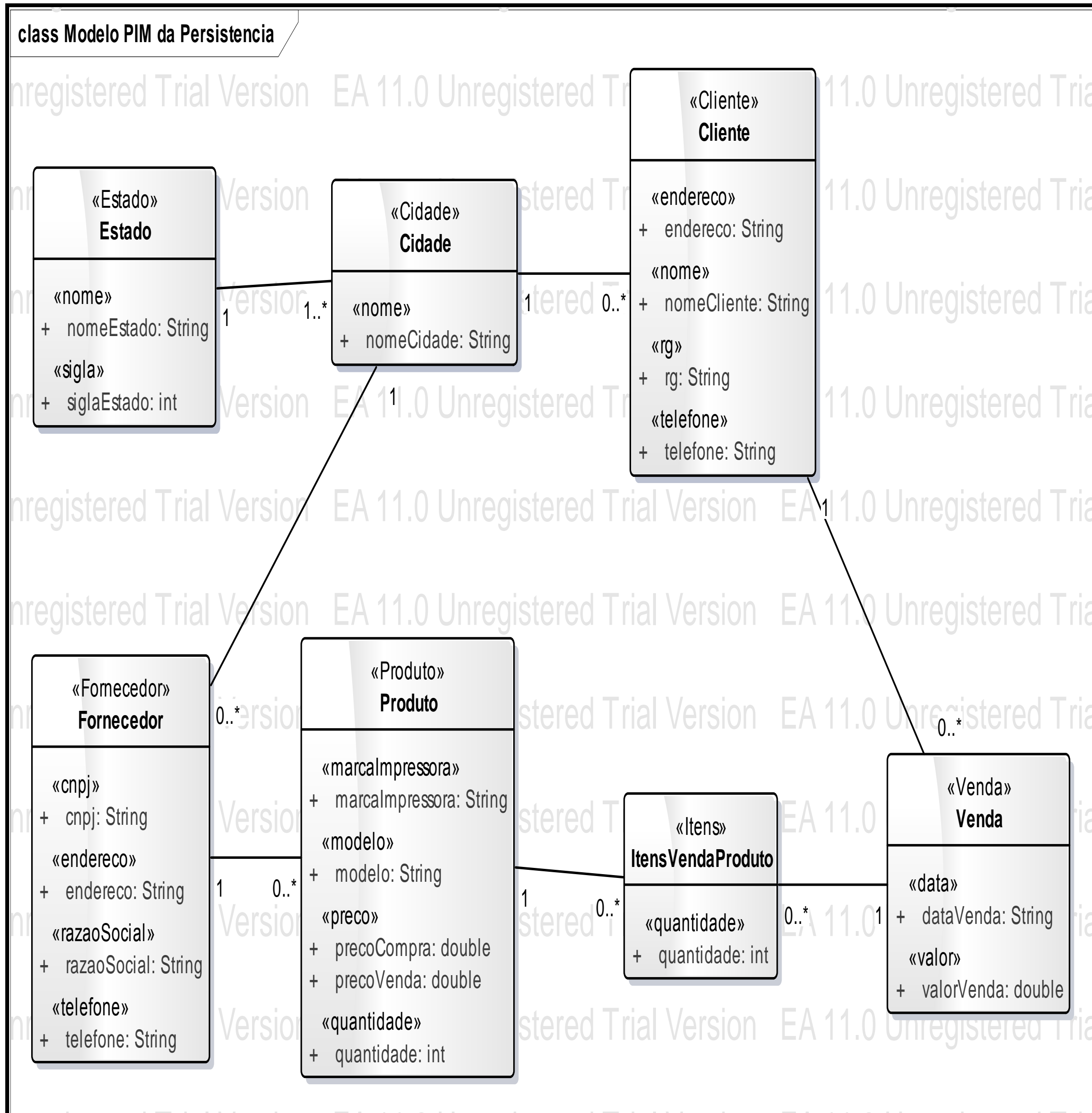


Figura 19: Modelo PIM Segunda Iteração

Em seguida uma nova transformação foi realizada de maneira automática na ferramenta a partir do modelo PIM proposto na segunda iteração colocando a *Data Definition Language* (DDL) como plataforma alvo. A base de dados configurada para essa plataforma será o MySQL. A vantagem de usar MDA para criação do modelo que possibilitará a criação da base de dados fica evidente com as transformações dos modelo PIM em PSMs. A Figura 20 mostra o PSM gerado a partir do modelo PIM da segunda iteração:

