UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

UNIDADE ACADÊMICA DE SERRA TALHADA

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Projeto PaMDA: um *framework* semântico para geração de sistemas

PABLO VINICIUS ALVES DE BARROS

Serra Talhada,

Julho, 2011

PABLO VINICIUS ALVES DE BARROS

Projeto PaMDA: um *framework* semântico para geração de sistemas

Trabalho de Conclusão de Curso apresentada ao curso de Bacharelado em Sistemas de Informação da Unidade Acadêmica de Serra Talhada da Universidade Federal Rural de Pernambuco como requisito parcial à obtenção do grau de Bacharel.

Orientador: Prof. M.e Richarlyson Alves D'Emery

Serra Talhada

2011

Ficha catalográfica
Setor de Processos Técnicos da Biblioteca Central – UFRPE

Barros, Pablo

Projeto PaMDA: um *framework* semântico para geração de sistemas – SERRA TALHADA: PABLO BARROS, 2011

77 folhas : fig.

Monografia (Graduação) - Universidade Federal Rural de Pernambuco. Unidade Acadêmica de Serra Talhada. Bacharelado em Sistemas de Informação, 2011.

Inclui bibliografia a apêndice.

1. Ontologia. 2. Web Semântica 3.Gerador de Código 4.MDA 5. Motor de Inferência I. Barros, Pablo (Richarlyson D'Emery)

574. 018 2 CDD (edição)

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
UNIDADE ACADÊMICA DE SERRA TALHADA
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

PABLO VINÍCIUS ALVES DE BARROS

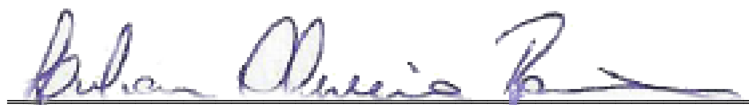
Projeto PaMDA: um *framework* semântico para geração de sistemas

Trabalho de Conclusão de Curso julgado adequado para obtenção do título de Bacharel em Sistemas de Informação, defendida e aprovada por unanimidade em 05/07/2011 pela banca examinadora.

Banca Examinadora:

Prof. M.e Richarlyson Alves D'Emery
Orientador
Universidade Federal Rural de Pernambuco

Prof. M.e Sérgio de Sá Leitão Paiva Júnior
Universidade Federal Rural de Pernambuco



Prof.ª M.ª Lilian Oliveira Ramires
Universidade Federal Rural de Pernambuco

Prof.ª M.ª Isledna Rodrigues de Almeida
Universidade Federal Rural de Pernambuco

DEDICATÓRIA

Dedico este trabalho aos meus pais por terem me apoiado em todos as etapas da minha vida.

AGRADECIMENTOS

Agradecimentos especiais ao Prof. Richarlyson Alves D'Emery por ter auxiliado e orientado em toda as etapas deste trabalho.

A todos os meus professores da UAST-UFRPE, que auxiliaram direta ou indiretamente na conclusão deste projeto.

A turma 2007.1 que apesar de todos os problemas encontrados conseguimos solucioná-los e chegar ao fim dessa jornada, de uma forma ou de outra, vitoriosos.

A toda a equipe Chronos por ter auxiliado e incentivado, mesmo que de forma totalmente errada, este projeto, sempre estando dispostos a opinarem e auxiliarem na construção técnica desta monografia.

A todo o pessoal do Nex por ter acompanhado de perto todas as madrugadas prolongadas que permitiram a conclusão deste trabalho, e de toda minha vida acadêmica.

A minha família que me apoiou em todas as formas possíveis e imagináveis a realização deste e de todos os outros projetos em que participei.

Veni.Vidi.Vici.

RESUMO

Esta monografia investiga a utilização de ontologias para o desenvolvimento de um *framework* que permite a geração de sistemas com características semânticas a partir da descrição dada por um usuário. O trabalho envolve três partes principais: (1) Representação semântica do modelo de domínio do sistema do usuário; (2) Auxilio na captação dos dados para o modelo semântico, através de um motor de inferência de dados; (3) Geração de código a partir do modelo captado pelo sistema e modelado na ontologia de domínio. Para realizar a representação semântica do modelo de domínio, foi criada uma ontologia genérica que representa todos os componentes que compõem um *software*, para auxiliar o usuário a captar os dados e preencher esse modelo. Um motor de inferência foi desenvolvido utilizando as regras definidas na ontologia e utilizando como base inferida os modelos já criados por outros projetos. Para gerar o código do sistema foi desenvolvido uma série de geradores de código que utilizam como base técnicas de *Model Driven Architecture* (MDA) para transformar o modelo descrito na ontologia preenchida pelo usuário em código. Para a arquitetura proposta foi disponibilizada um *framework* que poderá ser utilizado para geração de *softwares* em fábricas de *software* que tenham por objetivo a otimização do seu desenvolvimento em *softwares* com características semânticas, *framework* este desenvolvido obedecendo padrões de projeto, a exemplo de *Value Object*, *Factory*, *Façade* e *Data Acces Object*. Para validação da arquitetura proposta foi desenvolvido o projeto PaMDA que consiste em uma ferramenta disponível online para a geração de sistemas semânticos.

Palavras-chave: Ontologia, Web Semântica, Gerador de Código, MDA, Motor de Inferência.

ABSTRACT

This work investigates the ontology´s utilization for propose to develop a framework that makes possible the generation of semantic systems using the user´s given description. The work evolves three parts: (1) Semantic representation of a domain model for the user´s system.(2)Some help at the data captation used in the semantic model, by a data inference motor. (3) Code generation using the domain model as bases and modeled into the ontology. To make the semantic representation of domain model was created a generic ontology that contain all the parts of software, to help the data acquisition from a user and model the domain model. An inference motor war developed using the well-defined rules from the ontology and uses the models created for others users to be the inferred base. To code generation was developed a group of code generators that uses Model Driven Architecture (MDA) techniques for the transformation of the ontology model into code. For the proposed architecture was available a framework that can be used to generate software in soft-houses that a optimization propose in your software development, this framework was developed using project patterns as the Value Object, Factory, Façade and Data Access Object. For validation of the proposed architecture was developed the *Projeto PaMDA,* an online tool for semantic code generation.

Keywords: Ontology, Semantic Web, Code Generation , MDA, Inference Motor.

LISTA DE ILUSTRAÇÕES

Figura 2.1	Evolução da Web	19
Figura 2.2	Exemplo de Sabedoria	20
Figura 2.3	Etapas para Criação de uma Ontologia	23
Figura 2.4	Descrição de uma Ontologia Usando a Linguagem OWL	25
Figura 3.1	Padrão Fachada	28
Figura 3.2	Exemplo da Utilização do Padrão *Factory*	29
Figura 3.3	Exemplo de Utilização do Padrão VO	29
Figura 4.1	Ontologia de Domínio	34
Figura 4.2	Camadas da Arquitetura	38
Figura 4.3	Classes Básicas Contidas no namespace VO	38
Figura 4.4	Modelo Entidade-Relacionamento do Sistema	39
Figura 4.5	Estrutura DataSet do Ssistema	40
Figura 4.6	Classes Contida na namespace Entidades da Camada de Negócios	41
Figura 4.7	Método `editar( )` da Classe `BOProjeto`	42
Figura 4.8	Método `buscarEntidade( )`	42
Figura 4.9	Método `copiarAtributo( )`	43
Figura 4.10	Arquitetura do Sistema Gerado	43
Figura 4.11	Camada DAL da Arquitetura Gerada	44
Figura 4.12	Camada DAL Arquitetura Gerada	45
Figura 4.13	Interface `IGerador`	46
Figura 4.14	Classe `Gerador` do namespace geradorSGBD	46
Figura 4.15	Resultado da Normalização para o Atributo Multivalorado Atributo2	48
Figura 4.16	Resultado da Normalização para o Atributo Tipado Atributo2	49
Figura 4.17	Algoritmo para Retirar todos os Atributos Multivalorados ou Tipados	49
Figura 4.18	*Factory* dos Geradores do SGBD	50
Figura 4.19	Classe `SGBDMySQL`	50
Figura 4.20	Método `criarBancoDados( )`	51
Figura 4.21	Método `criarTabela( )`	51
Figura 4.22	Método `criarCampo( )`	52
Figura 4.23	Fachada Gerador VO	53
Figura 4.24	Factory dos Geradores do VO.	53
Figura 4.25	Classe Geradora PHPVO	54
Figura 4.26	Gerador DAO	54
Figura 4.27	Factory DAO	55
Figura 4.28	Classe Geradora Conexão	55
Figura 4.29	Classe Geradora Interface	56

