UNIVERSIDADE ESTADUAL PAULISTA
“JÚLIO DE MESQUITA FILHO”
Instituto de Geociências e Ciências Exatas – IGCE
Curso de Bacharelado em Ciências da Computação

THALES EDUARDO NAZATTO

DESENVOLVIMENTO DE SISTEMAS ORIENTADO A MODELOS: O Framework Xtext e sua contextualização no Model-Driven Development

Trabalho realizado sob orientação do Prof. Dr. Frank José Affonso, DEMAC/IGCE
Período: 15.07 a 03.12.2011

Rio Claro-SP
2011

DESENVOLVIMENTO DE SISTEMAS ORIENTADO A MODELOS: O Framework Xtext e sua contextualização no Model-Driven Development

Trabalho de Conclusão do Curso, modalidade Trabalho de Graduação, apresentado, no 2º semestre de 2011, à disciplina ES/TG do Curso de Bacharelado em Ciências da Computação, período Noturno, do Instituto de Geociências e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", campus de Rio Claro, para apreciação segundo as normas estabelecidas pelo Conselho do Curso, em 27.11.2007.

Aluno: Thales Eduardo Nazatto

Orientador: Prof. Dr. Frank José Affonso

Rio Claro-SP
2011

005 Nazatto, Thales Eduardo
N334d Desenvolvimento de sistemas orientado a modelos: o Framework Xtext e sua contextualização no Model-Driven Development / Thales Eduardo Nazatto. - Rio Claro : [s.n.], 2011
71 f. : il., figs., tabs.

Trabalho de conclusão de curso (bacharelado - Ciências da Computação) - Universidade Estadual Paulista, Instituto de Geociências e Ciências Exatas
Orientador: Frank José Affonso

1. Engenharia de software. 2. Programação Orientada a Aspectos. 3. Domain-Specific Languages. I. Título.

Ficha Catalográfica elaborada pela STATI - Biblioteca da UNESP
Campus de Rio Claro/SP

THALES EDUARDO NAZATTO

DESENVOLVIMENTO DE SISTEMAS ORIENTADO A MODELOS

Estágio Supervisionado ou Trabalho de Graduação apresentado ao Instituto de Geociências e Ciências Exatas - Câmpus de Rio Claro, da Universidade Estadual Paulista Júlio de Mesquita Filho, para obtenção do grau de Bacharel em Ciências da Computação.

Comissão Examinadora

Prof. Dr. Frank José Affonso (orientador)

Prof. Dr. Leandro Alves Neves

Prof(a). Dr(a). Maria Cecília V. S. Carneiro

Rio Claro, 12 de dezembro de 2011.

Assinatura do(a) aluno(a)

assinatura do(a) orientador(a)

AGRADECIMENTOS

Primeiramente, à minha família, pela dedicação que tiveram em me criar e pela liberdade de escolha para fazer o que gosto.

Ao meu orientador, Prof. Dr. Frank José Affonso, pela disponibilidade, paciência, compreensão e forma de me guiar na confecção deste trabalho.

A todos os meus atuais companheiros de trabalho pela experiência adquirida, a qual me auxiliou muito nesse trabalho, e em especial a meus chefes Alberto Margarido, Anderson Lazzari e Ronaldo Cerqueira pela compreensão, mesmo que isso pudesse acarretar em momentos mais difíceis.

A todas as pessoas com quem morei: Adilson Tiritan Ramos (Osama), Diego Sansão Monteiro (Rambo), Fabio Castello Novo (Castello), Gabriel Carlos Oliveira de Carvalho (Gozada), Gabriel Ferrari Mariano (Sapão), Hugo Pereira de Godoy (Grego), José Fernando da Silva Cruz (Fiscal), Paulo Renato Corrêa Costa (Papo 10), Raphael Caminoto (Bile), Renan Aleixo Paganatto (Renan) e Thiago Romeiro Tavares (Feijão), pelos momentos de alegria e de tristeza e ao aprendizado que puderam me proporcionar.

A todas as outras pessoas que conheci neste campus, pela troca de novas experiências.

E finalmente, a Deus, pela fé que me ajudou a priorizar os estudos e a continuar motivado nesta reta final da graduação.

“Todas as coisas são difíceis antes de se tornarem fáceis”

Thomas Fuller

RESUMO

A Engenharia de Software se originou com a motivação de produzir componentes em massa para aumento de produtividade na produção de sistemas. Desde sua origem, foram propostos inúmeros estudos sobre o assunto a medida que novidades na criação de novos sistemas, como a Programação Orientada a Objetos e a Programação Orientada a Aspectos, foram criadas e metodologias foram elaboradas para controlá-las de maneira eficiente. Entretanto, anos de estudos na área não foram suficientes para criar uma metodologia para reúso de artefatos de software realmente eficiente e fácil o suficiente para ser amplamente difundida. Dado isso, o Model-Driven Development (MDD) tenta promover isso utilizando a modelagem de sistemas como referência, tornando-se parte dele e estabelecendo um enorme ganho de produtividade. Uma de suas abordagens é o chamado Model-Driven Software Development (MDSD), que se foca no aperfeiçoamento das práticas de desenvolvimento de sistemas e utiliza Domain-Specific Languages (DSL) para tal fim. Neste Trabalho de Conclusão de Curso é usado o Xtext como ferramenta para provar a produtividade e eficiência desta abordagem, e para isso foram feitos estudos bibliográficos sobre a abordagem e a ferramenta, além de mostrar a metodologia utilizada e um estudo de caso para mostrar resultados e conclusões a respeito deste trabalho.

Palavras-chave: Engenharia de Software, Programação Orientada a Aspectos, Model-Driven Software Development, Domain-Specific Languages, Xtext

ABSTRACT

The Software Engineering originated with the motivation to mass produce components for increased productivity in production systems. Since its origins, numerous studies have been proposed on the subject as new features in the creation of systems, like the Object-Oriented Programming and Aspect-Oriented Programming, have been established and methodologies have been developed to control them efficiently. However, years of studies in the area were not sufficient to create a methodology for reusing software artifacts really efficient and easy enough to be widespread. Given this, the Model-Driven Development (MDD) is trying to promote it using the modeling of systems as a reference, becoming part of it and establishing a huge productivity gain. One of his approaches is called Model-Driven Software Development (MDSD), which focuses on improving the practices and systems development using Domain-Specific Languages (DSL) for this purpose. In this Final Paper, Xtext is used as a tool to prove the productivity and efficiency of this approach, and for that bibliographic studies were made on the approach and the tool, and show the methodology and a case study to demonstrate results and conclusions regarding this work.

Keywords: Software Engineering, Aspect-Oriented Programming, Model-Driven Software Development, Domain-Specific Languages, Xtext

LISTA DE ILUSTRAÇÕES

Figura 1. Arquitetura de metamodelagem com 4 níveis proposta pelo OMG (Quintero & Valderrama, 2007)..17
Figura 2. Processo de transformação entre PIM e PSM (Issa, 2006)................................20
Figura 3. Um PIM gerando diversos PSMs (Belix, 2008)...21
Figura 4. Principais elementos do MDD (Lucrédio, 2008)...23
Figura 5. Exemplo de instruções feito em SQL. (10)..26
Figura 6. Diferenças entre logging e reconhecimento de URL no Apache Tomcat (11)....27
Figura 7. Processo de recomposição de um sistema orientado a aspectos (Kiczales, 1997) 29
Figura 8. Padrão proxy para controle de transação (Santos, 2008)..................................32
Figura 9. Controle de transação utilizando aspectos (Santos, 2008)................................32
Figura 10. Padrão proxy para padrão DAO (Santos, 2008)...33
Figura 11. Padrão DAO utilizando aspectos (Santos, 2008)...33
Figura 12. Esquematização básica de transformação e modelagem do EMF (Quintero & Valderrama, 2007)..37
Figura 13. Gramática válida pelo Xtext (17)...38
Figura 14. Exemplo de um arquivo `.xpt` para o Xpand. (18)..40
Figura 15. Exemplo de geração de arquivos no Xtend..40
Figura 16. Exemplos para definir templates no Xtend..41
Figura 17. Etapas da abordagem MDSD utilizando o framework Xtext..........................42
Figura 18. “Erros” de geração de código..48

LISTA DE TABELAS

Tabela 1. Comparação entre código escrito e código gerado por MDSD............................49

Tabela 2. Comparação entre código escrito e código gerado pelo Eclipse..........................49

Tabela 3. Linhas de código implementadas para a geração de código...............................50

LISTA DE ABREVIATURAS E SIGLAS

MDA: Model-Driven Architecture

OMG: Object Management Group

UML: Unified Modeling Language

MOF: Meta Object Facility

CWM: Common Warehouse Metamodel

XMI: XML Metadata Interchange

XML: eXtensible Markup Language

CIM: Computation Independent Model

PIM: Plataform Independent Model

PSM: Plataform Specific Model

DAO: Data Access Object

MDD: Model-Driven Development

SF: Software Factories

MDSD: Model-Driven Software Development

DSL: Domain-Specific Language

GPL: General Purpose Language

SQL: Structured Query Language

SGBD: Sistema Gerenciador de Banco de Dados

POO: Programação Orientada a Objetos

AOP: Aspect-Oriented Programming

IDE: Integrated Development Environment

AJDT: AspectJ Development Tools

EMF: Eclipse Modeling Framework

API: Application Programming Interface

EBNF: Extended Backus-Naur Form

LL: Left-to-right Leftmost

ANTLR: Another Tool for Language Recognition

MWE: Modeling Workflow Engine

SUMÁRIO

Página

LISTA DE ILUSTRAÇÕES....7
LISTA DE TABELAS....8
LISTA DE ABREVIATURAS E SIGLAS....9
SUMÁRIO....11

1 INTRODUÇÃO....13
1.1 Contexto e Motivação....13
1.2 Objetivo....13
1.3 Estrutura do Documento....14

2 ESTUDO BIBLIOGRÁFICO....15
2.1 Model-Driven Architecture (MDA)....15
2.1.1 Meta Object Facility (MOF)....15
2.1.2 Modelagem....17
2.1.3 Mapeamento....18
2.1.4 Transformação....19
2.1.5 Propósitos....21
2.2 Model-Driven Development (MDD)....22
2.2.1 Model-Driven Software Development (MDSD)....23
2.2.2 Software Factories (SF)....24
2.3 Domain-Specific Languages (DSL)....24
2.4 Programação Orientada a Aspectos (AOP)....26
2.4.1 Composição Do Sistema....28
2.4.2 Fases....29
2.4.3 Elementos....30
2.4.4 Aplicações....31
2.5 Eclipse IDE....34

3 O FRAMEWORK XTEXT....36
3.1 Eclipse Modelling Framework (EMF)....36
3.2 Xtext....37
3.2.1 Modeling Workflow Engine (MWE)....38
3.3 Xtend....39

4 METODOLOGIA....42

5 ESTUDO DE CASO....45
5.1 Especificação Dos Requisitos....45
5.2 Desenvolvimento Da Gramática E Geradores....45
5.3 Modelagem Do Sistema....47
5.4 Geração De Código....48
5.5 Resultados E Considerações....48

6 CONCLUSÃO....51

7 REFERÊNCIAS....53

8 APÊNDICES..55
APÊNDICE A: Requisitos do Sistema...56
APÊNDICE B: Gramática, Geradores e Código Gerado...57
APÊNDICE C: Código da Aplicação...69

1 INTRODUÇÃO

1.1 CONTEXTO E MOTIVAÇÃO

A Engenharia de Software teve sua origem em 1968, quando foi realizada uma conferência na OTAN onde foi proposto o uso de componentes em massa para evitar ou remediar a chamada *"crise do software"* **(Lucrédio, 2008)**. Com isso, foram propostos inúmeros estudos sobre o assunto a medida que novidades na criação de novos sistemas, como a Programação Orientada a Objetos e a Programação Orientada a Aspectos, foram criadas e metodologias foram elaboradas para controlá-las de maneira eficiente.

Entretanto, anos de estudos na área não foram suficientes para criar uma metodologia para reúso de artefatos de software realmente eficiente e fácil o suficiente para ser amplamente difundida. A proposta do *Model-Driven Development* (MDD) é promover isso utilizando a modelagem de sistemas como referência, tornando-se parte dele. Com base na transfomação de modelos para refinamento dos mesmos e geração de código final, o MDD se estabelece como uma abordagem com enorme ganho de produtividade.

Uma de suas abordagens é o chamado *Model-Driven Software Development* (MDSD), que se foca no aperfeiçoamento das práticas de desenvolvimento de sistemas e utiliza *Domain-Specific Languages* (DSL) para tal fim. Utilizando o Xtext para criar essa DSL, é possível fazer uma estrutura coerente e geradores de código para criar uma ferramenta sólida e altamente produtiva.

1.2 OBJETIVO

O trabalho objetiva realizar um estudo de abordagens do *Model-Driven Development* (MDD), como *Model-Driven Architecture* (MDA), e verificar se o uso de *frameworks* como o Xtext "casam" com sua proposta e possibilitam o seu uso de maneira viável. Para isso, foram realizados estudos bibliográficos sobre elas e desenvolvida uma metodologia que possibilita a prática das mesmas.

Foi desenvolvido também um estudo de caso com o desenvolvimento de uma aplicação

baseado na metodologia utilizada, verificando a viabilidade de seu uso no mercado atual. O mesmo pode ser aperfeiçoado em momentos futuros para abranger todas as soluções necessárias aos problemas dos desenvolvedores.

1.3 ESTRUTURA DO DOCUMENTO

Esta monografia está organizada em cinco capítulos.

O capítulo 2 abordará as referências bibliográficas, tratando todos os conceitos fundamentais utilizados: *Model-Driven Architecture* (MDA), *Model-Driven Development* (MDD), *Domain Specific-Languages* (DSL), *Aspect-Oriented Programming* (AOP) e o uso da IDE Eclipse.

O capítulo 3 destina-se à explicação em específico do *framework* utilizado: O Xtext. Embora possa ser uma abordagem mais conceitual, o capítulo foi separado das referências bibliográficas por ter um foco maior nesse trabalho.

O capítulo 4 mostra a metodologia aplicada para o trabalho, mostrando como a junção dos capítulos anteriores pode ter uma função prática.

O capítulo 5 conterá a implementação de um estudo de caso utilizando o *framework* Xtext detalhando um passo-a-passo da metodologia abordada.

O capítulo 6 apresenta os problemas encontrados, as conclusões obtidas e o que pode ser sugerido em trabalhos futuros.

2 ESTUDO BIBLIOGRÁFICO

2.1 MODEL-DRIVEN ARCHITECTURE (MDA)

Segundo **Issa (2006)**, *Model-Driven Architecture* (MDA) é um conceito criada pelo *Object Management Group* (OMG) dirigido para aplicações feitas por empresas. Ela é feita não apenas para padronizar a modelagem em si e seu desenvolvimento, mas também a consistência entre modelos e código do sistema.

OMG (2003b), citado por Belix (2006) defende a visão de que a infraestrutura de TI de qualquer organização atual é um sistema de computação distribuída e para o manuseio correto de suas informações seu acesso deve ser feito através de divisões (departamentos, companhias, suprimentos e serviços de fornecedores, entre outras). Baseando-se nisso, o OMG afirma que as aplicações consideradas *standalone* (sem integração ou comunicação com algum outro serviço) não possuem mais espaço na infraestrutura do mercado atual. O grupo também define que os desenvolvedores de software apenas seguem o que foi especificado, ao invés de pensar na constante necessidade de integração e atualização de um sistema, e com isso a maior parte dos sistemas atuais ainda é feita sem considerar as constantes mudanças de infraestrutura e requisitos. Ao mesmo tempo, a medida em que modelagem e o código de um sistema começaram a convergir, também houve o aumento de interesse em modelagem de dados e aplicações.

Com base nos princípios citados, o OMG recomenda que o desenvolvimento de um software precisa de uma atenção especial em seu projeto e que sua modelagem, baseada em padrões determinados pelo grupo (UML, MOF e CWM, por exemplo), pode automatizar a construção de aplicações, bancos de dados e integração entre diferentes aplicações. Tomando em consideração este conjunto de fatores, a defesa pelo uso de MDA pelo OMG é bastante plausível.

2.1.1 META OBJECT FACILITY (MOF)

Segundo **Lucrédio (2008)**, a base do MDA é o MOF (*Meta Object Facility*), um meta-metamodelo que serve de base para a definição de todas as linguagens de modelagem. Com a padronização utilizada no MOF, as ferramentas de modelagem e transformação podem trabalhar em conjunto.