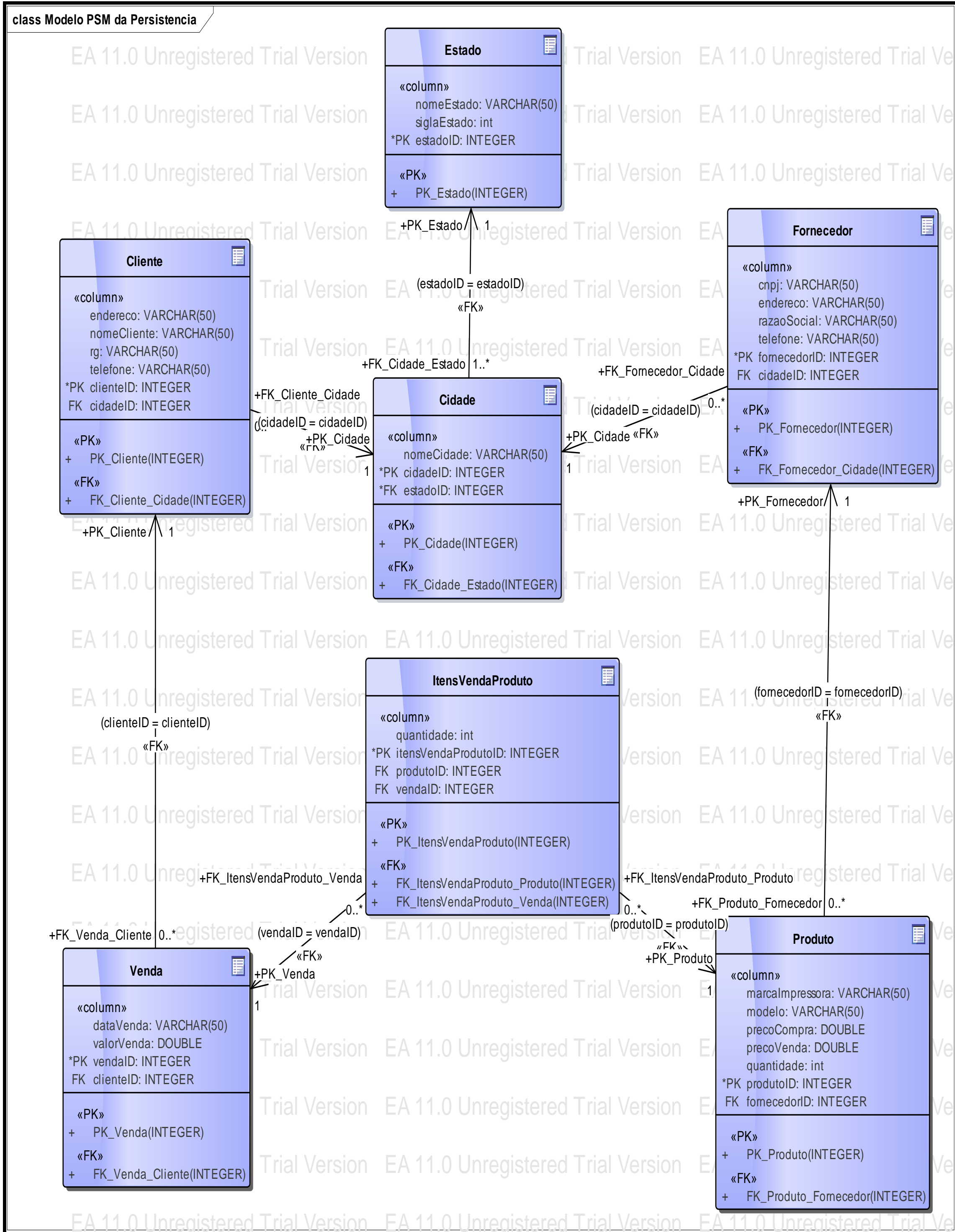


Figura 20: Modelo PSM Segunda Iteração

Os templates de transformação embutidos na ferramenta Enterprise e que são responsáveis por gerar PSM para essa plataforma criam de forma automática relações e características de

um modelo ER. Após a transformação, um incremento é gerado e executado para criação da base de dados da aplicação (Figura 21).

```
USE sistema
;

SET FOREIGN_KEY_CHECKS=0;


DROP TABLE IF EXISTS Cidade CASCADE
;
DROP TABLE IF EXISTS Cliente CASCADE
;
DROP TABLE IF EXISTS Estado CASCADE
;
DROP TABLE IF EXISTS Fornecedor CASCADE
;
DROP TABLE IF EXISTS ItensVendaProduto CASCADE
;
DROP TABLE IF EXISTS Produto CASCADE
;
DROP TABLE IF EXISTS Venda CASCADE
;

CREATE TABLE Cidade
(
	nomeCidade VARCHAR(50),
	cidadeID INTEGER NOT NULL AUTO_INCREMENT,
	estadoID INTEGER NOT NULL,
	PRIMARY KEY (cidadeID),
	KEY (estadoID)

)
;


CREATE TABLE Cliente
(
	endereco VARCHAR(50),
	nomeCliente VARCHAR(50),
	rg VARCHAR(50),
	telefone VARCHAR(50),
	clienteID INTEGER NOT NULL AUTO_INCREMENT,
	cidadeID INTEGER,
```

Figura 21: Código da Base de Dados Usada na Aplicação

A seguir é mostrado uma tabela contendo um resumo dos processos e atividades envolvidas nessa segunda iteração para desenvolvimento da aplicação usando MDD Ágil:

PRÁTICAS ÁGEIS	MDD	MDD ÁGIL
Projeto da Iteração	Construção do modelo PIM para compor a base de dados	Construção do segundo modelo executável
Segunda Iteração	Transformação do modelo em PSM para plataforma DDL	Transformação do modelo executável
Incremento	Geração do código a partir do PSM	Execução do modelo para gerar código
Inspeção do código	Verificação do código gerado pelo modelo	Criação do banco a partir do código gerado pelo modelo

Tabela 2: Segunda iteração da aplicação usando MDD ÁGIL

3.5 Terceira Iteração

Por fim chega à última iteração com a criação do modelo que vai atender os requisitos de maior complexidade da aplicação. O modelo criado será independente da plataforma e servirá de base para criação dos métodos CRUD do sistema.