Figura 4.30	Classe Geradora DAO	56
Figura 4.31	Método `adicionarAtributoTipado( )`	57
Figura 4.32	Método `adicionarAtributoMultivalorado( )`	58
Figura 4.33	*Factory* Gerada	58
Figura 4.34	Fachada Gerador do `geradorView`	59
Figura 4.35	Factory do geradorView	59
Figura 4.36	Geradores do geradorView	60
Figura 4.37	Método `gerarInput( )`	60
Figura 4.38	Reasoner Semântico	61
Figura 4.39	Método `entidadesSimilares( )`	62
Figura 4.40	Método `atributosSimilares( )`	63
Figura 4.41	Método `isEntidadeIgual( )`	64
Figura 4.42	Método `isAtributoIgual( )`	64
Figura 4.43	Método `isEntidadeVerificada( )`	65
Figura 4.44	Método `extrairEntidades( )`	65
Figura 4.45	Método `extrairAtributos( ).`	66
Figura 5.1	Tela Incial Projeto PaMDA	67
Figura 5.2	Tela Principal	68
Figura 5.3	Tela da Etapa de Informar as Entidades	68
Figura 5.4	Método para Criar um Novo Projeto	69
Figura 5.5	Extração de Entidades na Página `entidadesProjeto.aspx`	69
Figura 5.6	Tela Personalizada para Criação de Interfaces	70
Figura 5.7	Tela Ilustração da Consulta a Classes Semelhantes	70
Figura 5.8	Tela Ilustração a AtributosEntidades.aspx	71
Figura 5.9	Tela de Resumo de Projeto	72
Figura 5.10	Arquivos Gerados	72

LISTA DE ABREVIATURAS E SIGLAS

VO	Value Object
DAL	Data Access Layer
DAO	Data Access Object
SGBD	Sistema Gerenciador de Banco de Dados
URI	Unified Resource Identifier
OWL	Web Ontology Language
MDA	Model Driven Architecture
PIM	Plataform Independent Model
PM	Plataform Model

SUMÁRIO

1 INTRODUÇÃO **14**
1.1 Motivação 14
1.2 Objetivos da monografia 16
1.3 Descrição do trabalho 16
1.4 Organização da monografia 17
2 WEB SEMÂNTICA **18**
2.1 Introdução 18
2.2 Ontologias 21
2.3 Metodologias para criação de ontologias 22
2.4 Resumo do capítulo 25
3 PADRÕES DE DESENVOLVIMENTO **26**
3.1 MDA 26
3.2 Padrões de projeto 27
3.3 Linguagens para desenvolvimento WEB 30
3.4 Resumo do capítulo 32
4 ARQUITETURA DO SISTEMA PROPOSTO **33**
4.1 Criação da ontologia 33
4.2 Descrição da arquitetura geradora 37
4.2.1 Camada de dados 38
4.2.2 Camada de negócios 40
4.2.2.1 Entidades 40
4.2.2.2 Geradores 43
4.2.2.2.1 Gerador SGBD 46
4.2.2.2.2 Gerador VO 52
4.2.2.2.3 GeradorDAO 54
4.2.2.2.4 GeradorView 59
4.3 Wizzard 60
4.4 Resumo do capítulo 66
5 ESTUDO DE CASO **67**
5.1 Introdução 67
5.2 Resumo do capítulo 72
6 CONCLUSÃO **73**
6.1 Considerações finais 73
6.2 Constribuições deste trabalho 73
6.3 Proposta para trabalhos futuros 74
REFERÊNCIAS BIBLIOGRÁFICAS **75**
ANEXO A **75**

1 Introdução

Neste capítulo é apresentada a motivação desta monografia e apresenta uma breve introdução do trabalho. Na seção 1.1 demarcam-se as motivações desta arquitetura proposta. A seção 1.2 contém uma sucinta descrição dos objetivos deste trabalho. A seção 1.3 contém uma sucinta descrição do trabalho realizado na monografia. E na seção 1.4 é fornecida uma visão dos capítulos da monografia.

1.1 Motivação

A busca pelo desenvolvimento de ferramentas que auxiliem no desenvolvimento de programas computacionais sempre fascinou os desenvolvedores de projetos. Nas pesquisas por sistemas que possam gerar a codificação de projetos, uma das mais intensas batalhas que os programadores vêm travando na competitiva indústria de *software* é tentar produzir *framework*s aptos a ajudar, gerar, estender ou melhorar a produção dos *softwares* semânticos além dos seus limites naturais.

No entanto, dentre os produtos gerados, existe a dificuldade de na produção destes *softwares*, nos quais precisa casar a utilização de muitas ferramentas em separado, o que causa um sério problema na integração dos sistemas. Os sistemas semânticos, em especial, se caracterizam pelos muitos aspectos inerentes ao processo da busca da informação em sistemas web, nos quais existem diversas tentativas no intuito de produzi-los. Este fato motivou o estudo dos processos envolvidos na representação dos dados; e mesmo das tecnologias para a modelagem destes, impulsionaram o estudo e o desenvolvimento de vários sistemas para a representação dos dados baseados em ontologia.

Atualmente denomina-se dados semânticos as informações que estão bem catalogadas de forma que possam ser interpretadas por um motor de inferência de forma que máquinas possam extrair as informações necessárias para suas operações sem a necessidade de alteração dos dados (GOUVEIA, 2007).

Os geradores de sistemas semânticos têm sido utilizados com grande sucesso em diversas áreas, como, por exemplo, o Total Code Generator (TOTAL CODE GENERATOR, 2008), uma solução completa para geração de código baseada na modelagem do banco de dados e no trabalho de Chauvel e Jézéquel (2005), que usam a geração de código semântico a partir de diagramas UML de estados, e tem despertado especial interesse na Engenharia de *Software*, onde podem ser utilizados como ferramentas na busca da informação, principalmente na web como é o caso do OntoSearch (ZHANG; VASCONCELOS; SLEEMAN, 2005), um motor de buscas ontológicas que funciona na web e utiliza motores de inferência para realizar buscas baseadas em uma ontologia e o Swoogle (DING et al., 2005) que é um indexador web que utiliza motores semânticos para realizar buscas em arquivos RDF e OWL. Existem muitas outras aplicações, a exemplo da Wikipedia (2011) que utiliza ontologias para catalogar seus versículos online, aplicações de comércio eletrônico, como o Amazon.com (1995) no qual utiliza ontologias para realizar busca e inferência semântica em seus produtos, de forma que os produtos sejam filtrados de forma diferente para cada usuário que realiza uma compra.

Devido à importância da representação semântica dos dados, o estudo de ontologia tem recebido crescente atenção em pesquisas e desenvolvimentos. Igualmente importante para as mais diversas áreas, a representação de dados é um campo em crescente expansão no domínio da Engenharia de *Software*. Dentre as abordagens investigadas objetivando a representação dos dados, resultados promissores vêm sendo encontrados tanto nas abordagens de modelagem como na fábrica de *softwares*, como no trabalho de (DOMINGUÉZ et al., 2009), que exibe a criação de uma ontologia que abarca a estruturação de todos os processos envolvidos em uma fábrica de *software*, para a auxiliar nas práticas de reuso e melhoria de todos os procedimentos envolvidos. No decorrer deste trabalho são abordadas técnicas já consagradas na literatura como a Ontologia, além da produção do *framework* com o uso de padrões no Projeto de *Softwares*, como os padrões *Value Object*, *Data Access Layer*, *Data Access Object, Factory* e *Façade*, capazes de tornar o *software* com uma arquitetura bem definida para o desenvolvimento de aplicações semânticas com uma interface amigável, de forma a melhorar a experiência do usuário ao utilizá-lo, incrementando sua eficiência em gerar aplicações semânticas. Este ambiente desenvolvido é portável, de forma a facilitar seu desenvolvimento de especializações futuras facilitando sua adaptação às tecnologias que estão por vir. Através do uso da Web Semântica, estimula-se a usabilidade do sistema, pois o próprio sistema pode auxiliar o usuário a obter um objeto mais apurado, satisfazendo assim o seu desejo ao utilizar o ambiente.

1.2 Objetivos da monografia

Neste trabalho se estuda a representação dos dados por Ontologia, bem como as diversas abordagens no desenvolvimento de projetos de *software* disponíveis na literatura para este fim, apresentando os procedimentos, vantagens e desvantagens.

O objetivo geral desta monografia é propor um *framework* a ser disponibilizado na web para o desenvolvimento de aplicações semânticas, que futuramente poderá servir de base para comparação com outros *framework*s.

Dentre os objetivos específicos deste trabalho, temos o desenvolvimento de uma ontologia de domínio específico, descrita no Capítulo 3, para modelar o produto da aplicação, assim como uma arquitetura para geração de código, através da ontologia especificada e um módulo semântico que é utilizado para coletar informações mais precisas do usuário. No intuito de validar a arquitetura proposta se propõe o Projeto PaMDA, a ser disponibilizado na internet como ferramenta para a geração de sistemas semânticos.