O MOF consiste em um padrão orientado a objetos que permite a definição de classes com atributos e relacionamentos, sendo comparável ao diagrama de classes da UML. Também define uma interface única para acessar os dados dos modelos, possibilitando a manipulação dos mesmos através de métodos, e regras para criar as peculiaridades de cada metamodelo, sendo possível atribuí-las a atributos, classes, relacionamentos, entre outros. Dessa forma, é possível definir uma padronização dos modelos através de pouco esforço.

Sendo um meta-metamodelo, o MOF se firma nos fundamentos e princípios de metamodelagem. De acordo com **Quintero & Valderrama (2007) e Lucrédio (2008)**, o OMG define 4 níveis diferentes de modelagem: O sistema em si (nível M0), correspondendo aos dados; o modelo do sistema (nível M1), correspondendo aos dados que descrevem os dados; o metamodelo (nível M2), que define os modelos em si; e o meta-metamodelo (nível M3), que define os metamodelos/linguagens de modelagem. A UML está no nível M2, e o MOF pode descrever sua sintaxe. Como um nível inferior é instância de um outro nível superior, linguagens de modelagem e padrões definidos pelo MOF podem ser usadas para uma proposta construída através de MDA. A Figura 1 mostra como são divididos esses níveis.

O MDA também define o XMI (*XML Metadata Interchange*) para definir representações dos níveis de metamodelagem, podendo ser representados dados e metadados. O XMI possui uma estrutura de fácil interpretação e a possibilidade de aplicar transformações **(Lucrédio, 2008)**. Enquanto o MOF é o núcleo conceitual do MDA, o XMI é o documento padrão dessa especificação, tornando a abstração do MOF mais concreta e de fácil entendimento para o uso dessa arquitetura.

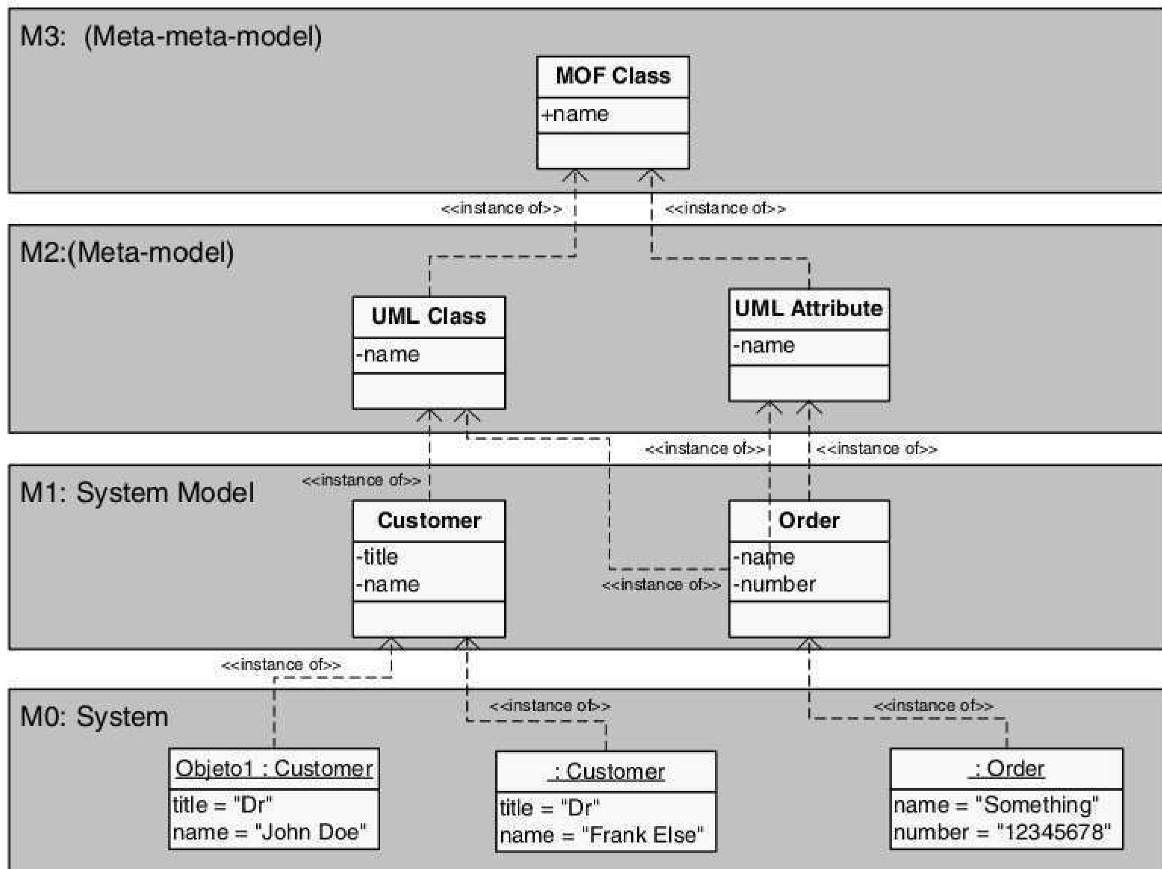

Figura 1. Arquitetura de metamodelagem com 4 níveis proposta pelo OMG (Quintero & Valderrama, 2007)

2.1.2 MODELAGEM

A modelagem do MDA tem como base 3 modelos principais: *Computation Independent Model* (CIM), *Plataform Independent Model* (PIM) e *Plataform Specific Model* (PSM). Estes três modelos são tratados como camadas da arquitetura, passando por transformações de um modelo para outro até chegar na geração de código própriamente dita. A seguir, é apresentada uma breve descrição dos modelos citados.

O *Computation Independent Model* (CIM) não é um modelo técnico para a geração do código em si, e sim um modelo para especificar o domínio do sistema. Ele exerce um papel apenas para especialistas do domínio de negócio e especialistas em design e construção dos artefatos.

Por não ser um modelo que envolve um conhecimento técnico específico, não são definidas transformações de um CIM para outro nível de modelagem. Esse modelo é útil como uma ponte entre clientes e desenvolvedores: por ser independente de software, é acessível a ambos. A modelagem de um sistema começa a ter uma abordagem técnica a partir do próximo modelo: o PIM, que pode ter um CIM como base principal de construção. **(Belix, 2006)**

O *Plataform Independent Model* (PIM) é o início de todo projeto em MDA. Geralmente expresso em UML, o PIM é construído por especialistas de modelagem e de negócio, sendo feito para expressar as regras de negócio e funcionalidades de um sistema da forma mais objetiva e universal possível, de modo a não serem distorcidas pela tecnologia. Por ser independente de plataforma, o PIM não perde seu valor, podendo ser reutilizado para outro sistema e apenas será mudado caso as regras de negócio também sejam alteradas.

Apesar de ter esse propósito, o PIM em si não é "perfeito" em sua teoria, e acaba incorporando aspectos tecnológicos. Conceitos como persistência, nível de segurança e suporte a transações podem ser representados pelo modelo, possibilitando um mapeamento mais preciso para o PSM. Não há uma distinção precisa sobre o que é dependente e independente de plataforma, podendo acarretar problemas pela separação de interesses entre os modelos. Por isso, um cuidado especial ao mapear esse modelo se mostra necessário. **(Belix, 2006)**

O *Plataform Specific Model* (PSM) descreve detalhes de implementação, tecnologia e plataforma de um PIM **(Issa, 2006)**, e gerado a partir deste através de um conjunto de transformações. Como um sistema possui em sua implementação o uso de diferentes plataformas (banco de dados, visualização web, servidor de aplicações, serviços web, entre outros), um PIM pode gerar através de suas transformações vários PSM diferentes.

Um PSM pode fornecer um nível diferente de detalhamento dependendo de sua funcionalidade. Para que um PSM seja adequado para gerar código para um sistema, ele precisa fornecer todas as informações necessárias para a construção de um sistema. O mesmo PSM pode ser utilizado também para refinamento da própria modelagem: Através de um conjunto de PSMs e transformações, ele pode refinar o PSM que será utilizado para gerar o código da implementação, organizando a modelagem e aumentando sua capacidade de manutenção caso algum aspecto da modelagem precise ser modificado ou acrescentado. **(Belix, 2006)**

2.1.3 MAPEAMENTO

Com esses modelos, para a geração de código em MDA é necessário o mapeamento e a transformação dos mesmos. O mapeamento é a fase intermediária, definindo as regras de transformação entre um modelo e outro. **OMG (2001a), citado por Belix (2006)** salienta: "*Os mapeamentos são utilizados pelas transformações entre PIM e PSM*". Existem duas maneiras diferentes para fazer mapeamentos **(Belix, 2006)**:

- **Mapeamento de tipos do modelo:** Especifica o mapeamento de tipos especificados no PIM para tipos especificados no PSM. Por exemplo, é possível definir no PIM classes específicas para uma arquitetura (DAO, Repository, Web Services, entre outros) e no PSM definir essa classe em um pacote, estrutura de pastas ou padronização de nome específica.

- **Mapeamento de instâncias do modelo:** Esse mapeamento entra quando é necessário uma especificidade maior em relação ao mapeamento anterior. Para isso, são necessárias marcas no PIM representando conceitos no PSM, processo chamado de *marking a model*. Com essa marca, é possível definir quantos PSM diferentes um PIM vai gerar.

2.1.4 TRANSFORMAÇÃO

A transformação é a geração de um modelo-fonte para um modelo-alvo. Ambos os modelos são representados por linguagens específicas para cada modelo, entretanto é possível transformações na mesma linguagem. As transformações utilizam os mapeamentos, mas contém outras informações, como a condição para a transformação e a linguagem de ambos os modelos. Nesta etapa, o principal ponto positivo é o detalhamento dos modelos vindo das transformações, ganhando tempo e, consequentemente, produtividade para os desenvolvedores, que precisam se preocupar menos com a implementação dos modelos e de suas peculiaridades. A Figura 2 mostra uma dessas transformações.

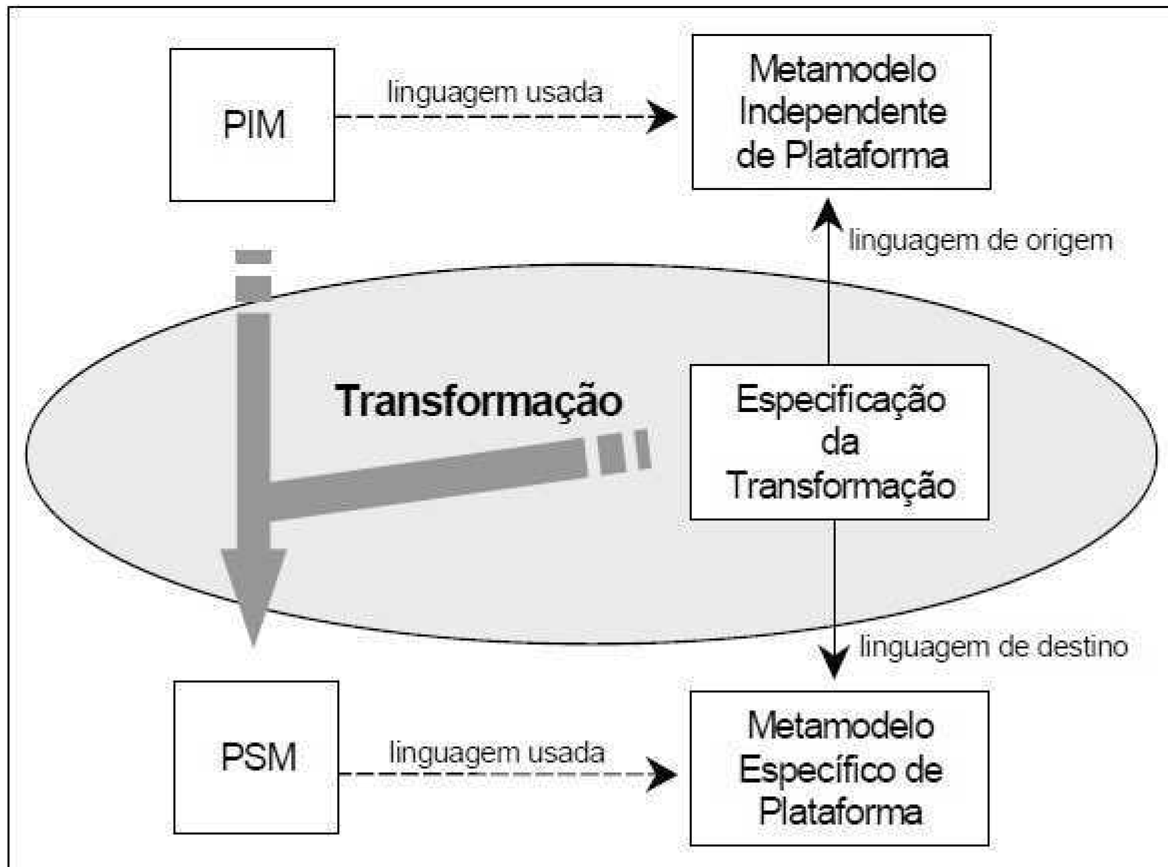

Figura 2. Processo de transformação entre PIM e PSM (Issa, 2006)

As transformações podem ser classificadas por seus tipos. De acordo com **Belix (2006)**, o OMG define os seguintes tipos de transformação:

- **PIM para PIM:** Transformação para aprimoramento, filtragem e especialização de modelos, dentro de suas limitações.

- **PIM para PSM:** Transformação para a passagem de um modelo convencional para um modelo de ambiente de execução. É necessário um PIM suficientemente refinado para a transformação ser satisfatória.

- **PSM para PSM:** Transformação para refinamento dos modelos, visando aprimoramento dos mesmos.

- **PSM para PIM:** Transformação para extração de modelos a partir de sistemas já implementados (Engenharia Reversa).

É importante lembrar que a transformação de CIM para PIM não é mencionada entre os tipos por essa transformação em específico ser desenvolvida exclusivamente por um ser humano e,

portanto, não há uma interferência direta do computador no processo de transformação, sendo dependente da competência do Engenheiro de Software. As transformações entre PSM e código-fonte existem, mas não há um detalhamento do conceito, apenas citações ao termo. Outro item a ser mencionado é o caso de transformações múltiplas: Um modelo-fonte pode gerar vários modelos-alvos, conforme mostrado na Figura 3.

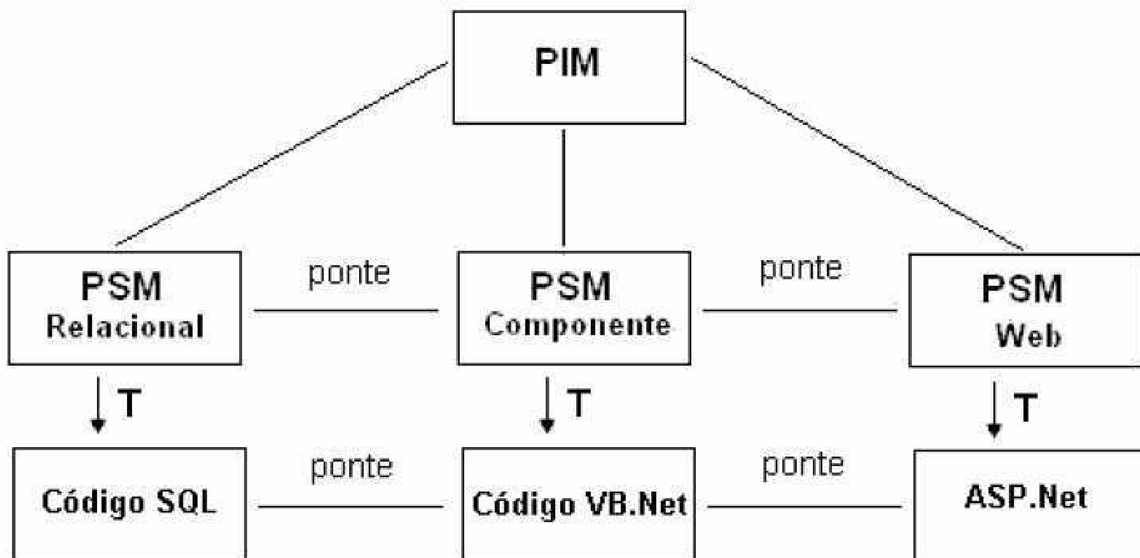

Figura 3. Um PIM gerando diversos PSMs (Belix, 2008)

2.1.5 PROPÓSITOS

Tendo esse processo inteiro em mente, o MDA visa garantir, segundo **Issa (2006)**:

- **Portabilidade:** A complexidade de uma aplicação é diminuída e com uma maior capacidade de reutilização, tendo poucas modificações devido a um domínio diferente de negócio.