Um diagrama de estados, por ser um modelo executável, será interligado com a classe principal da aplicação. O diagrama irá demonstrar o funcionamento da parte da venda para o cliente. A construção de modelos interligados ajudam não só na execução do sistema como traz um melhor entendimento do domínio por parte do cliente, um dos preceitos da metodologia ágil. A Figura 22 demonstra a modelo de estados da venda:

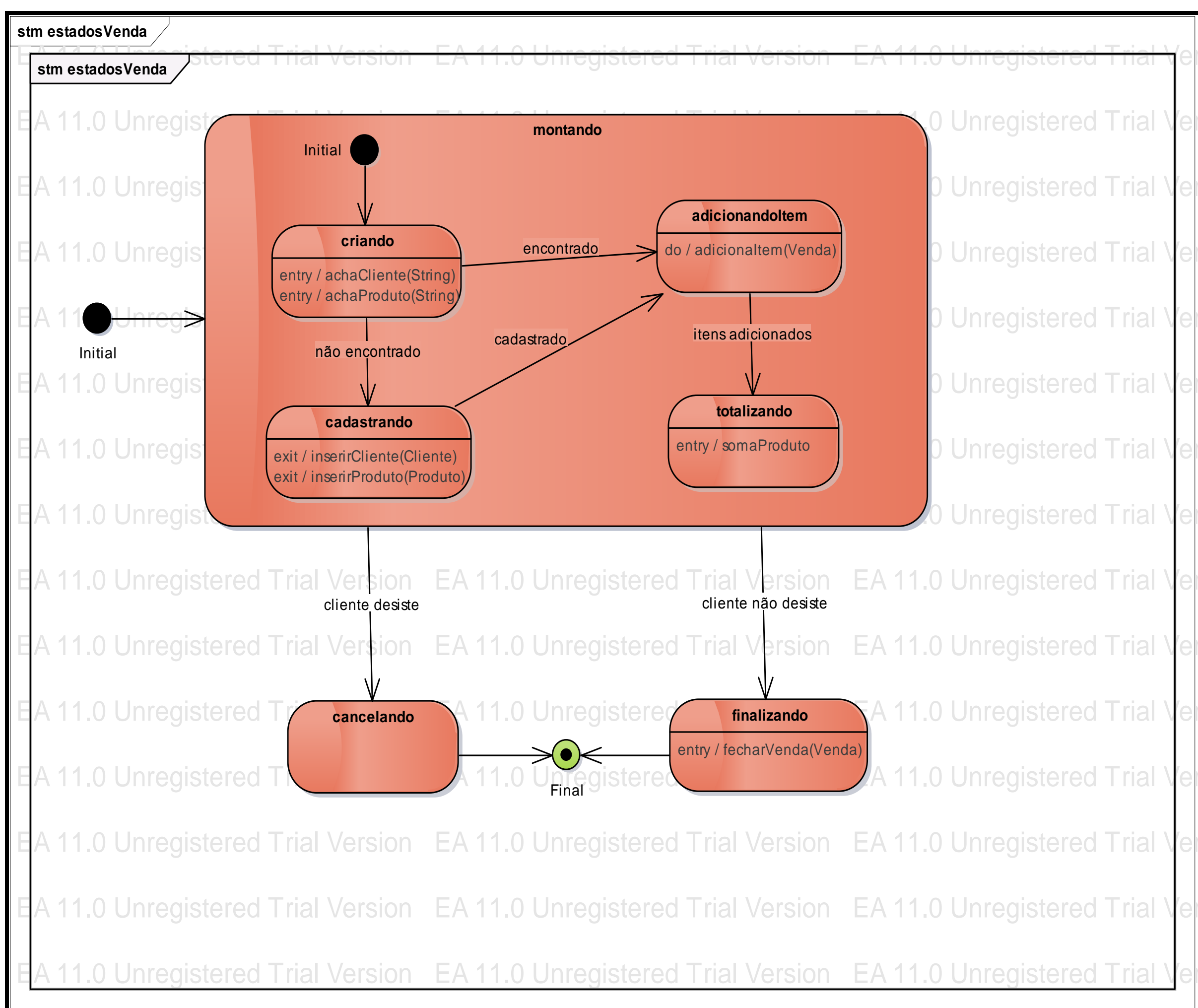


Figura 22: Diagrama de Estados da Venda

O modelo representa os estados possíveis da venda. Cada estado possui atividades inerentes a ele. Essas atividades são executadas à medida que o sistema vai transitando entre os estados. A ferramenta possui um simulador que permite a execução do modelo, característica de um

MDA ágil, onde a construção do sistema é formado por modelos executáveis. Para tanto, basta que o diagrama de estados esteja interligado a uma classe do sistema. A figura 23 mostra o modelo PIM com as funcionalidades das classes que atenderão os requisitos definidos no modelo casos de uso do projeto:

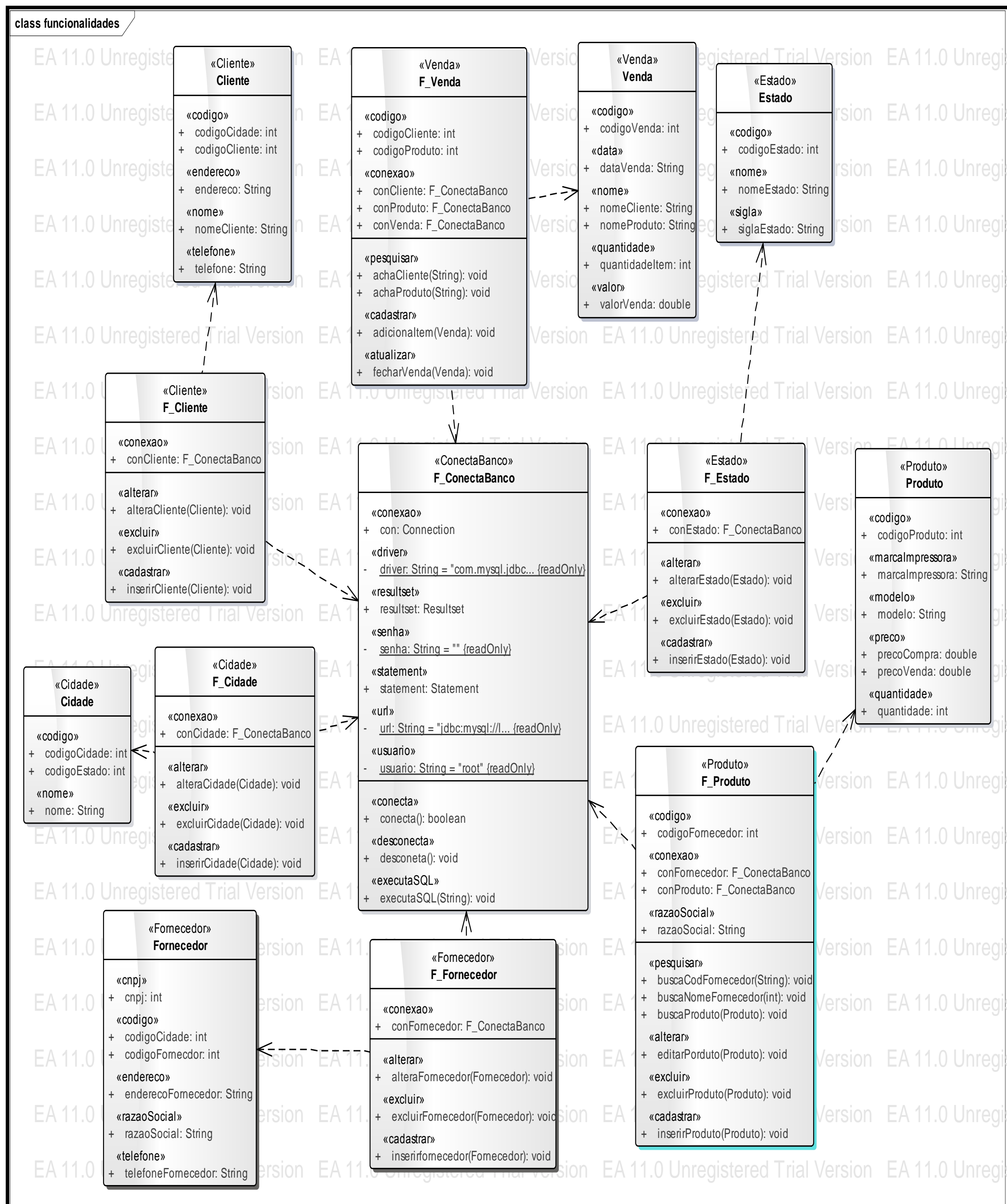
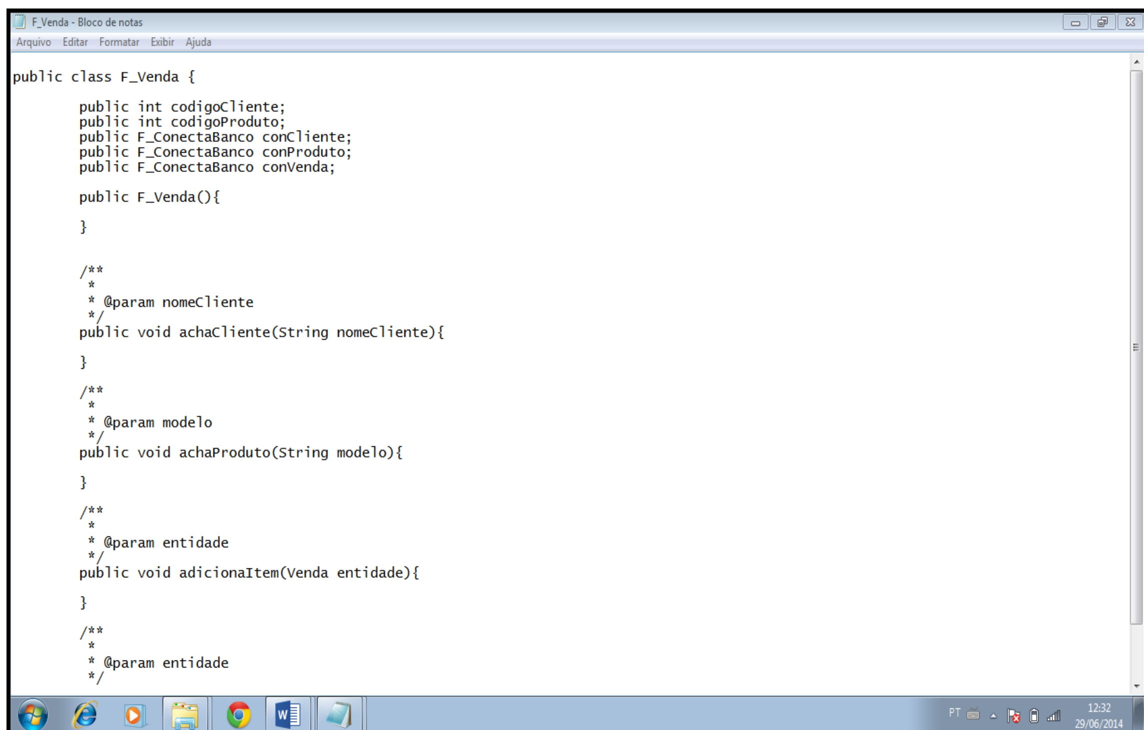