1.3 Descrição do trabalho

Este trabalho consistiu das seguintes etapas: (1) criação de uma ontologia de domínio; (2) construção de uma arquitetura semântica que auxilie o usuário na coleta das informações relevantes ao projeto proposto e (3) um conjunto de classes geradoras, que criarão todo o código do sistema a ser gerado, de acordo com a modelagem do usuário catalogada na ontologia de domínio do sistema.

A fase de desenvolvimento do *framework* compreendeu a sua utilização em uma aplicação específica, visando estudar o seu comportamento. A construção do Projeto PaMDA está intimamente ligada ao propósito geral do projeto, uma vez que possibilita a construção de sistemas para quaisquer aplicações específicas.

A avaliação do sistema compreendeu uma análise a partir do estudo de caso, uma aplicação web que utiliza os projetos criados pelos usuários como base para o motor semântico denominada, Projeto PaMDA , de forma que todos os projetos criados pelo sistema desenvolvido para o estudo de caso, sirvam como material semântico para ser utilizado pelo *framework* proposto.

Todo os sistema foi desenvolvido utilizando a plataforma .NET *Framework* 4.0 na linguagem C# e com o SGBD MySQL para realizar toda a persistência de dados. O Estudo de caso foi desenvolvido utilizando tecnologias ASP.NET , de forma que está totalmente multiplataforma, rodando sob a plataforma WEB.

1.4 Organização da monografia

Esta monografia está dividida em seis capítulos, no segundo destes se expõe uma definição e explanação sobre a Web Semântica, suas aplicações e funcionamento e as etapas para construção de um *softwar*e semântico.

No Capítulo 3 é mostrada e explanada a arquitetura proposta deste trabalho, exibindo todos os componentes e camadas utilizadas para a construção dos geradores de código, modelagem da ontologia de domínio e do motor de inferência semântica.

No Capítulo 4 é discutida algumas tecnologias de projeto utilizadas neste trabalho, como alguns padrões de projeto utilizados além de um breve resumo sobre desenvolvimento web.

No Capítulo 5 está descrita a principal contribuição desta monografia, detalhando a arquitetura do sistema proposto, enquanto que no Capítulo 6 é apresentado um *softwar*e denominado Projeto PaMDA que utiliza a arquitetura proposta para obter *softwares* descritos por um usuário, com o auxilio do motor semântico do *framework* aqui proposto.

As considerações finais sobre o trabalho apresentado nesta monografia são tecidas no Capítulo 6. Além das considerações finais, este capítulo apresenta uma descrição das contribuições desta monografia e algumas propostas para trabalhos futuros.

Ao final desta monografia é fornecido o Anexo A contendo um CD-ROM com o sistema desenvolvido.

2 Web Semântica

Neste capítulo é apresentada a Web Semântica, analisando sua evolução a partir da Web 1.0 até chegar aos dias atuais, e nas aplicações com motores de inferência. Na seção 2.1 dar-se uma explicação sobre a evolução da Web Semântica e sua utilização. A seção 2.2 exibe uma explanação sobre ontologias e seu papel em sistemas semânticos. A seção 2.3 contém uma descrição de metodologias para criação de ontologias e uma linguagem para escrever ontologias em linguagem formal. Ao final do Capítulo, a seção 2.4 realiza um breve resumo.

2.1 Introdução

A idéia de semântica computacional nos remete aos sonhos e desejos de toda a comunidade científica da década de 1970: criar máquinas que tenham o poder de compreender a linguagem natural humana de forma a poder captar, interpretar, processar e responder a um determinado estímulo semântico dado por humanos, sonho que vem sendo perseguido por vários anos e na última década ganhou um novo estágio: a Web Semântica.

Pode-se dizer que a Web Semântica nada mais é do que uma especialização da web atual, também chamada de Web 1.0, definida por JACOBS (2004) como um espaço de informações cujos itens de interesse, referenciados a recursos, são identificados por identificadores globais chamados *Uniform Resource identifiers* (URI), que dá às máquinas um maior poder de interpretação dos dados presentes na Web, de forma a auxiliar ou até mesmo substituir os humanos em algumas funções que antes eram impossíveis devido ao seu conteúdo semântico, e dessa forma instável, para poder ser explicitado em um algoritmo padrão.

Atualmente existe uma grande quantidade de informações que são usadas e que passam despercebidas pelos padrões da Web 1.0, a exemplo, dos dados financeiros em um banco, fotos e agenda de compromissos. Mas não é possível ver fotos em um calendário de

forma, a saber, o que se estava fazendo no dia daquela foto ou poder ver o saldo da conta bancária nesse mesmo dia. Mas por que não é possível fazer isso? (W3C, 2001).

Tim Bernes-Lee (2011), um cientista do Centro Europeu de Pesquisas Nucleares (CERN), é uma das mentes por trás da definição da Web 1.0 e da Web Semântica, sua famosa frase representa bem o que representa a Web Semântica para ele:

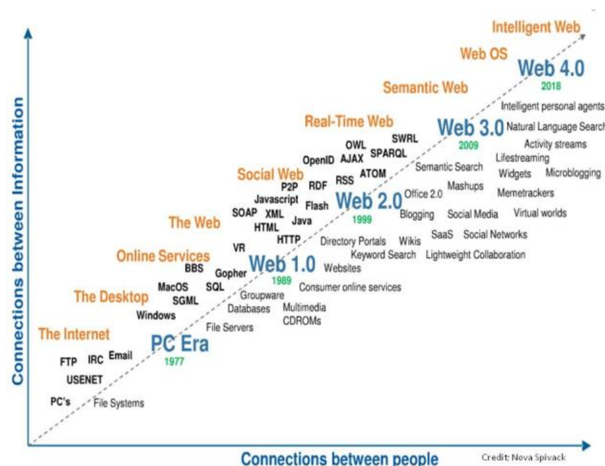
> Eu tenho um sonho para a web em que computadores se tornem capazes de analisar todas as informações contidas na web: conteúdo, links e transações entre pessoas e computadores. Uma ‘Web Semântica’, que deve possibilitar isso. Ainda não é possível, mas quando for, os mecanismos diários de troca, burocracia e organização da vida diária serão manipulados por máquinas que conversam com máquinas. O que o pessoal de Agentes Inteligentes vem sonhado por anos nós vamos finalmente materializar. (BERNES-LEE, 2011).

Na evolução da Web 1.0, surgiu a Web 2.0, na qual permite que esses documentos sejam criados e mantidos pelos próprios usuários, gerando assim um sentido de web colaborativa. Sua definição surge em 1999:

> A Web que nós conhecemos agora, que é carregada através da janela de um browser e é essencialmente estática. Ela é somente um embrião da web que está vindo. Os primeiros passos da Web 2.0 estão começando a aparecer e nós estamos somente começando a ver como esse embrião será desenvolvido. A web será entendida, não como sucessivas telas de texto e gráficos, mas como um mecanismo de transporte, um meio em que a interatividade aparece. (DINUCCI, 1999).

A partir da denominação da Web 2.0 a Web Semântica também é chamada de Web 3.0, demonstrando uma evolução na definição de Web, evolução esta de cunho físico e semântico, e após 10 anos da definição de DiNucci (1999), a Web 3.0 realmente entra em vigor, através das várias aplicações semânticas que estão no ar hoje em dia. A evolução histórica da Web é ilustrada na Fig. 2.1.

Figura 2.1 – Evolução da Web



Fonte: SPIVAK (2010)

A Web Semântica utiliza os dados disponibilizados pelos usuários colaborativos da Web 2.0 para gerar informações, trechos de conhecimento que podem ser interpretados e

inferidos, e a partir dessas informações é possível extrair um conhecimento inferido sobre o todo, de forma a otimizar a tomada de decisões pelos humanos que utilizam a Web 3.0. A partir desta hierarquia gera-se ainda mais um produto: "a sabedoria", que pode ser definida como a aplicação de experiências passadas, a partir dos humanos ou dos algoritmos semânticos, na interpretação do conhecimento tornado a decisão ainda mais eficaz, Ackof (1989) apud Rhind-Tutt (2010) demonstra um exemplo prático da hierarquia de funcionamento da criação de sabedoria. A Fig. 2.2 ilustra um exemplo de sabedoria.