- **Interoperabilidade entre plataformas:** É possível, com o uso de um único modelo, gerar código para múltiplos tipos de plataformas (Exemplo: Uma aplicação em código Java e outra em código C#) ao mesmo tempo em que esses códigos funcionem de maneira semelhante ao fazer sua compilação.

- **Independência de plataforma:** Com base na mesma situação acima, é possível escolher qual a melhor opção no momento para aplicar em um certo cliente, e com isso reusar o modelo levando em consideração as suas limitações.

- **Especificidade no domínio:** A modelagem garante um aprofundamento de seu domínio,

que por si só ajudam no desenvolvimento de aplicações baseadas no mesmo.

- **Produtividade:** Com a geração de um código, o MDA garante ao desenvolvedor algo "pronto", fazendo com que o desenvolvedor não perca tempo codificando e corrigindo os próprios erros. Sua arquitetura visa apenas a codificação do que é necessário ser codificado.

Apesar de todos esses benefícios, ainda não há maturidade suficiente para que o MDA se popularize no mercado de trabalho, estando mais presente ainda no campo das pesquisas acadêmicas. Ainda assim, seu potencial é notável e vale ser conferido.

2.2 MODEL-DRIVEN DEVELOPMENT (MDD)

O *Model-Driven Development* (MDD) é uma metodologia que foca a construção de modelos como desenvolvimento de um sistema. Ela é definida por diversas abordagens, que são completamente diferentes entre si embora carreguem algumas semelhanças entre elas.

O MDA é a mais tradicional de suas abordagens, mas existem outras duas abordagens bastante conhecidas: *Software Factories* (SF), defendida fortemente pela Microsoft, e *Model-Driven Software Development* (MDSD), semelhante a MDA, porém mais abrangente. **(Belix, 2006)**

Lucrédio (2008) informa a necessidade de uma ferramenta de modelagem para tal metodologia, utilizada para descrever o domínio, suas regras e especificações. Normalmente, é utilizada uma *Domain-Specific Language* (DSL) para isso, que precisa ser coerente e consistente com o que se deseja fazer com ela. Também é necessária uma ferramenta que defina quais serão as transformações que devem ser feitas com os modelos que forem gerados. Tais ferramentas devem ser intuitivas e de fácil utilização.

Com base nos modelos e nas transformações definidas, são feitas transformações semelhantes às de MDA para geração do código-fonte. A Figura 4 exemplifica esse processo.

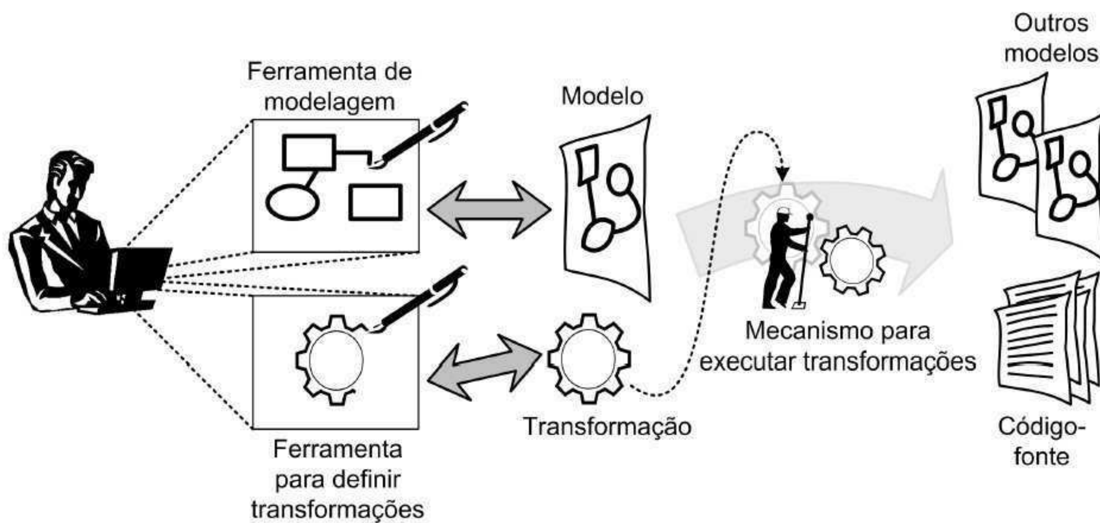

Figura 4. Principais elementos do MDD (Lucrédio, 2008)

2.2.1 MODEL-DRIVEN SOFTWARE DEVELOPMENT (MDSD)

Segundo **Bettin (2004), citado por Issa (2006)**, o MDSD é uma abordagem projetada para projetos com equipes distribuídas maiores do que 20 pessoas. Ele possui um foco muito parecido com o do MDA, se focando em análise de domínio, metamodelagem, geração dirigida por modelos, linguagem de template, projeto de *framework* dirigido pelo domínio e princípios de desenvolvimento de software ágeis.

Há diferenças entre o MDA e o MDSD. Enquanto o MDA tem um foco maior na teoria e na padronização de uma abordagem, utilizando conceitos rígidos e bem definidos, o MDSD é bem mais flexível, focando-se mais nas técnicas de programação a serem utilizadas em conjunto com as ideias e conceitos do MDA para tornar a padronização de uma linha de produção de software uma coisa concreta. **Issa (2006)** detalha algumas dessas diferenças:

- **Metamodelagem:** Enquanto MDA se baseia na fundamentação do MOF, MDSD não se amarra a um metamodelo, sendo baseado nos príncipios que regem a modelagem e a metamodelagem.

- **Abstração de sistemas:** Ambas propõem o aumento do nível de abstração, porém MDA não prevê peculiaridades e notações específicas de empresas em sua abordagem, sendo mais genérica. MDSD, por sua vez, reconhece isso usando DSLs.

- **Conceitos:** MDA se foca mais em ser uma base para as abordagens, mas não define uma metodologia propriamente dita. MDSD se foca mais em utilizar essa base para propor uma

abordagem de desenvolvimento de sistemas.

- **Defesa ao *Open Source:*** Enquanto MDA é totalmente neutro quanto ao uso de ferramentas *Open Source*, MDSD encoraja e considera o uso delas essencial.

2.2.2 SOFTWARE FACTORIES (SF)

Quintero e Valderrama (2007) informa que a abordagem de *Software Factories* (SF) propõe *"o uso de ferramentas extensíveis e configuráveis para automatizar o desenvolvimento e manutenção de diferentes tipos de software, e tal automação é obtida pela composição e configuração de diferentes componentes"*.

Belix (2006) complementa essa ideia comparando o SF com o MDA e o MDSD:

- **Linguagem de Modelagem:** enquanto MDA utiliza padrões como o UML e o MOF e MDSD pode utilizar tanto UML quanto DSL para modelagem, SF não adota o UML, adotando modelos baseados em DSLs.

- **Processo de Desenvolvimento de Sistemas:** MDSD e SF são processos focados ao desenvolvimento de sistemas, enquanto MDA fornece apenas linhas gerais para um processo de desenvolvimento. SF e MDSD também são considerados processos pouco prescritivos – a diferença, no caso, é que no MDSD isso tem a ver com práticas de desenvolvimento ágil.

- **Independência da Plataforma:** MDSD e MDA utilizam todo o conceito de PIM, PSM, seu mapeamento e transformações, enquanto SF não utiliza tal abordagem. É possível produzir software seguro, confiável e rápido a partir de especificações que não levem em consideração dependência de plataforma.

- **Padrões e código aberto:** MDSD e MDA são baseados em código aberto, enquanto SF usa padrões e ferramentas proprietárias.

2.3 DOMAIN-SPECIFIC LANGUAGES (DSL)

Segundo a documentação do Xtext **(Xtext, 2011)**, *Domain-Specific Languages* (DSL) são linguagens de programação pequenas, que se focam num domínio em particular. Um domínio pode ser considerado como um problema em específico, como procurar alguma coisa em algum banco de dados ou o mapeamento de certos dados em alguma situação, como configuração de aplicativos e

coordenadas de mapas.

O oposto de DSL é a GPL, *General Purpose Language*, que são linguagens visando providenciar todos os tipos de soluções por se apoiar nas bases da computação, mas em certos casos não são a melhor indicação para solucionar problemas. Podemos citar como exemplos de GPL C, Python e Java, linguagens que são bastante utilizadas em sistemas atuais.

Ela ainda traz uma metáfora bastante interessante que transmite exatamente a diferença de usos entre DSL e GPL: *"Imagine que você queira remover o núcleo de uma maçã. Você pode utilizar com toda a certeza um canivete suiço para cortá-lo fora, e isso é razoável se você fizer uma vez ou duas. Mas se você precisa fazer isso regularmente pode ser mais eficiente usar um 'apple corer'"*. Em outras palavras, a GPL traz um número N de usos diferentes, porém certas vezes é melhor usar uma DSL caso um desses usos tenha uma recorrência frequente. Existem dois tipos de DSL, mostrados a seguir:

- **DSL internas:** São bibliotecas implementadas para linguagens já existentes, e são consideradas DSLs por ter uma forma particular de ser codificada, comparável a uma linguagem propriamente dita. O objetivo de uma DSL interna é expandir as funcionalidades existentes destas linguagens.

- **DSL externas:** São linguagens completamente novas, escritas especificamente para atender as necessidades do domínio em questão.

O maior exemplo de uma DSL bem-sucedida é o SQL. De acordo com a Wikipedia **(6)**, "*O SQL foi desenvolvido originalmente no início dos anos 70 nos laboratórios da IBM em San Jose(California), dentro do projeto System R, que tinha por objetivo demonstrar a viabilidade da implementação do modelo relacional proposto por E. F. Codd.*". Atualmente, o SQL é a linguagem padrão utilizada em Sistemas Gerenciadores de Bancos de Dados (SGBDs). "*Isto decorre da sua simplicidade e facilidade de uso. Ela se diferencia de outras linguagens de consulta a banco de dados no sentido em que uma consulta SQL especifica a forma do resultado e não o caminho para chegar a ele. Ela é uma linguagem declarativa em oposição a outras linguagens procedurais. Isto reduz o ciclo de aprendizado daqueles que se iniciam na linguagem.*" **(6)**.

```
-- New date and time types
CREATE TABLE t1 (c1 DATE, c2 TIME(3),
    c3 DATETIME2(7) NOT NULL DEFAULT GETDATE(),
    c4 DATETIMEOFFSET CHECK (c4 < CAST(GETDATE() AS
                                   DATETIMEOFFSET(0))));
INSERT INTO t1 VALUES ('0001-01-01', '23:59:59',
    '0001-12-21 23:59:59.1234567','9999-12-31 23:59:59.1234567 -07:00');
SELECT c4, DATEPART(TZOFFSET, c4), DATEPART(ISO_WEEK, c4),
    DATEPART(MICROSECOND, c4) FROM t1;

-- Table Value Parameter
-- Create a user TABLE type
CREATE TYPE myTableType AS TABLE (id INT,
    name NVARCHAR(100), qty INT);
-- Create a stored procedure that accepts a table-variable as a param
CREATE PROCEDURE myProc (@tvp myTableType READONLY)
AS
    UPDATE Inventory SET qty += s.qty
    FROM Inventory AS i INNER JOIN @tvp AS tvp ON i.id = tvp.id
GO

-- Multi-row insert through single insert stmt
INSERT INTO contacts VALUES ('John Doe', '425-333-5321'),
 ('Jane Doe', '206-123-4567'), ('John Smith', '650-434-7869');
```

Figura 5. Exemplo de instruções feito em SQL. (10)

DSLs são parte importante do MDD, pois podem representar aspectos importantes da modelagem com apenas o uso de poucas palavras para tal. Por exemplo, o uso de um padrão em específico pode ser codificado com uma linha indicando seu uso, e a geração de seu código será totalmente modificada por causa da modificação dessa palavra. A própria UML é uma linguagem para esse fim, simplificando e muito a programação de seus modelos devido ao poder que ela proporciona ao desenvolvedor.

2.4 PROGRAMAÇÃO ORIENTADA A ASPECTOS (AOP)

De acordo com **Nelson, citado por Santos (2008)**, o termo "separação de interesses" foi definido por Edsger Dijkstra em 1974 para denotar o princípio que guia a divisão em partes: Todo software lida com uma diferente gama de interesses específicos, como dados e operações, e sua separação faz com que seu código seja melhor compreendido. Baseando-se nisso, é possível concluir que os paradigmas de programação foram criados para satisfazer necessidades de separação que os paradigmas antigos não possuíam ou não satisfaziam: a programação estruturada foi criada para facilitar a interpretação de código de uma linguagem de máquina, e a programação orientada a objetos (POO) foi criada para dar uma forma mais concreta aos códigos da programação estruturada para que tenham um sentido mais próximo do mundo real.

Atualmente, os sistemas possuem muitos interesses que são colocados no mesmo local de

código por serem difíceis de serem modularizados. Interesses como lógica de negócio, persistência e *logging* são, muitas vezes, encontrados no mesmo ponto por causa de uma deficiência do paradigma de programação orientada a objetos: a não-diferenciação de interesses ortogonais.

Interesses ortogonais (também definidos como transversais **(Santos, 2008)** ou entrecortantes **(Junior & Winck, 2004)**) são interesses que, ao mesmo tempo em que tratam de diferentes partes do sistema, são interdependentes entre si. Por exemplo, logging é um interesse separado do sistema cuja função é trazer informações...dos próprios interesses do sistema! Entretanto, a POO não consegue modularizar satisfatoriamente esse tipo de interesse com os interesses do sistema em questão, fazendo com que esses tipos de interesses sejam de difícil manutenção. A Figura 6 mostra no Apache Tomcat a difícil modularização desse tipo de interesse.

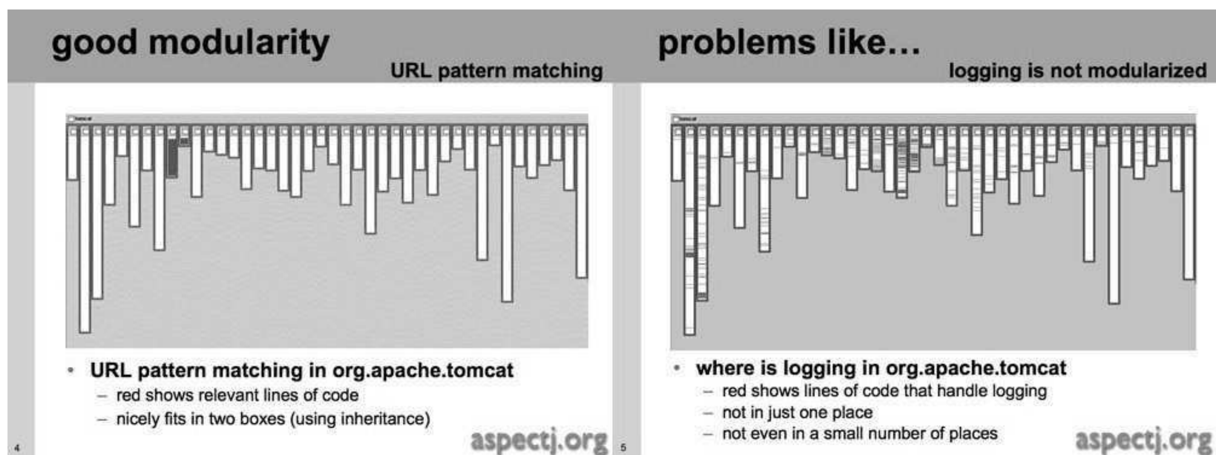

Figura 6. Diferenças entre logging e reconhecimento de URL no Apache Tomcat (11)

Com base nesses fatores, **Santos (2008)** realça uma série de fatores facilmente notados:

- **Fraca coesão:** As classes terão que tratar de funções incoerentes à sua modelagem original, implementando interesses que não eram para ser de responsabilidade das mesmas.
- **Forte acoplamento:** *"As classes passam a ter uma maior dependência externa."*
- **Dificuldade de manutenção:** Muitos códigos semelhantes estarão espalhados pelo sistema. Se for necessária a mudança de um dos interesses, é necessária a mudança em uma grande quantidade de classes.
- **Comprometimento da escalabilidade:** A medida que o software cresce o trabalho de fazer com que funcionalidades novas se "encaixem" com as antigas é muito grande, sendo necessária uma modificação em partes-chave do sistema para que isso aconteça.