Figura 23: Modelo PIM Terceira Iteração

O diagrama demonstra a relação de dependência entre as classes do modelo PIM da primeira iteração com o modelo PIM da etapa final, bem como as funcionalidades que cada classe vai

possuir. O estereótipo venda ligado a classe *F_Venda* indica a relação de dependência que essa classe tem com a classe *Venda* construído na primeira iteração. O modelo de estados da venda vai ser interligado com a classe que contém as funcionalidades responsáveis por cada atividade pertencente ao estado do diagrama. No caso do exemplo, a classe *F_Venda*. A ferramenta permite que o modelo de estados seja adicionado a classe para viabilizar a comunicação entre a classe e o objeto da classe. A ligação entre os modelos representa uma característica da UML executável, princípio condizente com o MDA Ágil.

Após ser feito todo o processo de construção e ligação dos diagramas, o incremento é gerado (Figura 24).



```
public class F_Venda {

	public int codigoCliente;
	public int codigoProduto;
	public F_ConectaBanco conCliente;
	public F_ConectaBanco conProduto;
	public F_ConectaBanco conVenda;

	public F_Venda(){

	}


	/**
	 *
	 * @param nomeCliente
	 */
	public void achaCliente(String nomeCliente){

	}

	/**
	 *
	 * @param modelo
	 */
	public void achaProduto(String modelo){

	}

	/**
	 *
	 * @param entidade
	 */
	public void adicionaItem(Venda entidade){

	}

	/**
	 *
	 * @param entidade
	 */
```

Figura 24: Código da Classe *F_Venda* gerado pela EA

A ferramenta possibilita o usuário gerar o código do diagrama de estados sem sobrescrever o código da classe. Além disso, é possível simular a execução do diagrama usando o simulador pertencente a ela. A Figura 25 mostra um exemplo do código do diagrama de estados gerado a partir da classe *F_Venda*.

StateMachineCode - Bloco de notas

Arquivo Editar Formatar Exibir Ajuda

```
	}

	/**
	 *
	 * @param entidade
	 */
	public void fecharVenda(Venda entidade){

	}
	/* Begin - EA generated code for StateMachine */

	private enum StateType
	{
		estadosVenda_Final,
		estadosVenda_finalizando,
		estadosVenda_cancelando,
		estadosVenda_montando,
		ST_NOSTATE
	}
	private enum TransitionType
	{
		TT_NOTRANSITION
	}
	private enum CommandType
	{
		Do,
		Entry,
		Exit
	}
	private StateType currState;
	private StateType nextState;
	private TransitionType currTransition;
	private boolean transcend;
	private boolean bStayInCurrentState = false;
	private StateType estadosVenda_history;
	private void estadosVenda_finalizando(CommandType command)
	{
		switch(command)
		{
			case Do:
			{
				if(!bStayInCurrentState)
```

PT 12:44 29/06/2014

Figura 25: Código do Diagrama de Estados Gerado pela EA

O código do diagrama de estados é gerado junto ao código da classe que ele está ligado sem sobrescrevê-lo. O código é composto por uma enumeração do tipo de estados que pertence ao modelo (*StateType*) e do tipo de comando que vai ser executado *(Do, Entry e Exit*) durante a passagem desses estados. A transição entre os estados é realizada segundo uma condição chamada *Guard Condition* que a depender do seu resultado, ou seja, se a condição de guarda for verdadeira ou falsa, executa uma funcionalidade que seja inerente ao estado. O comando *Entry* executa a funcionalidade ao entrar no estado, o *Exit* executa a funcionalidade ao sair do

estado e o comando *Do* continua executando a funcionalidade até que sua execução saia do estado onde se encontra.

A inspeção do código nas iterações é uma tarefa fundamental para construção da aplicação, pois os modelos construídos não geram código em perfeito estado para a execução das funcionalidades do sistema, tendo que gastar um tempo na implementação nos métodos das classes. Ao final desta iteração é realizada uma revisão dos artefatos produzidos ao longo do processo de desenvolvimento e o restante da implementação é feita chegando ao fim o desenvolvimento da aplicação.