Figura 2.2 – Exemplo de Sabedoria

```
Dado:          Está chovendo.
Informação:    A temperatura caiu 15 graus e começou a chover.
Conhecimento: Se a umidade está alta e a temperatura caiu drasticamente, a
               atmosfera não é capaz de segurar o vapor de água das
               nuvens e então a chuva cai.
Sabedoria:     Como já aconteceu em épocas passadas, essa chuva pode ser
               rápida e vai acabar logo.
```

Fonte: Ackof (1989) apud Rhind-Tutt (2010)

Como todos os dados já estão na rede graças à colaboração dos usuários na Web 2.0, a Web Semântica passa agora a ser feita por aplicações de *softwar*e que coletam os dados já existentes na rede e realizam inferências semânticas, de forma a limitar o desenvolvimento dessas aplicações a somente os motores semânticos, trazendo a Web 3.0 quase que exclusivamente para o escopo de desenvolvimento de *softwares*, ao contrário do que houve com a Web 2.0, que sofreu um grande avanço de outras áreas como usabilidade e design gráfico.

Existem várias aplicações semânticas que funcionam nas diretrizes da Web 3.0 como o Livro de Fatos Mundias (*World Factbook*) Ackof (1989), mantido pela Central de Inteligência Americana (CIA), que está todo formatado e apresentado utilizando os conceitos da Web Semântica. A Wikipedia (2001), a gigante enciclopédia mundial, utiliza dados semânticos para catalogar, exibir e manter seus dados colaborativos.

Outro *softwar*e que utiliza Web Semântica é o Glue (ADAPTIVEGLUE, 2009), um *softwar*e que funciona ligado à barra de ferramentas do browser e permite o compartilhamento de informações sobre uma página visitada. O *softwar*e indica quem recomendou a página, outras páginas que se possa gostar além de informar o que outros usuários pensam sobre a página visitada.

2.2 Ontologias

O termo ontologia vem do grego "*onto*" e "*logos*", significando "conhecimento do ser", definido na filosofia como a forma que trata do ser, da realidade, da existência dos entes e das questões metafísicas em geral. Aristóteles utilizava ontologias para servirem de base para a classificação de entidades, utilizando o termo "*differentia*" para distinguir propriedades diferentes entre entidades parecidas, enquanto que o Dicionário Oxford de Filosofia define ontologia como "[...] o termo derivado da palavra grega que significa 'ser', mas usado desde o século XVII para denominar o ramo da metafísica que diz respeito àquilo que existe" (BLACKBURN; MARCONDES, 1997).

Na Ciência da Computação, ontologia tem um significado um pouco diferente, sendo utilizada para catalogar e classificar dados de forma a serem interpretados tanto por máquinas como por humanos como define Borst (1997) de uma forma bem simples e completa: "Uma ontologia é uma especificação formal e explícita de uma conceitualização compartilhada". Nessa definição, o termo "formal" significa legível para computadores; "especificação explícita" diz respeito a conceitos, propriedades, relações, funções, restrições, axiomas, explicitamente definidos; "compartilhado" quer dizer conhecimento consensual; e "conceitualização" se refere a um modelo abstrato de algum fenômeno do mundo real.

Dessa forma pode-se concluir que, para a Ciência da Computação, uma ontologia é a catalogação de um contexto do mundo real utilizando regras explícitas de propriedades, comportamento e relacionamentos em grupos de contextos similares que é chamado de domínio. Um domínio referencia certa aproximação entre as entidades que o compõe, de forma a relacionar grupos, e conseqüentemente seus membros, obtendo assim uma proximidade do comportamento real do contexto utilizado e limitando o contexto em domínios importantes para o seu uso, excluindo o que é supérfluo para o problema e obtendo um resultado mais apurado para um cenário trabalhado.

Esse comportamento é descrito por Platão em seu Mito da Caverna, onde ele define que as pessoas dentro da caverna, que se pode associar a aplicações, enxergam somente um cenário produzido pela realidade, as especificações dadas pelos programadores através dos algoritmos. A partir da Web Semântica, as aplicações começam a utilizar ontologias para reagir e começarem a "saltar" o muro da caverna, obtendo um resultado mais apurado da realidade.

Segundo Peréz (2002) uma ontologia é composta de cinco componentes: conceitos, relações, funções, axiomas e instâncias. Os Conceitos são uma abstração de qualquer coisa que componha o cenário a ser utilizado na ontologia, ou seja, as entidades que serão descritas, e que pode ser simples ou composto, abstrato ou concreto, reais ou fictícios, de forma que em um paradigma orientado a objetos podem ser assimilados a classes, e assim como as mesmas possui características de hierarquia: conceitos pais e conceitos filhos definindo um conceito como uma abstração mais ou menos específica de outro conceito.

Por Relações entendem-se as interações existentes entre os conceitos que podem ser descritas de forma que sempre referenciem um conceito com o outro, como por exemplo, um relógio é consertado por um relojoeiro, enquanto que as Funções são relações especiais que geram um resultado como, por exemplo: relojoeiro concerta relógio antigo, criando um relógio novo. Já os Axiomas são restrições criadas entre conceitos que possibilita uma seqüência de regras a serem seguidas para que aquele cenário seja válido como se pode observar nos exemplos a seguir: uma caneta tem que possuir tinta para poder escrever; um homem deve possuir pelo menos um ouvido para poder escutar.

As Instâncias representam os elementos descritos pelo conceito, relações, funções e axiomas e pode ser representada por um objeto de uma determinada classe em um paradigma de orientação a objetos.

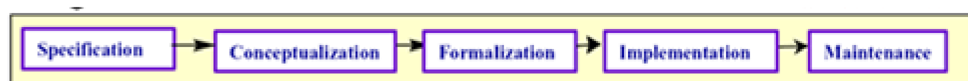
2.3 Metodologias para criação de ontologias

Metodologias têm sido desenvolvidas no intuito de sistematizar a construção e a manipulação de ontologias (LÓPEZ, 1999). Existem várias metodologias para construção e manipulação de ontologias, as que mais se destacam são a *Methontology* (FERNÁNDEZ-LÓPEZ, 1999), que utiliza conceitos do domínio e nas atividades de especificação, conceitualização, formalização, implementação e manutenção. A Metodologia Sensus (SWARTOUT, 1997) constrói ontologias a partir de outras ontologias, ligando sempre uma ontologia a alguma outra mais abrangente. Também existem outras metodologias a exemplo da metodologia de Gruninger e Fox (1995), o método de Uschold e King (1995), o método Cyc (REED e LENAT, 2002) e o método Kactus (BERNARAS, LARESGOITI e CORERA, 1996), das quais podem ser verificadas no estudo comparativo realizado por Silva et al. (2008).

Ao longo da década passada, foram desenvolvidas várias outras metodologias para construção e manipulação de ontologias, de forma que para cada tipo de ontologia necessitada existe uma metodologia associada.

A Fig. 2.3 ilustra a metodologia *Methontology*, na qual separa o processo de criação de uma ontologia em cinco etapas: especificação, modelo conceitual, formalização, implementação e manutenção.

Figura 2.3 – Etapas para Criação de uma Ontologia



Fonte: (CROCHO, 2005)

A primeira etapa é de Especificação, onde um documento é produzido em língua natural contendo o objetivo principal da ontologia, seus usuários, cenários, nível de formalismo e o escopo, e tem como resultado a geração de um documento de requisitos. Na segunda etapa, chamada de Geração, um documento final é gerado contendo as especificações de requisitos, sem redundâncias, omissões ou inconsistências, de forma que nessa etapa o problema seja resolvido e modelado utilizando somente os elementos, cenários e usuários delimitados na etapa de especificação.

Na terceira etapa, Formalização do modelo conceitual, serão aplicados todos os axiomas e funções lógicas, sendo nessa etapa o teste do documento de requisitos final, de forma a ser confrontado com o problema definido na etapa de especificação, depois de testado ocorre a implementação, já na quarta etapa, onde o modelo testado e aprovado é então descrito em uma Linguagem Formal de forma que possa ser interpretado pelo motor de inferência de uma máquina, sobrando agora só a etapa de Manutenção, onde toda a ontologia é periodicamente revista para garantir a sua atualidade e a sua consistência com os requisitos do problema.

Para a construção de ontologias, além de ser empregada uma metodologia, deve-se ser escolhida uma linguagem que permita formalizar a ontologia, de modo que possa ser interpretada em um motor de inferência, por exemplo. Dentre as diversas linguagens destaca-se a *Ressource Description Framework* (RDF) – uma linguagem desenvolvida pelo W3C Consortium (2004), e têm por objetivo a representação de conhecimento através da idéia de redes semânticas, além de ser uma linguagem que permite a representação de conceitos, axiomas de conceitos e relações binárias (LASSILA, 2001); a linguagem *Ontology Interchange Language* (OIL) - uma linguagem que combina primitivas de modelagem das linguagens baseadas em frames com a semântica formal e serviços de inferência da lógica

descritiva (ALMEIDA; BAX, 2003); e a linguagem *Web Ontology Language* (OWL) – a mais utilizada, onde define classes, propriedades e relacionamentos. A linguagem OWL foi projetada para ser utilizada na *web* e tem o seu uso em aplicações que precisam processar o conteúdo das informações, ao invés de só exibi-los (W3C, 2004), a OWL é uma recomendação da W3C e está substituindo a linguagem RDF.