- **Aumento de redundância:** Muitos códigos estarão redundantes por causa de sua implementação em inúmeras classes e métodos.

- **Diminuição da reutilização de código:** *"O código será duplicado e não reutilizado."*

Em 1997, Gregor Kiczales e sua equipe apresentaram o artigo "Aspect-Oriented Programming", apresentando as bases do paradigma de programação orientado a aspectos (*Aspect-Oriented Programming*, ou AOP) segundo pesquisas feitas por eles nos laboratórios Xerox PARC, que visa a otimização da separação de interesses proposta em POO **(Kiczales, 1997)**. Segundo **Junior & Winck (2004)**, *"A programação orientada a aspectos possibilita que o programador separe o código referente ao negócio dos interesses ortogonais, centralizando estes interesses em aspectos bem definidos no sistema"*.

2.4.1 COMPOSIÇÃO DO SISTEMA

Um sistema no paradigma de programação orientado a aspectos é composto pelos seguintes componentes:

- **Linguagem de componentes:** A linguagem de componentes é utilizada para implementar as funcionalidades básicas do sistema. GPLs são exemplos de linguagem de componentes.

- **Linguagem de aspectos:** A linguagem de aspectos é uma linguagem auxiliar à linguagem de componentes e deve oferecer todos os benefícios do paradigma em uma sintaxe simples e acessível. Seu principal exemplo é o AspectJ, linguagem de aspectos para Java. É nessa linguagem que serão implementados os interesses ortogonais de um software e também a linguagem utilizada neste trabalho.

- **Combinador de aspectos:** O combinador de aspectos, ou *weaver*, tem a tarefa de entrelaçar o código implementado em uma linguagem de componentes com o código implementado em linguagem de aspectos, gerando o código final. Sua função em AOP é comparável, embora não semelhante, a um *proxy*.

- **Programas escritos:** Serão necessários pelo menos um programa escrito em linguagem de componentes e outro programa escrito em linguagem de aspectos para que a AOP faça o trabalho.

2.4.2 FASES

A Programação Orientada a Aspectos envolve três fases distintas para implementação e montagem de um sistema:

- **Decomposição:** Essa fase envolve a identificação dos interesses e sua classificação como comuns ou ortogonais. **(Santos, 2008)**

- **Implementação:** Após a decomposição dos interesses, os interesses considerados como comuns serão implementados em classes e os interesses considerados ortogonais serão implementados em aspectos. Para os aspectos, são estabelecidas regras de corte, onde cada uma define onde o interesse ortogonal interceptará o comum e será executado. **(Santos, 2008)**

- **Recomposição:** É nessa fase que ocorre o entrelaçamento entre código aspectual e código componente (processo este chamado de *weaving*). A Figura 7 mostra esse processo, que é composto de outras 3 fases: na primeira, o *weaver* gera um grafo do programa componente. Na segunda, o grafo gerado é editado de acordo com as especificações presentes no código aspectual, otimizando-o no processo. Finalmente, na terceira é feita a geração de código com base no grafo construído **(Kiczales 1997)**

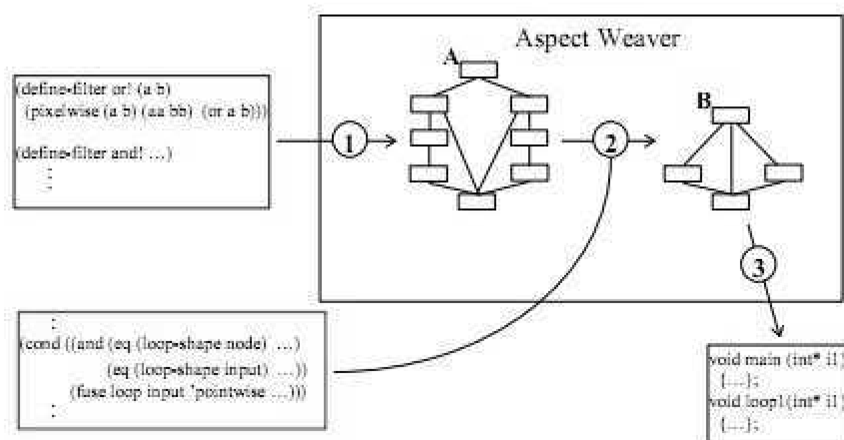

Figura 7. Processo de recomposição de um sistema orientado a aspectos (Kiczales, 1997)

2.4.3 ELEMENTOS

A programação em AOP, assim como outros paradigmas de linguagem, possui elementos próprios para a codificação de seus sistemas:

- **Aspecto:** Um aspecto em AOP é comparável a uma classe em POO: é o principal elemento de encapsulamento deste paradigma. A diferença é que, de acordo com **Kiczales (1997)**, uma classe (ou qualquer outro tipo de componente, como métodos e interfaces) é utilizada para interesses claramente encapsulados em um procedimento geral, enquanto um aspecto é utilizado para interesses que não são claros. Em outras palavras, o aspecto cuidaria dos interesses ortogonais, dando a modularidade necessária para que os componentes cuidem apenas da lógica de negócio do sistema.

  É no aspecto que são declarados os *join points*, *pointcuts* e *advices* necessários para a implementação dos interesses ortogonais de um sistema.

- **Pontos de Junção (Join Points):** Um ponto de junção é um ponto bem definido na execução de um sistema. São como marcações para definir onde os aspectos atuarão, e é possível definir exemplos como chamada e execução de métodos, atribuição de variáveis, execução de exceções e leitura e escrita de dados. Seu funcionamento em AOP é comparável a de um *listener*.

- **Pontos de Atuação (Pointcuts):** Basicamente, o *pointcut* é um conjunto de um ou mais *join points* que é declarado para a interpretação de algo em específico. O objetivo dele é a criação de regras para a execução de determinados trechos do aspecto, através de operadores lógicos usados no conjunto de *join points*. Com um *pointcut* é possível obter vários detalhes e informações a respeito de um *join point* específico, como parâmetros de um método, método ou classe onde o *pointcut* foi encontrado, tipo de exceção, entre outros detalhes e informações.

- **Adendos (Advices):** É o elemento mais importante de AOP. Comparável a um método de uma classe, o *advice* é onde os interesses ortogonais são implementados. Com base no *pointcut* que foi definido, o *advice* controlará se tal interesse será executado antes, durante ou depois dele, fazendo, como mencionado no item 1.4.1, uma espécie de proxy com o sistema implementado na linguagem de componentes utilizada, embora não seja necessariamente um.

Tendo esses elementos em mente, **Santos (2008)** menciona um raciocínio para a implementação de um aspecto: *"Uma forma fácil de montar um raciocínio para se chegar a um adendo completo é baseada em três pequenos passos: o primeiro passo é descrever os pontos de*

atuação em que são definidas as regras de captura dos pontos de junção; segundo é escolher o tipo de adendo para o qual se deseja que ocorra no fluxo de execução (antes, durante, depois); em terceiro é implementar o interesse (fazer o código) que será inserido conforme o segundo passo no local especificado pelo primeiro passo".

2.4.4 APLICAÇÕES

O paradigma AOP, conforme visto nos itens anteriores, visa a separação dos interesses ortogonais da lógica de negócio do sistema, entre eles os mostrados a seguir:

- **Logging:** Já mencionado acima, a função do *logging* é estabelecer um controle e estatística das informações presentes do sistema e do que os usuários dele estão fazendo e utilizando com o intuito de melhorá-lo no futuro. O paradigma é utilizado para fazer do logging um interesse mais organizado e de fácil manutenção.

- **Persistência e controle de transações:** Uma transação em banco de dados é um conjunto de instruções executadas em lote cuja vantagem é a possibilidade de cancelá-la caso alguma instrução dê errado, garantindo segurança e a consistência dos dados de uma aplicação. **Coutinho (2008)** diz que gerenciá-las de forma eficiente e não intrusiva é uma tarefa complicada para o desenvolvedor, ilustrando um exemplo que mostra um método chamando outros dois, que atualizam o banco após seu término. Caso o segundo método falhe, o primeiro já foi executado, e o banco fica inconsistente. Ele também mostra algumas soluções, mas todas acabam provando ser de pouca eficiência ou de pouca elegância. Para solucionar esses problemas, ele propõe o AOP como ferramenta, fazendo-se do uso de *annotations* do Java para mapear de maneira fácil cada método, propondo a melhor solução baseando-se nas peculiaridades de cada *annotation* mapeada.

  Curiosamente, **Santos (2008)** propõe solução semelhante, mostrando diferenças em uma estrutura baseada em *proxy* e em outra baseada em aspectos. Como o aspecto, por si só, já faz a função feita pelo *proxy* sua estrutura acaba simplificada, conforme mostrado nas figuras 8 e 9.

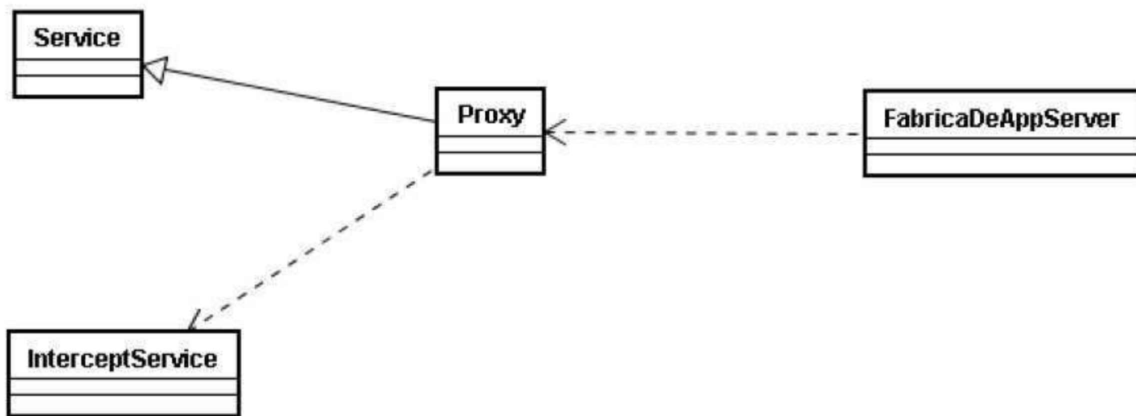

Figura 8. Padrão proxy para controle de transação (Santos, 2008)

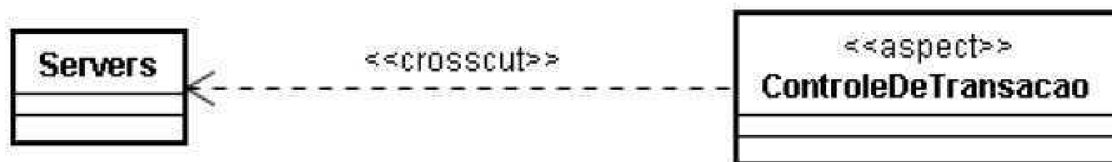

Figura 9. Controle de transação utilizando aspectos (Santos, 2008)

Santos (2008) ainda mostra a mesma diferença com objetos baseados no padrão DAO (*Data Access Object*). Além da simplicidade de se usar AOP, ele identifica problemas na solução, como mais de uma comparação para a execução de um método e menor desempenho devido ao overhead excessivo. As figuras 10 e 11 mostram a diferença de estruturas entre DAO usando *proxy* e DAO usando aspectos.

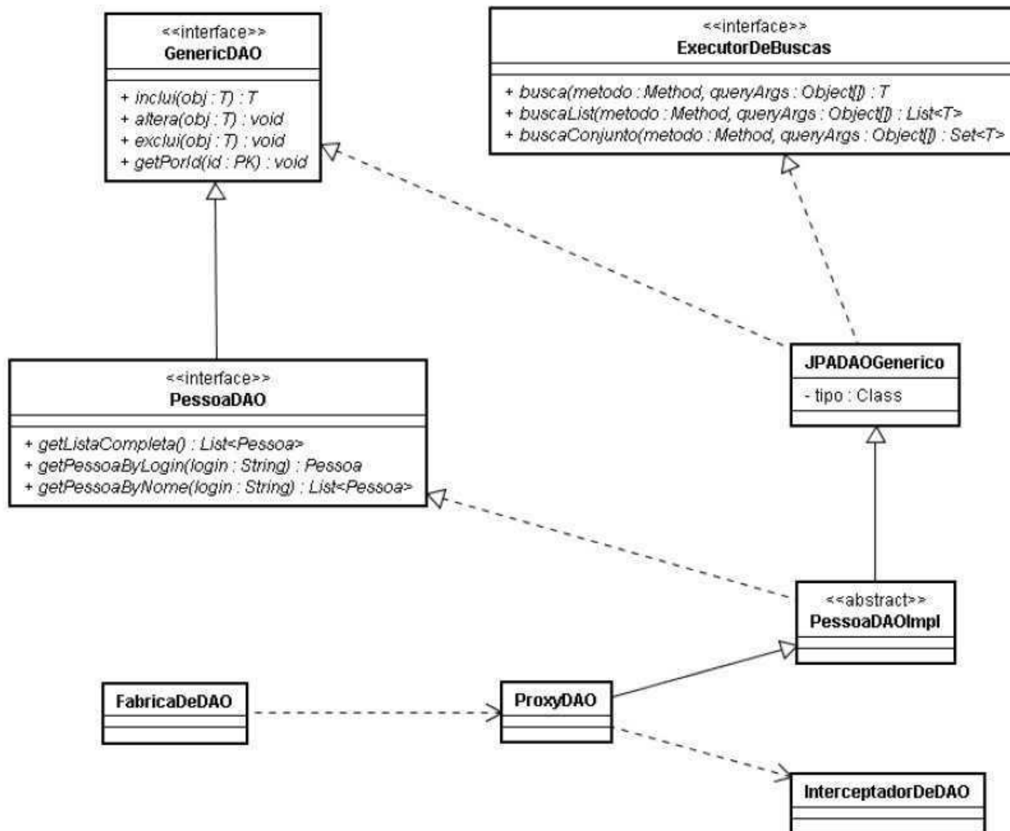

Figura 10. Padrão proxy para padrão DAO (Santos, 2008)

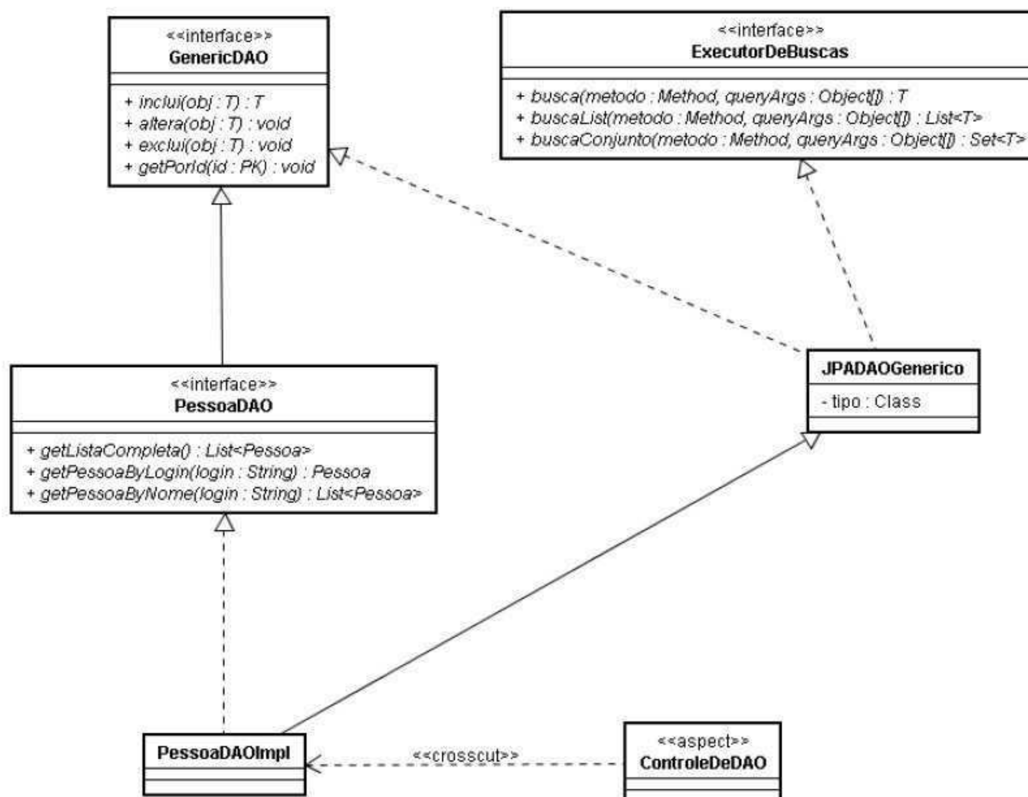

Figura 11. Padrão DAO utilizando aspectos (Santos, 2008)

Entretanto, o uso de AOP é muito mais amplo do que os fatores mencionados, e vem sendo aprofundado ao longo de seus 14 anos de estudos.