A seguir é mostrado uma tabela contendo um resumo dos processos e atividades envolvidas nessa terceira iteração para desenvolvimento da aplicação usando MDD Ágil:

PRÁTICAS ÁGEIS	**MDD**	**MDD ÁGIL**
Projeto da Iteração	Construção do modelo PIM que vai compor as funcionalidades do sistema	Construção do terceiro modelo executável
Terceira Iteração	Construção de modelos interligados	Construção de componentes essenciais para o modelo executável e ligação entre os modelos
Incremento	Geração do código do PIM e do modelo de estados para plataforma Java	Execução dos modelos para gerar código.
Inspeção do código	Verificação do código gerado pelo modelo	
Implementação	Código que irá compor as funcionalidades do projeto	Teste da aplicação

Tabela 3: Terceira iteração da aplicação usando MDD ÁGIL

A figura 26 mostra a tela principal da aplicação, que compreende a parte da realização da venda de um produto à um cliente.

Figura 26: Tela de Cadastro da Venda

A tela de cadastro da venda apresenta uma tabela de pesquisa onde mostra os clientes e os produtos que estão cadastrados no sistema e que são pesquisados no momento da venda pelo vendedor, bem como uma tabela com os itens escolhidos pelo cliente. A cada item adicionado o valor total da compra é mostrado na tela.

As transformações realizadas na segunda iteração do projeto para geração do código na plataforma DDL serviu para construção da base de dados usada pelo sistema. Como as linguagens de transformação da primeira e terceira iteração fornecem PSM e código da plataforma Java com assinaturas dos seus métodos, uma inspeção do código foi realizada e o restante da implementação com as funcionalidades CRUD que irão compor o sistema foi desenvolvida.

Entretanto, a ferramenta permite que uma engenharia reversa seja feita no desenvolvimento do projeto. A implementação pode ser realizada normalmente dentro dos códigos gerados pelos PSMs e novos modelos podem ser concebidos com a parte do código que foi implementado. Além disso, o MDD permite que modelos sejam modificados e transformados para gerarem código sem que haja alteração no corpo do código anterior a modificação.

3.6 Pesquisa para avaliação da abordagem

Esta seção apresenta uma pesquisa realizada com o objetivo de avaliar a abordagem MDA Ágil nas empresas que desenvolvem software. A pesquisa foi respondida por 6 pessoas que trabalham em empresas diferentes na área de desenvolvimento de sistema. O questionário foi aplicado a partir da ferramenta Google Docs constando as questões descritas no anexo 1. O trabalho foi anexado junto ao questionário para que os entrevistados tivessem uma referência sobre o tema abordado. As perguntas definidas para o questionário foram agrupadas em 3

blocos. O primeiro com o objetivo de identificar o perfil dos desenvolvedores entrevistados; o segundo para levantar a abordagem atual de desenvolvimento nas empresas pesquisadas; o terceiro com o objetivo de identificar qual o conhecimento que o desenvolvedor entrevistado possui com relação a abordagem MDD Ágil e como estes visualizam as vantagens e desvantagens que esta abordagem pode trazer para a empresa.

O questionário foi aplicado em 6 empresas num total de 6 desenvolvedores. O objetivo é aplicar a pesquisa em empresas diferentes para avaliar se o Desenvolvimento Dirigido a Modelos é adotado nos processos de desenvolvimento das aplicações atualmente. Dentre os entrevistados, 83% possuem de 2 a 5 anos de experiência na área de desenvolvimento de software e 17% tem acima de 10 anos de experiência. Com relação a formação, 50% são graduados, 33% pós graduados e 17% estudantes.

O cenário atual observado na pesquisa indica que aproximadamente 67% dos entrevistados trabalham em empresas que utilizam uma metodologia de desenvolvimento, variando entre as metodologias ágeis e tradicionais. Todos conhecem a linguagem UML e aproximadamente 67% a utiliza de maneira parcial no desenvolvimento de software.

O conceito do Desenvolvimento Dirigido a Modelos é desconhecido por 67% dos entrevistados. Os 23 % restantes já ouviram falar da abordagem, porém nunca utilizaram nos desenvolvimentos de sistemas nas empresas.

Após análise da aplicação construída usando conceitos da abordagem MDA Ágil realizada nesse trabalho, os entrevistados fizeram um levantamento das vantagens e desvantagens que poderiam ser aplicadas em projetos de software nas empresas onde trabalham. Dentre as vantagens as que se destacam são: a aplicação é construída de forma mais prática; o nível de abstração dos modelos facilita a comunicação com os Stakeholders, pois melhora o entendimento do domínio do problema e a não exigência do conhecimento da linguagem de programação em boa parte do desenvolvimento. Dentre as desvantagens a que mais se repetiu foi a inspeção do código produzido pelos modelos de transformação. Segundo os entrevistados isso poderia ocasionar atrasos nas entregas do software aos clientes.

Contudo, após análise dos benefícios e problemas que o MDA Ágil pode trazer no desenvolvimento de sistemas, 67% dos entrevistados falaram que concordariam em aplicar a metodologia nas empresas onde trabalham, justificando que a abordagem poderia aumentar a produtividade no desenvolvimento do software. No entanto, dos 13% restantes, metade não

concordaria por ser algo novo e a outra metade não concordaria pelo fato da abordagem ser mais ligada aos padrões da orientação a objeto, justificando que a empresa onde trabalha é utilizada um tipo de linguagem procedural no desenvolvimento do software.