A linguagem OWL define que uma classe ou entidade pode conter vários atributos, que podem guardar as características sobre essa classe sendo que um conjunto de atributos pode definir a estrutura sintática de uma classe, já uma classe possui relacionamentos que identificam a estrutura semântica da classe, definindo as regras cuja classe está submetida, sendo esses relacionamentos, na linguagem OWL, transitivos, simétricos ou funcionais.

Um relacionamento transitivo define que se existe uma relação entre (X, Y) e (Y, Z), pode-se inferir que existe o relacionamento entre (X, Z), já um relacionamento simétrico define que se existe a relação (X, Y) também existe a relação (Y, X), e um relacionamento funcional define que se existe as relações (X, Y) e (Y, Z) quer dizer que o elemento X é igual ao elemento Z.

Toda definição de um conceito, seja ele uma entidade, relacionamento ou propriedade, é definido em OWL por um *namespace* que é utilizado para referenciar o conceito por todo o código da ontologia OWL, e é escrito em sintaxe XML (acrônimo para *Extensible Markup Language*), de forma que suas propriedades, relacionamentos e entidades são descritos por *tags*, onde cada uma tem uma função na interpretação das informações, por exemplo, a tag SubClassOf define que o domínio referido é uma subclasse do domínio explicitado.

A Fig. 2.4 ilustra uma ontologia representada na linguagem OWL.

Figura 2.4 – Descrição de uma Ontologia Usando a Linguagem OWL


```
<owl2xml:Ontology xmlns="http://ontologiaDominio.owl#"
     xml:base="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
     xmlns:ontologiDominio="http://ontologiaDominio.owl#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     owl2xml:URI="http://ontologiaDominio.owl">
    <owl2xml:SubClassOf>
        <owl2xml:Class owl2xml:URI="&ontologiDominio;Assunto"/>
        <owl2xml:Class owl2xml:URI="&ontologiDominio;Disciplina"/>
    </owl2xml:SubClassOf>
    <owl2xml:SubClassOf>
        <owl2xml:Class owl2xml:URI="&ontologiDominio;Curso"/>
        <owl2xml:Class owl2xml:URI="&owl;Thing"/>
    </owl2xml:SubClassOf>
    <owl2xml:SubClassOf>
        <owl2xml:Class owl2xml:URI="&ontologiDominio;Disciplina"/>
        <owl2xml:Class owl2xml:URI="&ontologiDominio;Curso"/>
    </owl2xml:SubClassOf>
    <owl2xml:SubClassOf>
        <owl2xml:Class owl2xml:URI="&ontologiDominio;Professor"/>
```


Fonte: Elaborada pelo autor

2.4 Resumo do capítulo

Neste capítulo foi descrita a evolução da Web, desde a Web 1.0 a 3.0, e a esta última evolução se denomina Web Semântica. Também apresentou as formas de trabalhar com a Web Semântical e como a Web 3.0 está mudando os rumos das aplicações desenvolvidas na atualidade.

Foram apresentados os conceitos de ontologias, a base da Web Semântica, de forma a entender todos os seus conceitos e formas para a construção de ontologias que sejam eficazes em seu papel de estruturadora semântica, de forma a não haver inconscistência semântica na criação de uma ontologia; também foi destacada a existência da linguagem OWL para criação de ontologias em uma linguagem formal, que possa ser interpretada pelas máquinas.

No próximo capítulo serão explicados alguns padrões de desenvolvimento utilizados neste trabalho.

3 Padrões de desenvolvimento

Neste capítulo é explicada um pouco sobre os padrões de desenvolvimentos utilizados neste trabalho. Na seção 3.1 são mostradas as técnicas de MDA utilizadas. Na seção 3.2 são exibidos os padrões de projeto utilizados no desenvolvimento da arquitetura proposta. Na seção 3.3 são exemplificadas algumas linguagens de desenvolvimento WEB e finalizamos o capítulo com um breve resumo na seção 3.4.

3.1 MDA

A arquitetura aqui descrita utiliza a Web Semântica para coletar informações do usuário, de forma a obter o mais preciso conteúdo, e então modelar todo um sistema baseado em ontologia e a partir daí, uma estrutura contendo geradores de código, baseado em técnicas denominadas *Model Driven Architecture* (MDA), no qual o produto de *softwar*e gerado é capaz de realizar ações de inserção, exclusão, edição, busca e exibição de dados.

A MDA é uma visão de como um *softwar*e pode ser desenvolvido, colocando a modelagem no centro do processo de desenvolvimento e a partir de um modelo abstrato do sistema gerar um modelo mais concreto, através deste processo de refinamento dos modelos é possível gerar o código fonte a ser produzido, sendo este considerado como a mais concreta representação do sistema de *softwar*e. A chave para esse processo está na máxima automatização para cada etapa da geração (OMG, 2000).

Para este trabalho, são utilizadas três etapas da MDA, que foram descritas por Watson (2003). A primeira etapa consiste no Modelo Independente de Plataforma (PIM, acrônimo do inglês *Plataform Independent Model*), que consiste em uma especificação completa do sistema e que deve ser de altíssimo nível e completamente independente de plataforma. Neste trabalho é utilizada a descrição dada pelo usuário, e inferida pelo motor semântico, como PIM do sistema a ser gerado. Na segunda etapa é gerado um Modelo de Plataforma (PM, do inglês *Plataform Model*), que nada mais é do que uma modelagem específica para o PIM, contendo os dados necessários para que o sistema seja construído na

plataforma escolhida, para isso utilizada uma ontologia para descrever o PM do sistema gerado, e todo PM gerado será feito utilizando o paradigma de orientação a objetos.

A terceira etapa, denominada transformação em código, é onde todo o sistema gerado é construído a partir do PM, de forma que toda a extração de código seja feita de forma automática e sempre seguindo todos os dados contidos no PM do sistema a ser gerado. Neste trabalho um conjunto de classes realiza toda a etapa de transformação do PM em código, refinando-o quando necessário, de forma a automatizar o máximo possível do processo.

3.2 Padrões de projeto

O desenvolvimento de *softwar*e tem inicio na etapa de análise de requisitos e modelagem do sistema sendo essa etapa é crucial para o desenvolvimento de um sistema eficiente, com qualidade e que atenda as necessidades do cliente.

Devido ao seu papel crucial no desenvolvimento, a etapa de modelagem do sistema se torna uma tarefa difícil, pois é necessário pensar em todo o funcionamento do sistema, modelar esse funcionamento de uma forma que toda a equipe entenda e que no final, funcione perfeitamente.

Durante essa etapa, alguns desenvolvedores encontram problemas recorrentes de implementação que necessitam ser modelados, de forma que muitos desses problemas aparecem em vários tipos de modelagens e que na maioria dos casos podem ser resolvidos da mesma maneira, utilizando as mesmas técnicas de modelagem e/ou programação.

Na área da engenharia de *softwar*e, esses problemas foram resolvidos através de soluções compartilhadas e genéricas, que podem ser aplicadas a qualquer projeto, independente de sua forma, desde que se encaixe nos pré-requisitos de uso da solução, e foram denominadas padrões de projeto, do inglês *Designe Patterns.*

O livro "*Design Patterns*", de Erich Gamma et al. (1994), conhecidos como a gangue dos quatro (GoF – *Gang of Four*), descreve 23 padrões de projeto, ao mesmo tempo que eles definem padrões de projeto como "descrições de objetos que se comunicam e classes que são adaptadas para resolver um problema genérico de *design* em um contexto específico".

Neste trabalho são utilizados alguns padrões em várias das camadas da arquitetura do sistema sempre resolvendo os problemas de forma eficiente e eficaz, deixando toda a

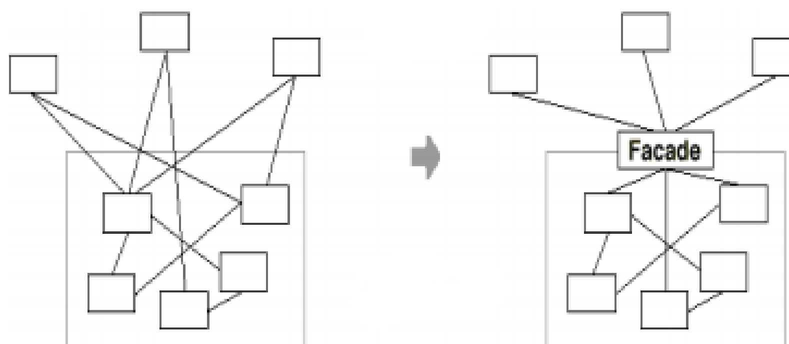
arquitetura mais desacoplada e de fácil manutenção, alteração e compreensão. Na arquitetura proposta foram utilizados os seguintes padrões: *Façade, Factory,* DAO, DAL e VO.