2.5 ECLIPSE IDE

O Eclipse é a IDE utilizada durante o desenvolvimento deste trabalho. Inicialmente, o Eclipse era um projeto da IBM até que em Novembro de 2001 foi formado um consórcio para torná-lo um projeto open-source, e em Janeiro de 2004, sua atual responsável, a Eclipse Foundation, foi criada para mantê-lo e atualizá-lo. Atualmente é uma das IDEs mais utilizadas para desenvolvimento de sistemas, principalmente na linguagem Java.

IDE é o acrônimo de Ambiente de Desenvolvimento Integrado (Integrated Development Environment), um tipo de programa que possui diversas ferramentas para auxílio no desenvolvimento de um outro programa. Algumas das principais ferramentas presentes em uma IDE como o Eclipse são:

- **Editor de texto:** O editor de texto de uma IDE é a funcionalidade mais importante do sistema, pois é através dele que os programas são codificados. Em uma IDE como o Eclipse encontramos funcionalidades importantes para esse fim, como auto-completar, procura e substituição de palavras, procura de arquivos e aviso de erros na estrutura de seu código. Tais funcionalidades ajudam a evitar extensas buscas em documentações e erros de compilação ao compilá-lo, que costumam ser a principal causa de tempo gasto e problemas por parte do desenvolvedor.

- **Montagem de um projeto:** A montagem de um projeto pode ser definida como a etapa de toda a sua compilação, linkagem e geração de arquivos necessários para a execução de um projeto criado. Com o uso de uma IDE, é possível a geração de todas essas etapas com o pressionar de um botão, garantindo economia de tempo.

- **Depuração de código:** A depuração de código é uma etapa muito importante no processo de desenvolvimento de um projeto, especialmente em sua manutenção. Através de *Breakpoints*, é possível simular a execução de cada linha de código e analisá-las, sendo possível achar erros de lógica, problemas na utilização de uma biblioteca ou alguma função do sistema ou mesmo novas possibilidades para codificar o mesmo problema.

- **Refatoração:** A refatoração é o processo de modificação de um arquivo ou de sua estrutura com o propósito de otimizar o código ou facilitar sua manutenção. Com a refatoração, é possível, por exemplo, renomear e mover arquivos sem que o sistema possua erros em decorrência desse fato.

Apesar dessas ferramentas serem de bastante ajuda para o desenvolvedor, um dos aspectos mais interessantes do Eclipse é a sua plugabilidade, ou seja, a capacidade de se expandir usando plugins específicos para o programa. Com isso, o programa ganha novas funcionalidades e a capacidade de criar programas em um curto período de tempo é ampliada com as possibilidades que o plugin instalado oferece para a IDE, ao mesmo tempo em que a equipe de desenvolvimento da IDE deixa de se preocupar com a expansão de suas funcionalidades, deixando isso em grande parte para outros bons desenvolvedores ao redor do mundo, e se preocupa mais na estabilidade e no desempenho de seu sistema.

Outro aspecto interessante do Eclipse é o uso de Perspectivas, espaços de trabalho com uma interface propriamente feita para um propósito específico. Estes espaços podem ser customizados de acordo com a necessidade de cada desenvolvedor e podem ser adicionados conforme a instalação de novos plugins ou customização de Perspectivas já existentes. O bom uso de suas Perspectivas garante um grande aumento de produtividade.

O Eclipse se encontra atualmente em sua versão 3.7 (Denominada "Indigo"). Ainda é possível encontrar desenvolvedores usando suas versões anteriores, sendo mais comuns as versões 3.6 (Denominada "Helios"), 3.5 (Denominada "Galileo") e 3.4 (Denominada "Ganymede"). Os plugins utilizados em uma versão também podem sofrer problemas de compatibilidade ao passar para outra (por isso normalmente seu site oficial oferece a compatibilidade das versões ao baixar um plugin).

Segundo o site oficial do projeto, sua comunidade já produziu mais de 200 projetos para estender suas funcionalidades. Estes projetos podem estar em constante atualização ou simplesmente descontinuados, porém estes também podem ser usados caso haja necessidade. Alguns dos principais projetos mantidos pela Eclipse Foundation são os projetos utilizados neste trabalho: O AJDT (AspectJ Development Tools) para o desenvolvimento de uma aplicação Java usando o paradigma de Programação Orientada a Aspectos e o Xtext para o desenvolvimento de *Domain-Specific Languages*. O Xtext possui dentro de seu projeto o Xtend, também utilizado, para o desenvolvimento de geradores de código que acabarão gerando *templates* para a DSL criada.

3 O FRAMEWORK XTEXT

Neste trabalho, os princípios de modelagem citados acima terão como base o *framework* Xtext, que trabalha em conjunto com outra linguagem: O Xtend. Com os dois juntos é possível criar os modelos do sistema e trabalhar com a geração de um código compatível, mas antes é necessário um melhor entendimento de tal ferramenta.

3.1 ECLIPSE MODELLING FRAMEWORK (EMF)

A base para o Xtext é o *Eclipse Modelling Framework* (EMF). O EMF é uma iniciativa da Eclipse Foundation para a realização dos princípios de modelagem. A modelagem do EMF se baseia no *Ecore*, meta-metamodelo próprio do *framework* que surgiu como uma implementação do MOF, mas com o passar do tempo evoluiu e sua modelagem se tornou mais eficiente que a abordagem usada por seu ancestral **(Lucrédio, 2008)**.

A geração desse modelo é relacionada a 4 metaclasses básicas no EMF: `EClass`, `EAttribute`, `EReference` e `EDataType`. Cada uma dessas metaclasses tem, respectivamente, o objetivo de mapear classes, atributos, relacionamentos entre classes e o tipo de atributo de um modelo *Ecore*. Para cada classe do modelo uma interface é definida, contendo um *getter* para cada atributo ou relacionamento da classe. O EMF os reconhecerá pela ajuda de anotações colocadas em cada uma dessas interfaces e *getters*, e produzirá 2 arquivos: um `.ecore` e um `.genmodel`. O `.ecore` é um XML com as informações de modelagem, e o `.genmodel` contém as informações necessárias para gerar um código com base nela **(Quintero & Valderrama, 2007)**. A Figura 12 exemplifica esse processo.

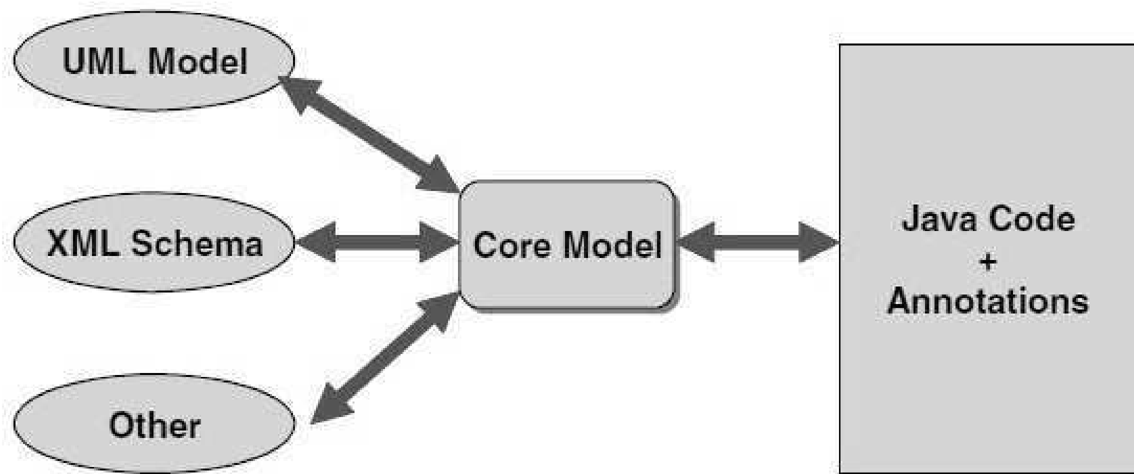

Figura 12. Esquematização básica de transformação e modelagem do EMF (Quintero & Valderrama, 2007)

O Xtext também usa e gera o `.ecore` e o `.genmodel` em seu núcleo. Entretanto, ele possui ferramentas mais específicas para modelagem e geração de código, permitindo um maior controle de seus artefatos.

3.2 XTEXT

Xtext (2011) explica que o Xtext é um *framework* para a construção de linguagens, sejam DSLs ou GPLs. Como exemplo, ela cita o fato de existirem linguagens que não possuem um suporte sólido com as ferramentas que possuem atualmente e podem usar o Xtext para gerar essa linguagem e ter toda a robustez que o Eclipse oferece como IDE. Neste trabalho, foi utilizada a versão 2.0 do Xtext, que apresenta algumas diferenças entre suas versões anteriores.

O Xtext é um projeto integrado ao EMF, usando o `.ecore` e o `.genmodel` para gerar a linguagem especificada. De acordo com **(17)**, além do *Ecore* o Xtext criará um *parser*, um *serializer* e um editor no Eclipse para o teste de sua nova linguagem ao fazer a montagem do projeto. Para isso, ele providencia suas próprias DSLs e APIs para a construção da mesma.

O Xtext gera os arquivos necessários para a execução da linguagem com base em um arquivo `.xtext`. Este arquivo possui todas as informações de gramática que a linguagem precisa para gerar um código sintaticamente válido. **(19)** explica um pouco a geração da linguagem do Xtext, justificando o uso da norma EBNF como base para formular a gramática da mesma e a construção da árvore de derivação. Para complementar a ideia acima, **Xtext (2011)** possui uma breve menção de que a gramática definida terá que ser do tipo LL(*), gerando um *parser* ANTLR

que é por si só descendente recursivo.

A Figura 13 mostra um modelo de um arquivo `.xtext`, com gramática já implementada e testada. Nela é possível ver as palavras reservadas `grammar`, `with` e `generate`, necessárias para iniciar todo arquivo dessa extensão. Após definir estas configurações, os não-terminais e as regras de produção são definidos. Palavras reservadas são definidas entre aspas, regras semânticas são definidas como atribuições e outros não-terminais são definidos apenas com seus nomes já mencionados no arquivo.

```
grammar org.xtext.example.Entity with org.eclipse.xtext.common.Terminals

generate entity "http://www.xtext.org/example/Entity"

Model:
  (types+=Type)*;

Type:
  TypeDef | Entity;

TypeDef:
  "typedef" name=ID ("mapsto" mappedType=JAVAID)?;

JAVAID:
  name=ID("." ID)*;

Entity:
  "entity" name=ID ("extends" superEntity=[Entity])?
  "{"
    (attributes+=Attribute)*
  "}";

Attribute:
  type=[Type] (many?="*")? name=ID;
```

Figura 13. Gramática válida pelo Xtext (17)

3.2.1 MODELING WORKFLOW ENGINE (MWE)

No Xtext, apesar de o arquivo `.xtext` ser a base de todas as linguagens geradas, ele não é apenas o único. O arquivo define terminais, não-terminais e regras de produção, mas não especifica exatamente qual será o não-terminal inicial da gramática. Para isso, o Xtext utiliza o *Modeling Workflow Engine* (MWE) para mapeá-lo.

O MWE se baseia um arquivo XML para leitura dos dados e definição dos componentes utilizados **(18)**. Entretanto, a versão 2.0 do Xtext utiliza uma nova versão do MWE: o MWE2, que possui uma sintaxe mais próxima a uma linguagem de programação. **(20)** fornece detalhes sobre a codificação deste arquivo, mas explicá-las aqui foge inteiramente do foco deste trabalho. Seu uso será melhor explicado no capítulo 5, onde o arquivo do MWE2 será utilizado para executar o ambiente onde será executada a linguagem.

Apesar de o Xtext e o MWE2 serem necessários para criação e mapeamento da linguagem de programação, nenhum dos dois é responsável pelo que essa linguagem vai gerar . Para isso, o Xtext é integrado com outra linguagem: O Xtend.

3.3 XTEND

Segundo **Xtext (2011)**, o Xtend é uma linguagem de programação que se integra a *Java Virtual Machine* (JVM), tem suas raízes na linguagem Java, aprimorando alguns conceitos como tipagem fraca, expressões de *template* e sobrecarga de operadores. O Xtend tem suas origens no Xpand, outra linguagem com o mesmo propósito, entretanto oferece mais opções para manipular e gerar arquivos.

Apesar de ser originária do Xpand, a forma de codificar no Xtend é muito diferente. Enquanto no Xpand os arquivos e templates gerados são definidos em arquivos `.xpt` com uma sintaxe própria para tal (Figura 14), no Xtend os mesmos são definidos em arquivos `.xtend` com uma estrutura peculiar. A geração de arquivos é feita dentro do método `doGenerate`, gerado pelo no Eclipse através do próprio plugin do Xtext ao criar um projeto do *framework*, que contém um parâmetro que possui encapsulado um método chamado `generateFile`, e que possui como parâmetros o nome do arquivo e o template que será gerado dentro dele (Figura 15). A geração de código é feita através dos elementos da gramática definidos no Xtext, e é possível gerá-lo de duas formas diferentes, definidas através da palavra `def`. A primeira forma remete a forma original do Xpand, utilizando sintaxe equivalente. A segunda forma remete a uma concatenação normal de cadeias de caracteres, utilizando a sintaxe do próprio Xtend para concatená-las e retornar um template válido (Figura 16).

```
«IMPORT entity»

«DEFINE dao FOR entity::Entity»
  «FILE this.name + "DAO.java"»
    import java.util.Collection;
    import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
    public class «this.name»DAO
      extends HibernateDaoSupport {
      «EXPAND crud FOR this»
    }
  «ENDFILE»
«ENDDEFINE»

«DEFINE crud FOR Entity»
  public «this.name» load(Long id) {
    return («this.name»)getHibernateTemplate().get(«this.name».class, id);
  }

  @SuppressWarnings("unchecked")
  public Collection<«this.name»> loadAll() {
    return getHibernateTemplate().loadAll(«this.name».class);
  }

  public «this.name» create(«this.name» entity) {
    return («this.name») getHibernateTemplate().save(entity);
  }

  public void update(«this.name» entity) {
    getHibernateTemplate().update(entity);
  }

  public void remove(«this.name» entity) {
    getHibernateTemplate().delete(entity);
  }
«ENDDEFINE»
```

Figura 14. Exemplo de um arquivo `.xpt` para o Xpand. (18)

```
override void doGenerate(Resource resource, IFileSystemAccess fsa) {
    //TODO implment me
    for(e : resource.allContentsIterable.filter(typeof(Entity))){
        var fileNames = e.fullyQualifiedName.segments
        var entityFileName = ""
        var aspectFileName = ""
        var controllerFileName = ""
        var i = 1
        for(fileName : fileNames){
            if(i < fileNames.size()){
                entityFileName = entityFileName + fileName + "/"
                aspectFileName = aspectFileName + fileName + "/"
                controllerFileName = controllerFileName + fileName + "/"
            }else{
                entityFileName = entityFileName + fileName + ".java"
                aspectFileName = aspectFileName + "aspect/Aspecto" + fileName.toFirstUpper + ".aj"
                controllerFileName = controllerFileName + "control/" + fileName.toFirstUpper + "Controller.java"
            }
            i = i + 1
        }

        fsa.generateFile(entityFileName,e.compile)
        fsa.generateFile(aspectFileName,e.aspect)
        fsa.generateFile(controllerFileName,e.control)
    }

    var generateMain = true
    if(generateMain){
        fsa.generateFile("app/init/AppInit.java",generateMain.app)
    }
}
```

Figura 15. Exemplo de geração de arquivos no Xtend.

```
def functionHeader(Function f)'''«IF f.type == null»void«ELSE»«typeDeclaration(f)»«ENDIF» «f.name»(«parameters(f)»)'''

def typeDeclaration(Attribute a)'''«IF a.many»List<«a.type.name»>«ELSE»«a.type.name»«ENDIF»'''

def typeDeclaration(Function f)'''«IF f.many»List<«f.type.name»>«ELSE»«f.type.name»«ENDIF»'''

def variable(Attribute a)'''«typeDeclaration(a)» «a.name»'''

def aspectParameters(Function f){
    var pars = ""
    var i = 0
    for(p:f.parameters){
        pars = pars + typeDeclaration(p).toString
        i = i + 1
        if(i < f.parameters.size){
            pars = pars + ", "
        }
    }

    pars
}

def parameters(Function f){
    var pars = ""
    var i = 0
    for(p:f.parameters){
        pars = pars + variable(p).toString
        i = i + 1
        if(i < f.parameters.size){
            pars = pars + ", "
        }
    }

    pars
}
```

Figura 16. Exemplos para definir templates no Xtend.