3.7 Considerações Finais

Durante o desenvolvimento da aplicação alguns aspectos da integração da abordagem MDA com o Framework de Práticas Ágeis e características dos Métodos Ágeis foram notados.

O exemplo escolhido para conciliar a abordagem MDA com Métodos Ágeis e Framework de Práticas Ágeis é simples e contém poucas funcionalidades. Contudo, serviu de base para mostrar como integrar a construção e transformação dos modelos envolvendo processos MDA com a característica de um desenvolvimento ágil, adotando um processo iterativo e incremental e construção de modelos executáveis como base para desenvolver a aplicação. Mesmo com esse exemplo, é possível mostrar que as construções de softwares, ainda que sejam complexos podem adotar práticas da MDA para desenvolvimento ágil de sistemas, com intuito de apresentar a construção do projeto de uma forma que fique mais clara tanto para a equipe de desenvolvimento como para o cliente. Também é possível verificar que o código gerado pelos modelos seria insuficiente para a execução da aplicação, independentemente da complexidade do sistema.

O código gerado durante o desenvolvimento foi de baixa qualidade devido às transformações disponíveis. Modelos comportamentais podem ser construídos e gerados para que métodos mais complexos sejam gerados automaticamente. Esse ainda é um campo em aberto, mas existem pesquisas relacionadas à construção de linguagens de transformação que gerem código em perfeito estado, com o intuito de reduzir o tempo que o desenvolvedor gasta na etapa de implementação da aplicação.

Com relação à pesquisa adotada é possível observar que apesar das empresas que desenvolvem software adotarem a UML como parte no desenvolvimento das aplicações, ainda sim a metodologia é mal aplicada. Boa parte dos desenvolvedores possuem um nível de imaturidade considerável em relação a modelagem dos sistemas e muitos deles desconhecem sobre o Desenvolvimento Dirigido a Modelos. Além disso é importante ressaltar que se a empresa não adota um processo de desenvolvimento com uma linguagem de modelagem bem definida fica difícil adotar o MDD. É preciso uma mudança cultural e uma reestruturação da forma como as empresas trabalham para que se possa adotar o MDD.

4 CONCLUSÃO

Este trabalho teve como objetivo analisar a viabilidade da integração do Desenvolvimento Dirigido a Modelos com os Métodos ágeis, apontando os benefícios e problemas da integração.

O experimento realizado com o Framework de Práticas Ágeis e a abordagem MDA apontou que é possível conciliar práticas adotadas pelos métodos ágeis com processos de construção da abordagem MDA, pois diversas práticas ágeis foram aplicadas com sucesso no processo de desenvolvimento com MDA. Por exemplo a descrição das funcionalidades com a construção do modelo de casos de uso do sistema, o projeto da iteração proporcionado pelos processos de transformações entre os modelos e inspeção do código ao final de cada iteração.

Esta integração é viável mas não é completa, pois nem todas as práticas adotadas em cada atividade são passíveis de integração com o MDA como por exemplo as escritas dos testes de unidade de aceitação, realização de refactoring e integração paralela ao desenvolvimento. A conciliação dessas práticas do Framework com a abordagem MDA varia para cada domínio da aplicação.

Atualmente há uma carência com relação as tecnologias que contemple o desenvolvimento de uma aplicação MDA em sua plenitude, o que faz com que haja uma análise de qual ferramenta se deva adotar para conciliar esta abordagem com as práticas ágeis. É preciso adequar custo com as funcionalidades que a ferramenta possui para uma aplicação MDA de forma adequada. É importante que a produção realizada pela ferramenta seja maior do que o custo para adquiri-la e que o tempo para obter o conhecimento necessário para utilização dela possa prover bons benefícios para construção do projeto.

Dentre os benefícios observados destacam-se a utilização de modelos executáveis com maior nível de abstração na construção do sistema, não obrigatoriedade de conhecimento da linguagem de desenvolvimento utilizada no projeto, construção de modelos que ajudam a descrever as funcionalidades do sistema e os artefatos que são produzidos durante o desenvolvimento são os mesmos utilizados para documentar o projeto.

Alguns problemas também foram identificadas com o uso de MDA Ágil. Por exemplo, as práticas do framework ágil, como a inspeção do código, devem estar presentes durante o

desenvolvimento da aplicação pois a tecnologia atual gera código ineficiente para uma execução completa da aplicação. É importante salientar que se faz necessário a produção de artefatos para transformações completas entre os modelos. Além disso, o framework possui limitações na integração de algumas práticas ágeis pertencentes no corpo de suas atividades com o MDA. Outras maneiras podem ser adotadas a fim de manter a conciliação das abordagens. Para isso conceitos com UML executável, intrínsecos ao MDA Ágil, são aplicados no desenvolvimento da aplicação. Contudo, o desenvolvimento de todos componentes que integra o modelo executável ainda se constitui como uma atividade complexa passiva de ser construída durante a aplicação.

Como sugestão de trabalhos futuros podem ser estudadas as técnicas de simulação de modelos e geração de casos de teste para possibilitar inclusão da orientação a testes dos métodos ágeis na integração com MDA. A simulação de modelos pode antecipar erros de concepção aumentando a qualidade dos sistemas desenvolvidos. Da mesma forma, testes podem ser gerados a partir da especificação para serem submetidos ao sistema.