O padrão *Façade*, também chamado de Fachada, é utilizado para disponibilizar uma interface que permite acesso a outras classes, sem que as mesmas possam ser acessadas diretamente, como define bem Junior (2004):

> Existem circunstâncias onde é necessário utilizar diversas classes diferentes para que uma tarefa possa ser completada, caracterizando uma situação onde uma classe cliente necessita utilizar objetos de um conjunto específico de classes utilitárias que, em conjunto, compõem um subsistema particular ou que representam o acesso a diversos subsistemas distintos.

Como percebido, o padrão *Façade* é bastante utilizado como uma porta de entrada para um conjunto de classes impossibilitando o acesso direto às classes internas desse conjunto, restringindo e validando o acesso somente por uma via, implicando numa maior segurança e integridade de dados e uma independência de funcionalidades daquele conjunto de classes, que podem agora ser distribuídas em pacotes fechados e acessados somente pela fachada, assim como ilustra a Fig. 3.1.

Figura 3.1 – Padrão Fachada



Fonte: Elaborada pelo autor

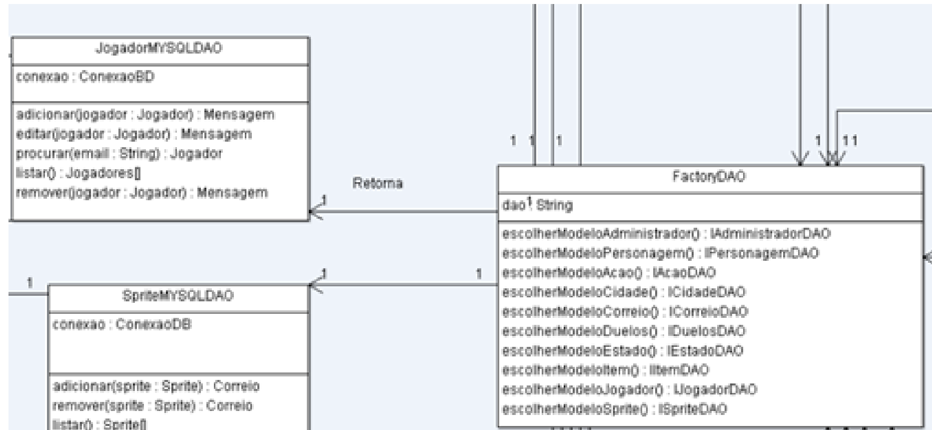
Já o padrão *Factory,* também conhecido como Fábrica, oferece uma forma de instanciar objetos, sem a necessidade de saber detalhes sobre a criação dos mesmos, como bem define Destro (2004):

> O padrão Factory é útil para se construir objetos individuais, para um propósito específico, sem que a construção requeira conhecimento das classes específicas sendo instanciadas.

O padrão *Factory* é bastante utilizado em casos que existem mais de um objeto que realize uma determinada ação, e somente em tempo de execução é possível saber qual objeto será instanciado para tal, com isto, a *Factory* tem o papel de perceber qual objeto melhor se encaixa no cenário em que se encontra e instanciá-lo, sem oferecer os detalhes disto para a classe que a invoca, assim como ilustra a Fig 3.2, onde uma factory, FactoryDAO é utilizada

para instanciar as classes jogadorMySQLDAo ou SpriteMySQLDAO, de acordo com o que for determinado em tempo de execução pelas classes que a invocam.

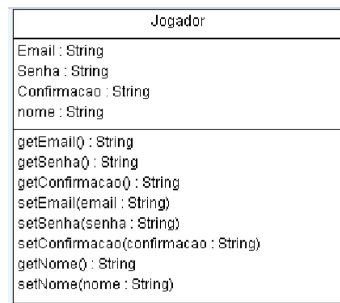
Figura 3.2 – Exemplo da Utilização do Padrão *Factory*



Fonte: Elaborada pelo autor

Um problema comum na estruturação de projetos com diversas camadas é o transporte dos dados entre essas camadas sem a necessidade de ter que conhecer todos os dados das entidades base desta aplicação, e para resolver este problema foi utilizado o padrão *Value Object*, também denominado VO que nada mais é do que um simples objeto que funcionará como um repositório de dados na aplicação, contendo somente os dados referentes ao modelo de domínio ao qual está relacionado e sendo utilizado para encapsular informações durante a troca de dados entre as camadas do sistema, assim como ilustra a Fig 3.3 que exibe um VO contendo os dados que juntos representam um Jogador.

Figura 3.3 – Exemplo de Utilização do Padrão VO



Fonte: Elaborada pelo autor

Outro problema bastante comum nas aplicações desenvolvidas por projetistas de *software* se encontra nos métodos para manter, acessar e persistir os dados da aplicação, de forma que esses dados possam ser utilizados por todas as camadas do sistema sem ocorrer inconsistência nos dados a serem armazenados ou recuperados.

Esse problema pode ser solucionado utilizando-se uma camada de implementação exclusiva para isso, denominada Camada de Acesso a Dados (DAL, acrônimo do inglês *Data Access Layer*). A camada DAL pode conter várias classes que de fato irão realizar todas as ações sob as formas de persistência de dados, porém ela deve funcionar totalmente independente das outras camadas, de forma que possa ser destacada e/ou trocada por outras, sem a necessidade de alteração em nenhuma outra camada da arquitetura, isso possibilita uma independência de modelagem enorme, o que auxilia no desenvolvimento de projetos por várias equipes e na manutenção do *softwar*e, já que ela pode ser trocada por outra camada DAL que utilize tecnologias mais novas a qualquer momento, por exemplo.

Dentro da camada DAL podem existir várias classes que implementam padrões de processo, como fachadas, *factories* e objetos de acesso a dados, também conhecido como DAO, acrônimo do inglês *Data Access Object.* O padrão DAO realiza todas as operações diretamente na base de dados da aplicação, sendo ele o responsável por cuidar de todo o acesso e sincronização dos dados, além de ter que garantir a consistência dos dados obtidos ou repassados para outras camadas. Para cada fonte de dados da aplicação uma classe DAO é necessária de forma que cada classe DAO seja única em sua função o que proporciona uma fácil manutenção e atualização por parte dos desenvolvedores. O DAO pode ser utilizado em conjunto com outros padrões, obtendo assim uma forma de acesso a dados reusáveis por vários projetos independente da lógica de negócio.

3.3 Linguagens para desenvolvimento WEB

A WEB virou uma plataforma rica para desenvolvimento de aplicações, possuindo suas próprias regras e maneiras de desenvolver soluções e sistemas integrados que evolui muito mais de pressa que qualquer outra plataforma, forçando seus desenvolvedores a criar idéias não-comuns para seus aplicativos, tudo forçado por um publico alvo que está evoluindo tão ou mais rapidamente do que a própria plataforma.

Para que os desenvolvedores criem todas essas aplicações e soluções, várias tecnologias e ferramentas estão disponíveis no mercado e cabe a cada desenvolvedor escolher qual utilizar através de critérios como afinidade, suporte a algumas funções específicas, integração com outros sistemas, facilidade de desenvolvimento e até mesmo disponibilização de documentação sobre a ferramenta ou tecnologia.

No desenvolvimento WEB destacam-se algumas linguagens de programação, plataformas e ferramentas como as linguagens PHP, JAVA e C# e as plataformas JAVA e ASP.NET. Cada uma delas possui vantagens e desvantagens bem definidas e realizam praticamente as mesmas funções, sendo quase que, na maioria das vezes, escolha do próprio desenvolvedor por afinidade ou conhecimento qual linguagem e plataforma utilizar.

A linguagem PHP é uma linguagem de programação de ampla utilização, interpretada, que é especialmente interessante para desenvolvimento para a Web e pode ser mesclada dentro do código HTML, vem se destacando no desenvolvimento de aplicações WEB desde que foi criada, em 1995 por Rasmus Lerdof, como uma linguagem de script para desenvolvimento de páginas na internet Ao chegar a sua terceira versão começa a ganhar funções de orientação a objetos e hoje, na sua quinta versão, continua uma das linguagens mais utilizadas por muitas aplicações online, devido a vários fatores como por ser uma linguagem dinamicamente tipada, possuir uma sintaxe muito similar a linguagem C e possuir suporte a programação Orientada a Objetos. (LERDOF, 2000).

Já a plataforma JAVA possui uma gama de *framework*s para trabalhar na web, todos eles funcionando em cima da J2EE, a plataforma para programação de servidores, e muitos deles são implementados em cima de contextos servlets, componentes do lado servidor que abstraem as implementações do próprio servidor e geram dados para as camadas de visualização do sistema na WEB. A plataforma JAVA para web traz todo o poder e robustez da linguagem JAVA aplicada à dinamicidade das aplicações WEB e é hoje em dia o conjunto de linguagem/plataforma mais utilizada no desenvolvimento de aplicações WEB.