4 METODOLOGIA

Nos capítulos 2 e 3, foram estudados os conceitos a serem utilizados, principalmente a respeito de MDD e de MDA, discutindo suas motivações e seu embasamento. Também foi discutido conceitos a respeito do *framework* Xtext e seu conjunto de ferramentas, propiciando uma alternativa completa e de grande produtividade para a construção de novas linguagens. Neste capítulo, será discutida uma proposta para a seguinte questão: É possível criar uma linguagem que transforme os conceitos mencionados em uma alternativa para o desenvolvimento de sistemas atual?

De acordo com um de seus criadores, Sven Efftinge, o uso do Xtext é uma peça importante na abordagem MDSD, enfatizando que a mesma consiste no uso de DSLs, interpretadores e geradores de código para tornar uma abstração de domínio específico executável **(21)**. Para isso, é necessário uma metodologia que condense os procedimentos para formular uma linguagem no Xtext e os procedimentos de um Engenheiro de Software para o desenvolvimento de um novo sistema. Pensando nestas considerações, foi formulada uma série de etapas para serem utilizadas como base de trabalho, como mostra a Figura 17.

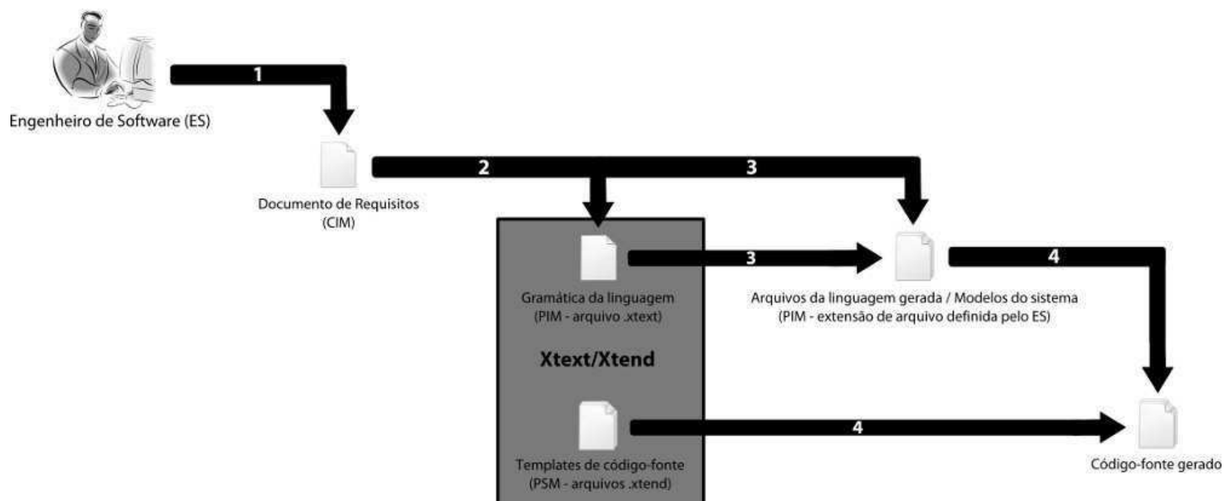

Figura 17. Etapas da abordagem MDSD utilizando o framework Xtext.

A etapa 1 se refere a especificação dos requisitos. Esta etapa já é conhecida pelo Engenheiro de Software, pois é a mesma etapa inicial que acontece no desenvolvimento de outros sistemas que não utilizariam esta abordagem. Ele, através de levantamentos, visitas e entrevistas com o cliente, verifica uma série de requisitos que julga serem necessários, e redige um documento contendo toda a especificação desses requisitos. O Documento de Requisitos é considerado na abordagem

MDA/MDSD como o CIM, e indica o que é necessário para a elaboração das próximas etapas.

A etapa 2 diz respeito a formulação da gramática e dos *templates*. Nesta etapa o Xtext será utilizado para geração da linguagem e o Xtend para geração de *templates* para código-fonte, conforme explicado no capítulo anterior. Com o Documento de Requisitos completo, o Engenheiro de Software analisará todos os recursos que deverão ser necessários para tais gerações, implementando a gramática da linguagem e os *templates* no processo. Como a gramática para geração da linguagem criada funciona como um modelo para gerar todos os recursos necessários, o arquivo `.xtext` pode ser classificado como um dos PIMs desta abordagem. Os *templates* funcionarão como PSMs, pois são eles que vão conter as informações para transformação do código-fonte.

A etapa 3 é o desenvolvimento da modelagem do sistema. Com base na linguagem criada através da gramática gerada, a modelagem é formulada. Estes modelos dependerão das informações presentes no Documento de Requisitos com maior intensidade do que na etapa anterior, pois ela está conectada ao sistema de uma maneira mais próxima. Os modelos gerados nesta etapa também são considerados PIMs e serão arquivos da linguagem gerada, que possui uma extensão própria definida pelo Engenheiro de Software na etapa anterior, pedida pelo Xtext ao criar um projeto no Eclipse.

A etapa 4 é a geração de código do sistema. Após a geração de modelos do sistema, é feita a montagem do projeto no Eclipse para gerar o código-fonte. Com a modelagem agindo como PIMs e os *templates* gerados através do Xtend agindo como PSMs, tem-se o conjunto de regras de transformação necessárias para a geração de um código pronto para execução ou para desenvolvimento de regras de negócio mais específicas.

É importante observar que, apesar de seguir procedimentos lógicos e bem definidos, esta abordagem não precisa ser necessariamente linear. Em outras palavras, é possível pular etapas e voltar conforme a necessidade do desenvolvimento. Por exemplo, no capítulo anterior há a informação de que o Xtext cria um editor para o uso de sua linguagem. Com base nela, o Engenheiro de Software pode, ao invés de começar o desenvolvimento dos *templates* para geração do código-fonte após o desenvolvimento da gramática, abrir esse editor para testar a sua linguagem e verificar se ela não apresenta problemas ou então perceber melhorias e após isso desenvolvê-los, uma vez que os arquivos do Xtend só terão utilidade real na etapa 4. Tal procedimento também pode ser notado na documentação oficial do *framework* **(Xtext, 2011)**, onde a linguagem é gerada e testada primeiro para depois gerar os *templates* necessários.

A utilização do Xtext para o MDSD evidencia as vantagens e desvantagens da abordagem, e acontece devido a uma necessidade constante de melhoria e incremento dos modelos, linguagem e *templates* utilizados. Como exemplo, supondo que a empresa que pediu um projeto utilizando essa metodologia decida mudar a infraestrutura completamente, implicando também em uma migração completa de código de uma linguagem para outra (Ex: Java para C#), isto significa que, embora os requisitos sejam os mesmos, será necessária uma marcação no modelo identificando a linguagem, implicando em uma mudança na própria linguagem que é utilizada para gerar os modelos, e também mudanças nos *templates* para possibilitar a geração do código-fonte necessário se eles não estiverem preparados para isso. Ao mesmo tempo que vantagens como reúso de código, independência de plataforma e portabilidade estejam evidenciadas nesse exemplo, as desvantagens também podem: Vale lembrar que atualmente há uma alta rotatividade entre funcionários em empresas de TI devido a falta de mão de obra qualificada, e pode acontecer o êxodo dos que detinham o conhecimento sobre a ferramenta. Por haver uma baixa difusão destes métodos, poderá ocorrer a contratação de novos funcionários com pouco ou nenhum conhecimento para usá-la, resultando em gastos com treinamentos e aprendizado.

Após a definição de uma metodologia coerente para desenvolver sistemas em conjunto com o Xtext, é necessário verificar se tal abordagem funciona na prática. Para isso, foi elaborado um estudo de caso, visto no capítulo 5, para verificar sua real eficiência.

5 ESTUDO DE CASO

Neste capítulo, será descrito um estudo de caso elaborado com base na metodologia descrita no capítulo 4, especificando o desenvolvimento de suas etapas e o comportamento do Engenheiro de Software em cada uma delas. Para este trabalho foi desenvolvida apenas uma aplicação simples de cadastro, cujos procedimentos são detalhados nos itens abaixo.

5.1 ESPECIFICAÇÃO DOS REQUISITOS

Para a Especificação dos Requisitos, foi desenvolvido um Documento de Requisitos contendo a especificação de um sistema para uma empresa de eventos fictícia, referente a um sorteio de prêmios em um evento conhecido. Tal documento encontra-se no Apêndice A e informa que estarão envolvidos no sistema funcionalidades de cadastro, sorteio e a exibição de um menu de acesso ao iniciar e após a conclusão de uma delas. O documento descreve também o log de informações assim que uma funcionalidade é acessada, justificando o uso de AOP para o sistema.

O Documento de Requisitos elaborado descreve o que é preciso para o Engenheiro de Software desenvolver a modelagem do sistema. Entretanto, para modelar e desenvolver um sistema baseando-se na metodologia especificada neste trabalho, é necessário primeiro criar a linguagem que será usada como modelagem e os geradores (*templates*) que irão conter as informações necessárias para a geração de código.

5.2 DESENVOLVIMENTO DA GRAMÁTICA E GERADORES

A gramática gerada contém elementos da linguagem gerada por Peter Friese **(17)** e elementos presentes na documentação do Xtext **(Xtext, 2011)**, mesclando-os para obter um maior poder da modelagem e da geração de código. O arquivo final, chamado `Tccdsl.xtext`, encontra-se presente no Apêndice B.

A linguagem desenvolvida tem a seguinte lista de palavras reservadas, cada uma colocada para definir algum elemento da modelagem ou algum mapeamento da mesma:

- `typedef`: Utilizada para definir um tipo de atributo que será usado na modelagem.

- `mapsto`: Utilizada após um `typedef` para mapear o tipo de atributo no momento da geração de código.

- `package`: Utilizada para definir um pacote de classes do sistema. Nela, podem ser definidos definições de tipos (`typedef`), atributos, novos pacotes e entidades.

- `entity`: Utilizada para definir uma entidade do sistema. Nela, podem ser definidos definições de tipos (`typedef`), atributos, novos pacotes e entidades.

- `extends`: Utilizada para atribuir herança a uma entidade.

- `functions`: Utilizada para atribuir funções a uma entidade. Podem ser definidas com os tipos já definidos ou como `void`, caso não retorne valor.

- `void`: Utilizada para definir uma função que não retornará valor (procedimento).

Após a definição da gramática e antes da definição dos geradores, é preciso que o Xtext gere toda a estrutura para que a linguagem funcione completamente. Para isso, o framework usa um arquivo auxiliar de extensão `.mwe2` que é gerado assim que o projeto é criado no Eclipse. O arquivo gerado, chamado `GenerateTccdsl.mwe2` e encontrado no Apêndice B, contém as informações necessárias para que o Xtext gere o *parser*, o *serializer* e o *Ecore* necessários para o uso desta linguagem. Após a utilização deste arquivo, um editor no Eclipse será aberto e para testar a linguagem basta apenas criar um novo projeto Java e criar um arquivo com a extensão definida no projeto (neste caso, a extensão `.tcc`).

Entretanto, a garantia de uma linguagem já estruturada não é a garantia de que já esteja gerando código. Em sua estrutura criada, o Xtext cria algumas classes que mapeiam os elementos presentes na gramática criada, que podem ser utilizadas nos arquivos do Xtend e com base nas mesmas é possível mapear os pontos necessários da modelagem criada e gerar código com esses pontos mapeados.

O arquivo `TccdslGenerator.xtend`, presente no Apêndice B, também é gerado ao criar um projeto no Eclipse e é ele que contém o método `doGenerate`. Para programar templates no Xtend, a documentação do Xtext recomenda que seja utilizada a interface `IQualifiedNameProvider` através da anotação `@Inject` para injetar uma classe/interface estender as funcionalidades presentes para a geração de código. Além disso, com essa anotação é possível simular heranças múltiplas,

"anotando" várias classes no processo e permitindo definir uma organização maior ao código, dividindo os *templates* em classes diferentes dependendo do interesse: Enquanto o arquivo `TccdslGenerator.xtend` é definido com o interesse principal de ser o gerador de arquivos, os outros arquivos estenderiam este gerador com definições de código, tornando-o poderoso e de fácil manutenção.

Os arquivos definidos como geradores de código, também presentes no Apêndice B, foram nomeados como `ModelExtensions.xtend`, `AspectExtensions.xtend`, `ModelExtensions.xtend` e `TccdslExtensions.xtend`. Este último arquivo tem como objetivo gerar as partes comuns de código entre os arquivos, como nomes de pacote, declaração de variáveis e parâmetros e *headers* de funções. Os outros três arquivos herdam este e tem o interesse respectivo de gerar as classes dos modelos, aspectos e classes dos controladores (que executarão as funcionalidades) do sistema.

A geração de código feita pelos geradores terá a seguinte arquitetura: É definida uma classe principal, chamada de `AppInit`, aspectos, classes de modelos e classes de controladores. No `AppInit` irão conter todas as chamadas de método vindas das classes de modelos. Estes, por sua vez, chamarão os métodos estáticos de mesmo nome vindos das classes de controladores, e os aspectos interceptarão estas chamadas e farão suas operações. Esta arquitetura não é configurável de acordo com as marcações feitas na linguagem pois demandaria um tempo maior de estudo e refinamento.

5.3 MODELAGEM DO SISTEMA

A modelagem do sistema é feita com base na linguagem elaborada no Xtext, e está no arquivo `dsl.tcc`, presente no Apêndice B. Para esta aplicação, é necessário apenas a modelagem de uma única entidade: a entidade `Pessoa`, que conterá dois atributos: `nome` e `idade`, e duas operações: `cadastrar` e `sortear`. A operação `cadastrar` é definida como um procedimento e a operação `sortear` retornará um objeto do mesmo tipo da entidade. As entidades estarão presentes no pacote `tcc.model`, e com ela também serão gerados os pacotes `tcc.model.aspect` para aspectos, `tcc.model.control` para controladores e `app.init` para a classe principal.

Um item interessante a se considerar referente a modelagem é que a linguagem deste trabalho acabou com uma forma semelhante a abordada pelo Diagrama de Classes presente na UML. Apesar da diferença de que a linguagem fica em um arquivo texto e o Diagrama é baseado em uma ilustração, as duas se tornam comparáveis pois ambas levam aos mesmos objetivos e,

portanto, geram código semelhante com a modelagem.

5.4 GERAÇÃO DE CÓDIGO

Após a etapa de modelagem, para o Eclipse usar a estrutura fornecida pelo Xtext para a geração de código é preciso a criação de uma pasta de código-fonte chamada *src-gen* e a montagem do projeto onde se localiza a linguagem **(Xtext, 2011)**. A estrutura de códigos, presentes no Apêndice B, é gerada nessa pasta criada e pronta para ser colocada em um projeto do AspectJ.

A princípio, é possível notar erros no código gerado (Figura 19). Isso ocorre pois o que é gerado não é o código pronto a princípio, e sim uma "casca" que está pronta para ser codificada pelos programadores. Também há o fato de que o projeto utilizado para codificar a linguagem gerada no Xtext é um projeto Java, não tendo as informações e bibliotecas necessárias para montar um projeto em AspectJ, por isso a necessidade de criar um outro projeto.