Adicionalmente é importante realizar estudos mais aprofundados de desenvolvimento de software com a integração sugerida para desenvolver linguagens de transformação que gerem modelos com alto nível de detalhes, onde possam gerar código em perfeito estado reduzindo o tempo na sua inspeção.

REFERÊNCIAS

AGUIAR, Fábio. **Introdução a Métodos Ágeis de Desenvolvimento de Software**. Disponível em: <http://www.slideshare.net/fabiogr/metodos-ageis-7831957#btnPrevious>. Acesso em 13 dezembro 2012.

AGUIAR, Guilherme. **Uso de MDA em um Framework para Seleção de Práticas Ágeis**. Disponível em: <http://projetos.inf.ufsc.br/arquivos_projetos/projeto_1232/TCC-Guilherme_Aguiar-VersaoFinal.pdf>.

ALVAREZ, Diego Perez; **Proposta de um Processo Ágil para Projetos com um único Desenvolvedor**. Florianópolis. 2010.

BASSO, Fabio Paulo; PILLAT, Raquel Mainardi. **Um Relato de Experiência no Desenvolvimento Ágil de Sistemas** MDA.Porto Alegre 2010.

BECK, K.; COCKBURN, A.; JEFFRIES, R.; HIGHSMITH, J., **Agile Manifesto**. Disponível em <http://www.agilemanifesto.org>, Ano: 2001. Acesso em 11dezembro 2012.

CALIARI, Giuliano Luz P. Transformações e Mapeamentos da MDA e sua implementação em três ferramentas.Dissertação apresentada a Escola Politécnica da Universidade de São Paulo. São Paulo,2007.

FARDIM, Vinícius Lima. **Processos Tradicionais versus Processos Ágeis**: Uma Proposta de Processo para Desenvolvimento de Software. 2009. Disponível em: <http://pmies.org.br/clickadmin/midias/data/processos_tradicionais_vs_processos_ageis_desenvolvimento_software.pdf>. acessado 10.Outubro.2013.

FIGUEIREDO, Eduardo. **Desenvolvimento Dirigido por Modelos**. Disponível em: <http://homepages.dcc.ufmg.br/~figueiredo/disciplinas/2012a/reuso/reuso-aula20.pdf>. Acesso em: 11 dezembro 2012.

GRANDO, Nei. **Metodologias Ágeis no desenvolvimento de Projetos de Software**. Disponível em: < http://neigrando.wordpress.com/2010/09/06/metodologias-ageis-no-desenvolvimento-de-projetos-de-software/>. Acesso em: 11 dezembro 2012.

MACIEL, Rita Suzana P.. **Moderne**. Disponível em: <http://homes.dcc.ufba.br/~ritasuzana/moderne>. Acesso em: 11dezembro 2012.

MACIEL, Rita Suzana P.; MAGALHÃES, Ana Patrícia F.; SILVA, Bruno C. da; ROSA, Nelson Souto. **An Integrated Approach for Model Driven Process Modeling and Enactment**. Salvador: UFBA, [s.d].disponível em :< http://www.lasid.ufba.br/publicacoes/artigos/ acessado em 03.julho de 2014.

MACIEL, Rita Suzana P.; SILVA, Bruno C. da; MAGALHÃES, Ana Patrícia F.; GOMES, Ramon; ARAÚJO, Filipe. **Model Driven Process Centered Software Engineering Environment.** Salvador. BA.2013.

NEGRE, Naara Soares. **Model-Driven Development – Transformação de Modelos**. Monografia apresentada ao Curso de Mestrado em Ciência da Computação, do Centro de Informática (CIn), da Universidade Federal de Pernambuco (UFPE),Recife. PE. 2014.

NETO, David Fernandes; FORTES, Renata P. de Matos; **CoMDD**: uma abordagem colaborativa para auxiliar o desenvolvimento orientado a modelos. Disponível em: <http://www2.icmc.usp.br/~posgrad/geral/artigos2012/Artigo_David_Fernandes_Neto>.

NEWSLETTER, n° 54, fev. 2006. **Vantagens do Desenvolvimento Orientado a Modelos**. Disponível em: < http://www.sinfic.pt/SinficNewsletter/sinfic/Newsletter54/Dossier4.html>. Acesso em: 11 dezembro 2012.

SOARES, Michel dos Santos. Comparação **entre Metodologias Ágeis e Tradicionais para o Desenvolvimento de Software**. 2004. Disponível em: <http://www.dcc.ufla.br/infocomp/artigos/v3.2/art02.pdf>.

MELLOR, Stephen J. [et.all]. **Mda Destilada Princípios da Arquitetura Orientada por Modelos**. São Paulo: Ciência Moderna, 2005.

SOUZA, Thiago Silva de. ***Model Driven Architecture*** – Conceitos Fundamentais. Disponível em: <http://www.linhadecodigo.com.br/ArtigoImpressao.aspx?id=1953>. Acesso em: 11 dez. 2012.

VENTORIN, Alessandro José. **Principais Problemas Relacionados ao Desenvolvimento de Sistemas**. Disponível em: <http://www.univen.edu.br/revista/n008/PRINCIPAIS%PROBLEMAS.20RELACIONADOS **.pdf>.**

ZHANG, Yuefeng; PATEL, Shailesh**; Agile Model-Driven Development in Practice.** [s.l].2010**.** Acessado em 02.07.2014.>