A Plataforma .NET é uma contrapartida da Microsoft para o desenvolvimento de aplicações em todos os níveis, e o ASP.NET é a plataforma para desenvolvimento de aplicações WEB. O .NET *Framework* surge no inicio da década de 2000 e vem obtendo uma grande adesão tanto no lado comercial como em pesquisas, devido ao grande apoio e investimento da própria Microsoft em ferramentas de ponta que auxiliam na produção e desenvolvimento das aplicações. Diferente das outras plataformas, o ASP.NET oferece a possibilidade de programação em diversas linguagens, o que vem permite que o desenvolvedor possa usar todo o seu conhecimento adquirido em qualquer outra linguagem para o desenvolvimento web, sem a necessidade de aprender uma nova linguagem para tal.

3.4 Resumo do capítulo

Neste capítulo foram descritas as tecnologias utilizadas na implementação deste trabalho, exibindo desde as que foram utilizadas para estruturar o desenvolvimento da modelagem, passando pelas que são utilizadas para a propria criação da arquitetura do sistema e chegando até as linguagens para implementação do projeto.

Foram apresentadas as técnicas de MDA que foram utilizadas para estruturar o desenvolvimento dos geradores e organizar a ontologia do sistema, todas os padrões utilizados para arquiteturar a implementação, utilizando o máximo de independencia entre camadas possível e exibindo as possiveis escolhas de linguagens e plataformas de desenvolvimentos que poderiam ser utilizadas.

No próximo capítulo será explicada a arquitetura proposta que utiliza como base de seu funcionamento a Web Semântica definida por uma ontologia.

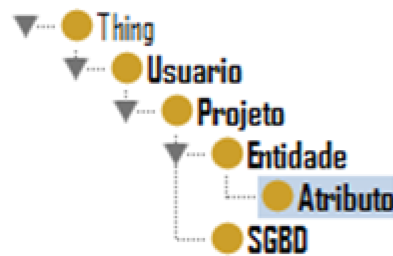
4 Arquitetura do sistema proposto

Neste capítulo é explicada a arquitetura do sistema proposto. Na seção 4.1 é mostrada como a ontologia do sistema foi criada e na seção 4.2 são exibidas toda a arquitetura do sistema gerador, contendo seus padrões, classes e relacionamentos. Na seção 4.3 é explanado o motor semântico do sistema, denominado Wizzard, e finalizamos o capítulo com um breve resumo na seção 4.4.

4.1 Criação da ontologia

Na definição de uma ontologia utilizada pelo MDA, um PM deve ser específico para cada sistema gerado de forma a conter todos os conceitos que resolvem o problema daquele sistema gerando um código específico, ou seja, baseado em uma ontologia específica, por tanto para cada sistema a ser gerado haverá um PM criado pela arquitetura proposta. Esse PM deve ser desenvolvido de forma que seus conceitos possam ser interpretados por todo o sistema, gerando assim, uma plataforma de colaboração entre os projetos criados e permitindo que o motor de inferência auxilie o usuário na solução de seu problema, então o PM do sistema a ser gerado deve estender de um PM genérico e este deve ser descrito de forma a permitir que o motor de inferência atue sobre ele, então na arquitetura proposta foi desenvolvida uma ontologia de domínio para servir como PM genérico, essa ontologia foi gerada através da metodologia de criação de ontologias *Methontology* (FERNÁNDEZ-LÓPEZ, 1999) e ao final da etapa de implementação, obteve-se o modelo exibido na Fig. 4.1.

Figura 4.1 – Ontologia de Domínio



Fonte: Elaborada pelo autor

A ontologia de domínio genérica possui conceitos, e cada conceito possui relacionamentos, axiomas e atributos, onde cada atributo é um conceito descritor que ajuda a explicar os outros conceitos, permitindo a flexibilidade da ontologia para vários casos, independente da aplicação.

Na arquitetura proposta é utilizada uma ontologia de domínio com conceitos de usuário, projetos, SGBD e entidades, propiciando uma diferenciação semântica na construção das aplicações, fazendo uso de definições geralmente utilizadas na construção de *softwares*, e cada conceito possui propriedades ontológicas que o identifica. Abaixo são descritos os conceitos da ontologia:

- **Usuário:** um usuário é o proprietário do projeto, é quem descreve e mantém o projeto além de deter o poder sobre o sistema gerado.
    - **Propriedades:** foram definidos como: data de nascimento, home page, país, e-mail, sexo, senha e nome.
    - **Relacionamentos:** um usuário possui o relacionamento *temProjetos***,** onde indica se um usuário possui um ou mais projetos.
- **Projeto:** um projeto é uma idéia modelada pelo sistema que será solucionada através de um sistema de informação. O projeto possui propriedades que definem sua geração:
    - **Propriedades:** foram definidos como: id, nome, descrição, data de criação, linguagem.
    - **Relacionamentos:** um projeto possui os relacionamentos *temUsuário* (onde um projeto pertence a um único usuário), *temEntidades* (onde um projeto pode conter uma ou várias entidades), *temSGBD* (onde um projeto pode conter um ou vários SGBD's).

- **SGBD:** é o conceito que permite a modelagem de sistemas gerenciadores de banco de dados, no qual representa a descrição de qual SGBD o sistema gerado utilizará. O SGBD possui propriedades de como deverá ser gerado.
  - **Propriedades:** são eles: id, nome, host, senha, usuário, database**.**
- **Entidade:** representa um conceito relacionado ao problema declarado pelo usuário, na qual é a base estrutural do problema, e que pode ser um ator, uma ação ou informações do sistema. Uma entidade possui as seguintes propriedades:
  - **Propriedades:** são eles: id, nome e descrição.
  - **Relacionamentos:** tem como relacionamento *temAtributos*, permitindo que uma entidade possua um ou vários atributos.
  - **Axiomas:** conforme discutido no Capítulo 2 seção 2.2, um axioma é uma regra utilizada para manter uma sequência lógica de regras que precisam ser seguidas para que o conceito seja validado, ou seja, para que o motor de inferência possa identificar e exibir ao usuário quais entidades são semelhantes à que ele esteja descrevendo é necessário seguir os axiomas definidos, auxiliando na construção do seu PM. Foram definidos dois axiomas para entidades:
    - **semelhança:** uma entidade é considerada semelhante à outra se elas possuírem nomes complementares, isto é, se os nomes forem iguais ou o nome de uma entidade faça parte do nome de outra e se possuírem pelo menos uma quantidade mínima de seus atributos iguais. Caso uma das entidades não possua atributos, somente a assertiva do nome será válida.
    - **igualdade:** uma entidade é considerada igual à outra, se elas possuírem a mesma quantidade de atributos e esses atributos forem os mesmos.

Vale ressaltar que para o conceito Entidade da ontologia existirá um conjunto de atributos que a identifica, representando uma descrição mais apurada de uma entidade, definindo a própria entidade, suas ações e suas relações com outras entidades. Um atributo possui as seguintes propriedades:

- **Atributos:** possui as seguintes propriedades, relacionamentos e axiomas, sendo este último uma maneira de garantir sua integridade no motor semântico.
  - **Propriedades:**

- **id:** identifica o atributo, e não deve existir mais de um atributo com o mesmo id.
- **tipoExibição:** define como esse atributo será exibido no sistema gerado. Pode ter os seguintes valores:
  - **Imagem:** referencia uma imagem, e será exibido como tal.
  - **Seleção**: pode ser um conjunto de valores, dos quais deverá existir pelo menos uma escolha.
  - **Texto:** permite a inclusão de um texto com várias linhas.
  - **Valor:** é um campo simples, contendo um valor que pode ser representado em apenas uma linha.
- **chavePrimária:** identifica se o atributo é o campo que referenciará a entidade. Para cada entidade, só poderá existir um atributo definido como chave primária.
- **multivalorado:** identifica se o atributo pode conter mais de um valor. Um campo multivalorado nunca poderá ser a chave primária da entidade.
- **nulo:** identifica se o atributo pode ser vazio ou nulo. Um atributo nulo nunca poderá ser a chave primária da entidade.
- **tamanho:** limita a quantidade de caracteres suportados pelo atributo.
- **tipoDado:** identifica o tipo de dado primitivo do atributo. Pode ser *Date*, *Double*, *Integer* e *String*.
- **descrição:** descreve o atributo.
- **dataCriação:** data de criação do atributo fornecida pelo sistema.
- **nome:** nome do atributo fornecido pelo usuário.

o **Relacionamentos:**

- **tipoDado:** um Atributo pode referenciar uma entidade, em seu tipo de dado. Caso isso ocorra, ele passará a apontar para o id da entidade selecionada.

o **Axiomas:**

- **semelhança:** definido pelo método `ehEntidadeSemelhante( )`, no qual informa se um atributo é considerado semelhante a outro, e em sua propriedade *nome* deverá conter parte do nome do atributo ao qual é semelhante, ou seja, os nomes

devem ser iguais ou o nome de um deve ser parte do nome do outro, ou ainda se possuírem pelo menos duas propriedades iguais, exceto nome, descrição e dataCriação, uma vez que estas não representam nenhum valor semântico para a construção da aplicação. Caso um dos atributos não possua propriedades preenchidas, somente a assertiva do nome será válida.