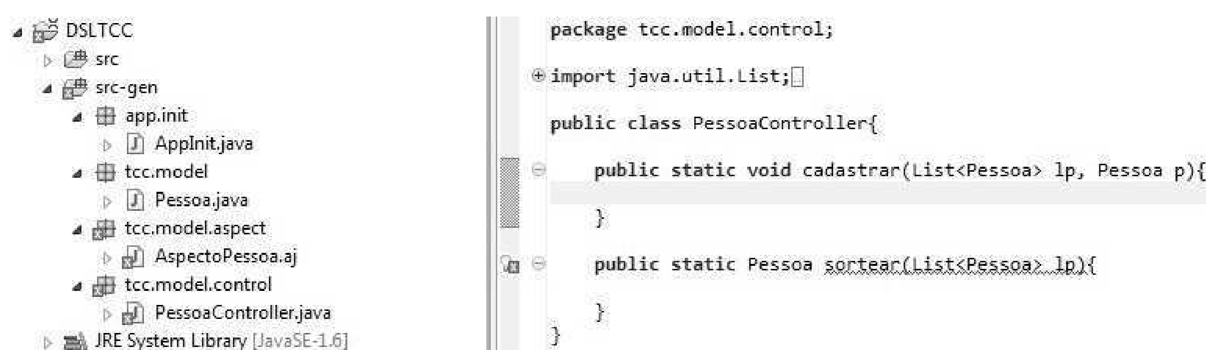

Figura 18. "Erros" de geração de código

5.5 RESULTADOS E CONSIDERAÇÕES

Após o código ser gerado e a aplicação completada, foi possível verificar se o MDSD foi realmente eficiente no processo. Para isso, foi realizada uma contagem das linhas presentes nos arquivos gerados e dos arquivos finais, e logo após feita uma comparação entre os mesmos. Para critério do que foi considerado como "linha de código", foram desconsideradas na contagem linhas com espaço em branco. Os resultados estão presentes na Tabela 1, onde as linhas geradas foram colocadas como "linhas úteis geradas" para atender melhor ao critério utilizado (O Xtext também gerará linhas em branco se elas forem colocadas em suas definições de *templates*). Os códigos dos arquivos modificados na implementação (`AppInit.java` e `PessoaController.java`) estão

presentes no Apêndice C.

Tabela 1. Comparação entre código escrito e código gerado por MDSD

Classe	Total de linhas de código	Linhas úteis geradas	Porcentagem de linhas geradas (%)
AppInit	72	5	6,94
Pessoa	28	28	100,00
PessoaController	29	11	37,93
Aspecto	**Total de linhas de código**	**Linhas úteis geradas**	**Porcentagem de linhas geradas (%)**
AspectoPessoa	29	29	100,00
TOTAL	158	73	46,20

Na tabela, é possível notar uma enorme diferença entre a quantidade de linhas que é gerada pela linguagem e a quantidade de linhas digitada pelo desenvolvedor, comprovando o ganho de produtividade que a geração de código pela linguagem criada proporcionou. Os códigos das classes de modelo e dos aspectos foram totalmente gerados, fazendo com que o desenvolvedor tenha mais foco no desenvolvimento da aplicação e das funcionalidades a serem implementadas.

Para ressaltar os benefícios do MDSD, também foi possível comparar estes resultados com a geração de código realizada pelo próprio Eclipse. Neste caso, foi realizada a criação de um novo projeto com uma estrutura equivalente a utilizada e as ferramentas do Eclipse para geração de arquivos, pacotes e código -fonte, e realizada a contagem de linhas pelo mesmo método para obtenção dos valores da tabela anterior. Seus resultados podem ser observados na Tabela 2.

Tabela 2. Comparação entre código escrito e código gerado pelo Eclipse

Classe	Total de linhas de código	Linhas úteis geradas	Porcentagem de linhas geradas (%)
AppInit	72	5	6,94
Pessoa	28	17	60,71
PessoaController	29	3	10,34
Aspecto	**Total de linhas de código**	**Linhas úteis geradas**	**Porcentagem de linhas geradas (%)**
AspectoPessoa	29	3	10,34
TOTAL	158	28	17,72

Comparando os dados de ambas das tabelas, a geração de código foi diminuída em quase 30% quando feita com o Eclipse. Há outros fatores importantes a serem analisados, como o fato de classes totalmente geradas na abordagem desse trabalho não serem geradas totalmente caso apenas

o Eclipse seja usado, evidenciando o alcance limitado da IDE para esse tipo de atividade. Entretanto, o Eclipse não é desenvolvido sob esse fim, e sua geração de código serve mais como auxílio ao desenvolvedor do que uma solução automatizada. Para isso, há o desenvolvimento de plugins por terceiros para complementá-lo e torná-lo apto a outras abordagens.

Contudo, apesar do aumento de produtividade o MDSD é uma opção de médio/longo prazo para desenvolvimento de sistemas. Antes de desenvolver um sistema com esta abordagem é preciso uma DSL bem estruturada, e seu desenvolvimento necessita de um constante aperfeiçoamento devido a quantidade enorme de casos e situações em que uma linguagem é utilizada para desenvolvimento. A quantidade linhas de código necessárias para o desenvolvimento da DSL deste trabalho estão na Tabela 3.

Tabela 3. Linhas de código implementadas para a geração de código

Arquivo Xtext	**Linhas criadas**
Tccdsl.xtext	37
Classe Xtend	**Linhas criadas**
TccdslGenerator	65
TccdslExtensions	73
ModelExtensions	48
AspectExtensions	39
ControllerExtensions	32
Arquivo Linguagem	**Linhas criadas**
dsl.tcc	21
TOTAL	315

Para gerar 46,2% de um código para uma aplicação simples, foi implementada uma solução com quase o dobro de linhas de código da mesma, o que mostra a inviabilidade do MDSD em projetos de pequeno porte e a curto prazo, sendo mais eficiente uma utilização contínua em projetos com equipes grandes. Para o sucesso da abordagem, é necessário um aperfeiçoamento contínuo, para que as ferramentas desenvolvidas sejam difundidas e tenham uma maior vida útil. Com o tempo e o uso constante, o tempo ganho no desenvolvimento de sistemas passará o tempo gasto no desenvolvimento da ferramenta.

6 CONCLUSÃO

Com estes estudos, podemos concluir que o Xtext é um *framework* viável para o desenvolvimento de um sistema baseado em modelos. Suas ferramentas permitem um controle muito maior da DSL a ser criada para tal feito e da geração de código final, e cada passo da geração de uma linguagem remete a um modelo específico da abordagem do MDSD. Sendo assim, a geração de uma linguagem de modelagem baseada no *framework* traz uma boa estruturação e uma manutenção facilitada graças a elas. O Xtext combinado com o Eclipse provém um editor de texto poderoso que permite a fácil identificação de erros léxicos/sintáticos/semânticos e uma construção rápida da modelagem baseada na DSL.

O uso de AOP na aplicação permitiu um controle poderoso dos interesses ortogonais do sistema, evitando que os mesmos se entrelacem com os reais interesses e permitindo soluções de escopo mais generalizado que abrangem o sistema todo em um único trecho e permitem ao desenvolvedor ganhar tanto tempo de desenvolvimento quanto facilidade na manutenção.

Abordagens do MDD como MDA e MDSD permitem ao Engenheiro de Software a geração de vários artefatos do sistema apenas com a modelagem do mesmo, permitindo que o desenvolvedor se concentre apenas com o que interessa. Com o conhecimento necessário, é possível criar um código em apenas algumas horas, ao invés de dias com a digitação manual e verificação de erros, ocasionando em um ganho notável de produtividade.

Entretanto, o uso do Xtext no MDD em um projeto de pequeno porte se mostra inviável, pois o trabalho e tempo para desenvolvimento de uma linguagem própria para isso é, em geral, maior do que o das aplicações que a utilizam. Tal *framework* e tal abordagem também não são tão difundidas no mercado de trabalho, o que significa que a empresa que adotar estas práticas deve estar preparada para fornecer treinamento para novos funcionários, resultando em novas despesas.

Também é possível notar o fato de que os geradores da linguagem nem sempre estarão prontos para todas as possibilidades em sua primeira versão. Seu uso só terá retorno após uma certa bagagem de projetos, e, por isso, a criação de novas metodologias para o uso do Xtext, visando o constante aperfeiçoamento da linguagem criada e de sua finalidade, se mostra válida.

Como o uso de tais práticas exige constante manutenção e aprimoramento, para estudos futuros é sugerida a criação de novas metodologias para o uso do *framework* e novas marcações na linguagem, como marcações para determinar a linguagem utilizada, os *frameworks* utilizados, o uso de *Design Patterns* e a opcionalidade de Aspectos na aplicação. Seria interessante também, embora não tão primordial quanto a sugestão anterior, marcações para determinação de relacionamentos entre classes e personalização da arquitetura do sistema, ao invés de trabalhar com uma já pré-determinada.

7 REFERÊNCIAS

(1) LUCREDIO, Daniel, Uma abordagem orientada a modelos para reutilização de software, Junho/2009, págs. 19-30, 34-50, presente em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-02092009-140533/pt-br.php>

(2) QUINTERO, A. M. Reina, VALDERRAMA, J. Torres, Using Aspect-orientation Techniques to Improve Reuse of Metamodels, Electronic Notes in Theoretical Computer Science 163 (2007) 29–43, Department of Languages and Computer Systems - University of Seville - Seville, Spain, págs. 29-31, presente em: <http://www.sciencedirect.com/science/article/pii/S1571066107001466>

(3) ISSA, Lucas V. N., Desenvolvimento de Interface com Usuário Dirigido por Modelos e Geração Automática de Código, págs. 3,7-29, presente em: <http://www.bibliotecadigital.ufmg.br/dspace/handle/1843/SLBS-6XYFNZ>

(4) BELIX, José E., Um estudo sobre MDA: suporte fornecido pela UML e reuso de soluções pré-definidas, págs. 25-44, presente em: <http://www.teses.usp.br/teses/disponiveis/3/3141/tde-11052006-165548/pt-br.php>

(5) Xtext Documentation, págs. 37-38, 55, presente em <http://www.eclipse.org/Xtext/documentation/>

(6) SQL, presente em: <http://pt.wikipedia.org/wiki/SQL> Acesso em: 24 out. 2011

(7) Ambiente de desenvolvimento integrado, presente em: <http://pt.wikipedia.org/wiki/Ambiente_de_Desenvolvimento_Integrado> Acesso em: 11 out. 2011

(8) Eclipse (software), presente em: <http://en.wikipedia.org/wiki/Eclipse_%28IDE%29> Acesso em: 11 out. 2011

(9) Eclipse – official site, presente em: <http://www.eclipse.org> Acesso em: 11 out. 2011

(10) Microsoft SQL Server 2008: Programmability, presente em: <http://www.microsoft.com/thailand/sqlserver/2008/programmability.aspx> Acesso em: 24 out.

2011

(11) AOP & The DataServices Project, presente em: <http://kosmaczewski.net/2007/03/27/aop-the-dataservices-project/> Acesso em: 26 out. 2011

(12) SANTOS, Heliomar K. da R., Programação Orientada a Aspectos, presente em: <http://code.google.com/p/aspectos/downloads/list>

(13)WINCK, Diogo, JUNIOR, Vicente G. De S., Programação Orientada a Aspectos Abordando Java e aspectJ, presente em: <http://inf.unisul.br/~ines/workcomp/cd/autores_d.html>

(14) KICZALES, Gregor, Aspect-Oriented Programming, presente em: <http://www2.parc.com/csl/groups/sda/publications/papers/Kiczales-ECOOP97/>

(15) COUTINHO, Paulo C., Hibernate e AspectJ - Gerenciando transações de forma 100% transparente, presente em: <http://www.cesar.org.br/site/hibernate-e-aspectj-gerenciando-transacoes-de-forma-100-transparente/>

(16) KICZALES, Gregor, HANNEMAN, Jan, Design Pattern Implementation in Java and AspectJ, presente em: <http://www.cs.ubc.ca/labs/spl/papers/2002/oopsla02-patterns.pdf>

(17) FRIESE, Peter, Getting started with Xtext, presente em: <http://www.peterfriese.de/getting-started-with-xtext/>. Acesso em: 08 nov. 2011

(18) FRIESE, Peter, Getting started with Xtext, part 2, presente em: <http://www.peterfriese.de/getting-started-with-xtext-part-2/>. Acesso em: 08 nov. 2011

(19) EFFTINGE, Sven, Parsing Expressions with Xtext, presente em: <http://blog.efftinge.de/2010/08/parsing-expressions-with-xtext.html >. Acesso em: 08 nov. 2011

(20) MWE2 in depth, presente em: <http://www.eclipse.org/Xtext/documentation/2_1_0/118-mwe-in-depth.php>. Acesso em 09 nov. 2011

(21) EFFTINGE, Sven, MDSD and DSLs, presente em: <http://blog.efftinge.de/2008/07/mdsd-and-dsls.html>. Acesso em: 14 nov. 2011

8 APÊNDICES

APÊNDICE A: REQUISITOS DO SISTEMA

Documento de Requisitos:

Sistema de Cadastro de Pessoas para Sorteio

Documento de Requisitos

Novembro de 2011

A - VISÃO GERAL DO SISTEMA

O sistema para a RT Eventos consiste basicamente no gerenciamento de pessoas para um sorteio de brindes, considerado novidade em um de seus maiores eventos. O sistema deve ainda emitir relatórios e consultas para caráter informativo.

B - REQUISITOS FUNCIONAIS

B1 – Usabilidade

1. Um menu de opções deverá aparecer ao iniciar o sistema e toda vez após uma funcionalidade ter sua execução concluída.

B2 - Lançamentos diversos

1. O sistema deve permitir a inclusão de pessoas no sistema, contendo os seguintes atributos: nome e idade.

B3 - Impressão de diversos tipos de relatórios e consultas

1. O sistema deve permitir a impressão de todas as pessoas presentes no sistema.
2. O sistema deve permitir a impressão de uma pessoa aleatória.

C - REQUISITOS NÃO-FUNCIONAIS

C1 - Lançamentos diversos

1. O sistema deve trazer informações de log quando for cadastrar ou sortear uma pessoa.

APÊNDICE B: GRAMÁTICA, GERADORES E CÓDIGO GERADO

Gramática da linguagem

Arquivo `Tccdsl.xtext`:

```
grammar org.xtext.tcc.Tccdsl with org.eclipse.xtext.common.Terminals

generate tccdsl "http://www.xtext.org/tcc/Tccdsl"

Model:
  (types+=Type)*;

Type:
  Package | TypeDef | Entity;

Package:
  "package" name=JAVAID
  "{"
    (types+=Type)*
  "}";

TypeDef:
  "typedef" name=ID ("mapsto" mappedType=JAVAID)?;

JAVAID:
  ID("." ID)*;

Entity:
  "entity" name=ID ("extends" superEntity=[Entity])?
  "{"
    (attributes+=Attribute)*
    ("functions"
     "{"
        (functions+=Function)*
     "}")?
  "}";

Attribute:
  type=[Type] (many?="*")? name=ID;

Function:
  "void" name=ID ("("(parameters+=Attribute)*")")? | type=[Type]
(many?="*")? name=ID ("("(parameters+=Attribute)*")")?;
```

Arquivo `GenerateTccdsl.mwe2`:

```
module org.xtext.tcc.GenerateTccdsl

import org.eclipse.emf.mwe.utils.*
import org.eclipse.xtext.generator.*
import org.eclipse.xtext.ui.generator.*

var grammarURI = "classpath:/org/xtext/tcc/Tccdsl.xtext"
var file.extensions = "tcc"
var projectName = "org.xtext.tcc.dsl"
var runtimeProject = "../${projectName}"

Workflow {
    bean = StandaloneSetup {
        scanClassPath = true
        platformUri = "${runtimeProject}/.."
    }

    component = DirectoryCleaner {
        directory = "${runtimeProject}/src-gen"
    }

    component = DirectoryCleaner {
        directory = "${runtimeProject}.ui/src-gen"
    }

    component = Generator {
        pathRtProject = runtimeProject
        pathUiProject = "${runtimeProject}.ui"
        pathTestProject = "${runtimeProject}.tests"
        projectNameRt = projectName
        projectNameUi = "${projectName}.ui"
        language = {
            uri = grammarURI
            fileExtensions = file.extensions

            // Java API to access grammar elements (required by
several other fragments)
            fragment = grammarAccess.GrammarAccessFragment {}

            // generates Java API for the generated EPackages
            fragment = ecore.EcoreGeneratorFragment {
            // referencedGenModels = "
            //
platform:/resource/org.eclipse.xtext.xbase/model/Xbase.genmodel,
            //
platform:/resource/org.eclipse.xtext.common.types/model/JavaVMTypes.g
enmodel
            // "
            }

            // Serializer 2.0
            fragment = serializer.SerializerFragment {}

            // the serialization component (1.0)
            // fragment =
parseTreeConstructor.ParseTreeConstructorFragment {}
```

```
            // a custom ResourceFactory for use with EMF
            fragment = resourceFactory.ResourceFactoryFragment {
                fileExtensions = file.extensions
            }

            // The antlr parser generator fragment.
            fragment = parser.antlr.XtextAntlrGeneratorFragment {
            //  options = {
            //      backtrack = true
            //  }
            }

            // java-based API for validation
            fragment = validation.JavaValidatorFragment {
                composedCheck =
"org.eclipse.xtext.validation.ImportUriValidator"
                composedCheck =
"org.eclipse.xtext.validation.NamesAreUniqueValidator"
            }

            // scoping and exporting API
            // fragment = scoping.ImportURIScopingFragment {}
            // fragment = exporting.SimpleNamesFragment {}

            // scoping and exporting API
            fragment = scoping.ImportNamespacesScopingFragment {}
            fragment = exporting.QualifiedNamesFragment {}
            fragment = builder.BuilderIntegrationFragment {}

            // generator API
            fragment = generator.GeneratorFragment {
                generateMwe = true
                generateJavaMain = true
            }

            // formatter API
            fragment = formatting.FormatterFragment {}

            // labeling API
            fragment = labeling.LabelProviderFragment {}

            // outline API
            fragment = outline.OutlineTreeProviderFragment {}
            fragment = outline.QuickOutlineFragment {}

            // quickfix API
            fragment = quickfix.QuickfixProviderFragment {}

            // content assist API
            fragment = contentAssist.JavaBasedContentAssistFragment
{}

            // generates a more lightweight Antlr parser and lexer
tailored for content assist
```

```
            fragment = parser.antlr.XtextAntlrUiGeneratorFragment {}

            // generates junit test support classes into
Generator#pathTestProject
            fragment = junit.Junit4Fragment {}

            // project wizard (optional)
            // fragment = projectWizard.SimpleProjectWizardFragment {
            //      generatorProjectName = "${projectName}"
            //      modelFileExtension = file.extensions
            // }

            // provides the necessary bindings for java types
integration
            fragment = types.TypesGeneratorFragment {}

            // generates the required bindings only if the grammar
inherits from Xbase
            fragment = xbase.XbaseGeneratorFragment {}

            // provides a preference page for template proposals
            fragment = templates.CodetemplatesGeneratorFragment {}

            // rename refactoring
            fragment = refactoring.RefactorElementNameFragment {}

            // provides a compare view
            fragment = compare.CompareFragment {
                 fileExtensions = file.extensions
            }

        }
    }
}
```

Geradores

Classe `TccdslExtensions.xtend`:

```
package org.xtext.tcc

import org.eclipse.emf.ecore.EObject
import org.eclipse.xtext.common.types.JvmDeclaredType
import org.xtext.tcc.tccdsl.*

class TccdslExtensions {
      def String packageName(Object o) {
      switch(o) {
            Package : concatPath(packageName(o.eContainer), o.name)
            EObject : packageName(o.eContainer)
            JvmDeclaredType : o.packageName
            default: null
      }
      }

      def concatPath(String prefix, String suffix) {
      if (prefix.nullOrEmpty)
```

```
		suffix
	else
		prefix + "." + suffix
	}

	def functionHeader(Function f)'''«IF f.type ==
null»void«ELSE»«typeDeclaration(f)»«ENDIF»
«f.name»(«parameters(f)»)'''

	def typeDeclaration(Attribute a)'''«IF
a.many»List<«a.type.name»>«ELSE»«a.type.name»«ENDIF»'''

	def typeDeclaration(Function f)'''«IF
f.many»List<«f.type.name»>«ELSE»«f.type.name»«ENDIF»'''

	def variable(Attribute a)'''«typeDeclaration(a)» «a.name»'''

	def aspectParameters(Function f){
	var pars = ""
	var i = 0
	for(p:f.parameters){
		pars = pars + typeDeclaration(p).toString
		i = i + 1
		if(i < f.parameters.size){
			pars = pars + ", "
		}
	}

	pars
	}

	def parameters(Function f){
	var pars = ""
	var i = 0
	for(p:f.parameters){
		pars = pars + variable(p).toString
		i = i + 1
		if(i < f.parameters.size){
			pars = pars + ", "
		}
	}

	pars
	}

	def parameterDeclaration(Function f){
	var pardecl = ""
	var i = 0
	for(p:f.parameters){
		pardecl = pardecl + p.name
		i = i + 1
		if(i < f.parameters.size){
			pardecl = pardecl + ", "
		}
	}

	pardecl
	}
}
```

Classe `AspectExtensions.xtend`:

```
package org.xtext.tcc.generator

import org.xtext.tcc.TccdslExtensions
import org.xtext.tcc.tccdsl.*
import org.eclipse.xtext.naming.IQualifiedNameProvider
import org.eclipse.xtext.common.types.JvmDeclaredType
import com.google.inject.Inject

class AspectExtensions extends TccdslExtensions {

	@Inject extension IQualifiedNameProvider nameProvider

	def aspect(Entity e)'''
	«IF e.eContainer instanceof Package»
	package «e.packageName».aspect;
	«ELSE»
	package aspect;
	«ENDIF»

	import java.util.List;
	import «e.fullyQualifiedName»;

	public aspect Aspecto«e.name.toFirstUpper»{
	«FOR f:e.functions»
	pointcut metodo«f.name.toFirstUpper»(): call (*
«e.packageName».control.«e.name.toFirstUpper»Controller.«f.name»(..))
;
	pointcut metodo«f.name.toFirstUpper»2(): call (public static
«IF f.type != null»«f.type.name»«ELSE»void«ENDIF»
«e.packageName».control.«e.name.toFirstUpper»Controller.«f.name»(«asp
ectParameters(f)»));
	«ENDFOR»

	«FOR f:e.functions»
	before(): metodo«f.name.toFirstUpper»(){
		System.out.println(thisJoinPoint.getSignature() + " -
Conectou!");
	}
	after() returning: metodo«f.name.toFirstUpper»2(){
		System.out.println(thisJoinPoint.getSignature() + " -
Inseriu!");
	}
	«ENDFOR»
	}
	'''
}
```

Classe `ModelExtensions.xtend`:

```
package org.xtext.tcc.generator

import org.xtext.tcc.TccdslExtensions
import org.xtext.tcc.tccdsl.*
import org.eclipse.xtext.naming.IQualifiedNameProvider
import com.google.inject.Inject

class ModelExtensions extends TccdslExtensions {
	def compile(Entity e) '''
	«IF e.eContainer instanceof Package»
	package «e.packageName»;
	«ENDIF»

	import java.util.List;
	«IF e.eContainer instanceof Package»
	import «e.packageName».control.«e.name»Controller;
	«ELSE»
	import control.«e.name»Controller;
	«ENDIF»

	«classes(e)»
	'''

	def classes(Entity e)'''
	public class «e.name»«IF e.superEntity != null» extends
«e.superEntity.name»«ENDIF»{

	«FOR a:e.attributes»
	private «variable(a)»;
	«ENDFOR»

	«FOR a:e.attributes»
	public «typeDeclaration(a)» get«a.name.toFirstUpper»(){
		return «a.name»;
	}

	public void set«a.name.toFirstUpper»(«variable(a)»){
		this.«a.name» = «a.name»;
	}
	«ENDFOR»
	«FOR f:e.functions»
	public «functionHeader(f)»{
		«IF f.type != null»return
«ENDIF»«e.name.toFirstUpper»Controller.«f.name»(«parameterDeclaration
(f)»);
	}
	«ENDFOR»
	}
	'''
}
```

Classe `ControllerExtensions.xtend`:

```
package org.xtext.tcc.generator

import org.xtext.tcc.TccdslExtensions
import org.xtext.tcc.tccdsl.*
import org.eclipse.xtext.naming.IQualifiedNameProvider
import org.eclipse.xtext.common.types.JvmDeclaredType
import com.google.inject.Inject

class ControllerExtensions extends TccdslExtensions {

      @Inject extension IQualifiedNameProvider nameProvider

      def control(Entity e)'''
      «IF e.eContainer instanceof Package»
      package «e.packageName».control;
      «ELSE»
      package control;
      «ENDIF»

      import java.util.List;
      import «e.fullyQualifiedName»;

      public class «e.name.toFirstUpper»Controller{
      «FOR f:e.functions»

      public static «functionHeader(f)»{

      }
      «ENDFOR»
      }
      '''
}
```

Classe `TccdslGenerator.xtend`:

```
/*
 * generated by Xtext
 */
package org.xtext.tcc.generator

import org.eclipse.emf.ecore.resource.Resource
import org.eclipse.xtext.generator.IGenerator
import org.eclipse.xtext.generator.IFileSystemAccess

import static extension
org.eclipse.xtext.xtend2.lib.ResourceExtensions.*
import org.xtext.tcc.tccdsl.*
import org.eclipse.xtext.naming.IQualifiedNameProvider
import com.google.inject.Inject

class TccdslGenerator implements IGenerator {


      @Inject extension IQualifiedNameProvider nameProvider

      @Inject extension ModelExtensions models
```

```
	@Inject extension AspectExtensions aspects
	@Inject extension ControllerExtensions controllers

	override void doGenerate(Resource resource, IFileSystemAccess
fsa) {
	//TODO implment me
	for(e : resource.allContentsIterable.filter(typeof(Entity))){
		var fileNames = e.fullyQualifiedName.segments
		var entityFileName = ""
		var aspectFileName = ""
		var controllerFileName = ""
		var i = 1
		for(fileName : fileNames){
			if(i < fileNames.size()){
				entityFileName = entityFileName + fileName +
"/"
				aspectFileName = aspectFileName + fileName +
"/"
				controllerFileName = controllerFileName +
fileName + "/"
			}else{
				entityFileName = entityFileName + fileName +
".java"
				aspectFileName = aspectFileName +
"aspect/Aspecto" + fileName.toFirstUpper + ".aj"
				controllerFileName = controllerFileName +
"control/" + fileName.toFirstUpper + "Controller.java"
			}
			i = i + 1
		}

		fsa.generateFile(entityFileName,e.compile)
		fsa.generateFile(aspectFileName,e.aspect)
		fsa.generateFile(controllerFileName,e.control)
	}

	var generateMain = true
	if(generateMain){

fsa.generateFile("app/init/AppInit.java",generateMain.app)
	}
	}
```

Modelagem

Arquivo `dsl.tcc`:

```
typedef int
typedef String

package tcc.model{
      entity Pessoa{
      String nome
      int idade
      functions{
            void cadastrar(
                  Pessoa* lp
                  Pessoa p
            )
            void imprimir(
                  Pessoa* lp
            )
            Pessoa sortear(
                  Pessoa* lp
            )
      }
      }
}
```

Código gerado

Classe `AppInit`:

```
package app.init;

public class AppInit{
      public static void main(String args[]){

      }
}
```

Classe `Pessoa`:

```
package tcc.model;

import java.util.List;
import tcc.model.control.PessoaController;

public class Pessoa{

      private String nome;
      private int idade;

      public String getNome(){
            return nome;
```

```
	}

	public void setNome(String nome){
		this.nome = nome;
	}
	public int getIdade(){
		return idade;
	}

	public void setIdade(int idade){
		this.idade = idade;
	}

	public void cadastrar(List<Pessoa> lp, Pessoa p){
		PessoaController.cadastrar(lp, p);
	}

	public void imprimir(List<Pessoa> lp){
		PessoaController.imprimir(lp);
	}

	public Pessoa sortear(List<Pessoa> lp){
		return PessoaController.sortear(lp);
	}
}
```

Classe `PessoaController`:

```
package tcc.model.control;

import java.util.List;
import tcc.model.Pessoa;

public class PessoaController{

	public static void cadastrar(List<Pessoa> lp, Pessoa p){

	}

	public static Pessoa sortear(List<Pessoa> lp){

	}
}
```

Aspecto `AspectoPessoa`:

```
package tcc.model.aspect;

import java.util.List;
import tcc.model.Pessoa;

public aspect AspectoPessoa{
	pointcut metodoCadastrar(): call (*
		tcc.model.control.PessoaController.cadastrar(..));
	pointcut metodoCadastrar2(): call (public static void
```

```
                  tcc.model.control.PessoaController.cadastrar(
                  List<Pessoa>, Pessoa));
      pointcut metodoImprimir(): call (*
            tcc.model.control.PessoaController.imprimir(..));
      pointcut metodoImprimir2(): call (public static void
            tcc.model.control.PessoaController.imprimir
            (List<Pessoa>));
      pointcut metodoSortear(): call (*
            tcc.model.control.PessoaController.sortear(..));
      pointcut metodoSortear2(): call (public static Pessoa
            tcc.model.control.PessoaController.sortear(
            List<Pessoa>));

      before(): metodoCadastrar(){
      System.out.println(thisJoinPoint.getSignature() + " -
Conectou!");
      }
      after() returning: metodoCadastrar2(){
      System.out.println(thisJoinPoint.getSignature() + " -
Inseriu!");
      }
      before(): metodoImprimir(){
      System.out.println(thisJoinPoint.getSignature() + " -
Conectou!");
      }
      after() returning: metodoImprimir2(){
      System.out.println(thisJoinPoint.getSignature() + " -
Inseriu!");
      }
      before(): metodoSortear(){
      System.out.println(thisJoinPoint.getSignature() + " -
Conectou!");
      }
      after() returning: metodoSortear2(){
      System.out.println(thisJoinPoint.getSignature() + " -
Inseriu!");
      }
}
```

APÊNDICE C: CÓDIGO DA APLICAÇÃO

Classes modificadas na aplicação

Classe `AppInit`:

```
package app.init;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

import tcc.model.Pessoa;

public class AppInit{
	public static void main(String args[]){
	List<Pessoa> listaPessoas = new ArrayList<Pessoa>();
	BufferedReader input = new BufferedReader(new InputStreamReader
(System.in));
	int op = 0;


	while(op != 4){
		System.out.println("1) Cadastrar uma pessoa");
		System.out.println("2) Imprimir pessoas cadastradas");
		System.out.println("3) Sortear uma pessoa");
		System.out.println("4) Sair");
		System.out.println();
		System.out.print("Opção: ");
		try{
			op = Integer.parseInt(input.readLine());
		}catch(Exception e){
			op = 0;
		}

		switch(op){
			case 1:
				Pessoa pessoaCadastro = new Pessoa();
				System.out.println();
				System.out.print("Nome da pessoa: ");
				System.out.flush();
				try{
					pessoaCadastro.setNome(
					input.readLine());
				}catch(Exception e){
					System.out.println("Erro!");
				}

				Integer idade = null;
				while(idade == null){
					try{
```

```
                                    System.out.print("Idade da
pessoa: ");
                                    idade =
Integer.parseInt(input.readLine());
                              }catch(Exception e){
                                    System.out.println("Idade
inválida! Tente novamente");
                              }
                        }
                        pessoaCadastro.setIdade(idade.intValue());
                        pessoaCadastro.cadastrar(listaPessoas,
pessoaCadastro);
                        System.out.println();
                        break;
                  case 2:
                        Pessoa pessoaSorteio = new Pessoa();
                        pessoaSorteio.sortear(listaPessoas);
                        try{
                              System.out.println();
                              System.out.println("Nome: " +
pessoaSorteio.getNome());
                              System.out.println("Idade: " +
pessoaSorteio.getIdade());
                              System.out.println();
                        }catch(Exception e){
                              System.out.println();
                              System.out.println("Nenhuma pessoa
cadastrada. Tente novamente.");
                              System.out.println();
                        }
                        break;
                  case 3:
                        break;
                  default: System.out.println("Opção inválida. Tente
novamente");
            }
      }
      }
}
```

Classe `PessoaController`:

```
package tcc.model.control;

import java.util.List;
import tcc.model.Pessoa;

public class PessoaController{

      public static void cadastrar(List<Pessoa> lp, Pessoa p){
      lp.add(p);
      }

      public static void imprimir(List<Pessoa> lp){
      if(lp.size() > 0){
            for(Pessoa p : lp){
                  System.out.println();
                  System.out.println("Nome: " + p.getNome());
```

```
                        System.out.println();
                  }
            }else{
                  System.out.println();
                  System.out.println("Nenhuma pessoa cadastrada.
                  Tente novamente.");
                  System.out.println();
            }
      }

      public static Pessoa sortear(List<Pessoa> lp){
            if(lp.size() > 0){
                  int i = (int)(Math.random() * lp.size());

                  return lp.get(i);
            }

            return null;
      }
}
```