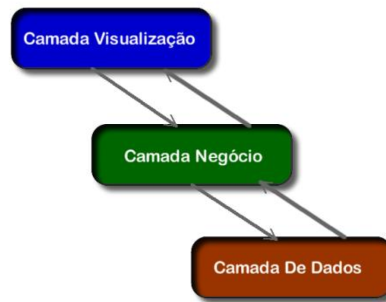
- **Igualdade:** para a verificação de uma igualdade deve-se utilizar o método `ehAtributoIgual( )`, onde um atributo é considerado igual a outro se ele possuir todas as propriedades definidas. Assim como no axioma de semelhança também não são consideradas as propriedades de nome, descrição e dataCriação pelos mesmos motivos.

4.2 Descrição da arquitetura geradora

Para o desenvolvimento da arquitetura do *framework* proposto foi utilizada a linguagem de programação C#, por meio da plataforma Microsoft .Net *Framework* 4.0, e para realizar a persistência dos dados foi escolhido o SGBD MySQL. A codificação do sistema gerado está escrita na linguagem PHP utilizando também o MySQL como SGBD.

A arquitetura proposta está implementada em três camadas: a camada de dados, a camada de negócios e a camada de visualização, nas quais podem ser visualizadas na Fig. 4.2. Cada camada possui suas próprias classes e através da comunicação entre elas obtém-se todo o funcionamento do sistema. Este capítulo descreve as camadas de negócio e de dados, uma vez que a camada de visualização é uma especialização deste trabalho e será exemplificada no estudo de caso descrito no Capítulo 5.

Figura 4.2 – Camadas da Arquitetura

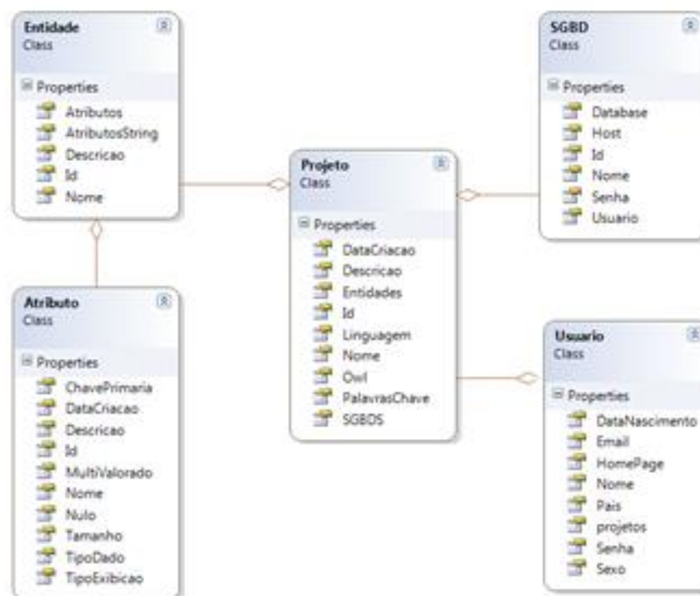


Fonte: Elaborada pelo autor

4.2.1 Camada de dados

A camada de dados é responsável por modelar à ontologia do sistema e tratar de toda e qualquer forma de persistência de dados que venha a ser utilizada. Esta camada é composta de dois *namespaces*: DataSet e VO. O namespace VO (do inglês, *Value Object*) representa os conceitos presente na ontologia do sistema onde estão contidas todas as classes de dados básicas e seus relacionamentos como demonstrado na Fig. 4.3, sob a forma do diagrama de classes da ontologia de domínio, e todo o sistema utiliza essas classes básicas para realizar as operações necessárias sempre seguindo as regras delimitadas na ontologia.

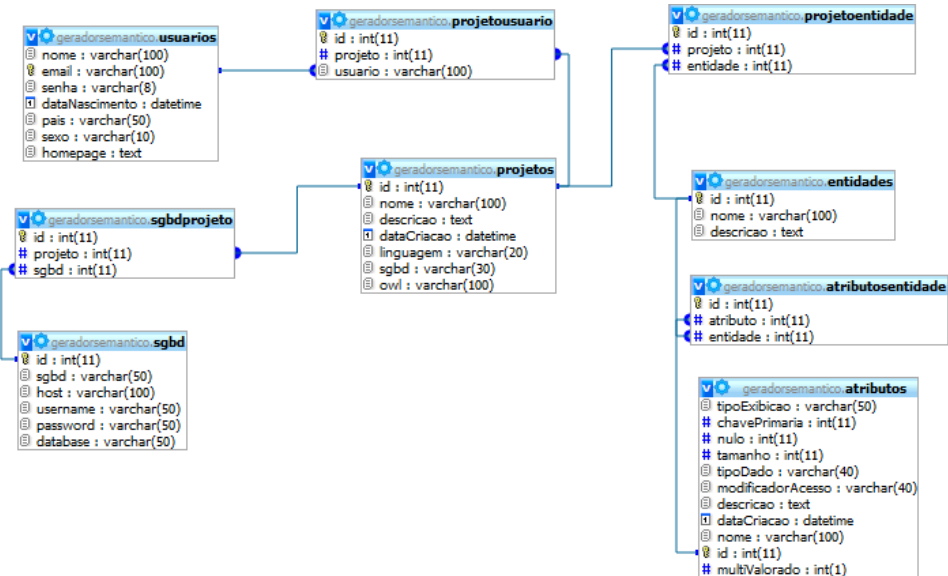
Figura 4.3 – Classes Básicas Contidas no namespace VO



Fonte: Elaborada pelo autor

O sistema utiliza um banco de dados para realizar a persistência da ontologia, de forma a manter todos os relacionamentos e definições dadas pelo usuário além de permitir que o motor semântico possa inferir suas decisões, esse banco de dados é ilustrado na Fig. 4.4 a partir do diagrama entidade-relacionamento do sistema.

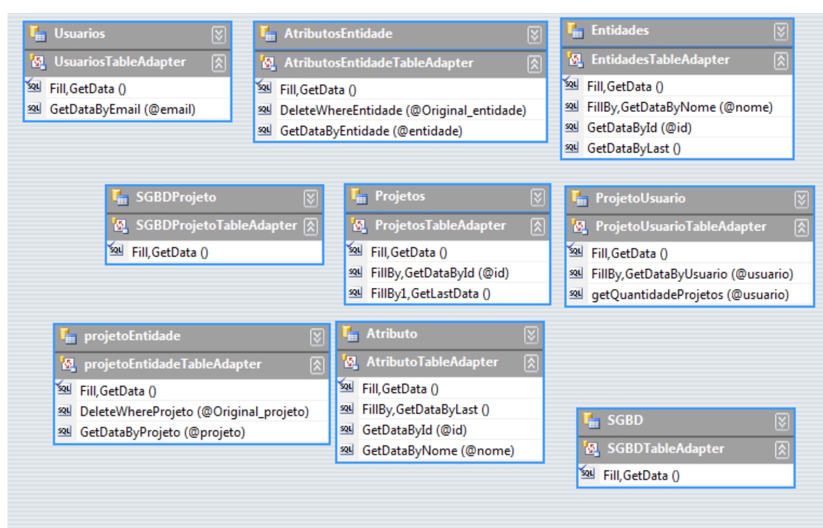
Figura 4.4 – Modelo Relacional do Sistema



Fonte: Elaborada pelo autor

Para a utilização do banco de dados, foi realizada a persistência e operações relacionadas à base de dados do sistema, para tanto foi preciso implementar uma camada de acesso aos dados que é representada pelo namespace DataSet que contém as estruturas necessárias para tais ações. Todo DataSet, ilustrado na Fig. 4.5, utiliza a estrutura da própria representação DataSet da plataforma Microsoft .NET *Framework*, que consiste em um mapeamento de todas tabelas e colunas do banco de dados em um modelo lógico, e a partir dessa estrutura é possível realizar consultas com a linguagem SQL, e obter os resultados em um formato passível de manipulação em tempo de execução utilizando a linguagem C#.

Figura 4.5 – Estrutura DataSet do Sistema



Fonte: Elaborada pelo autor

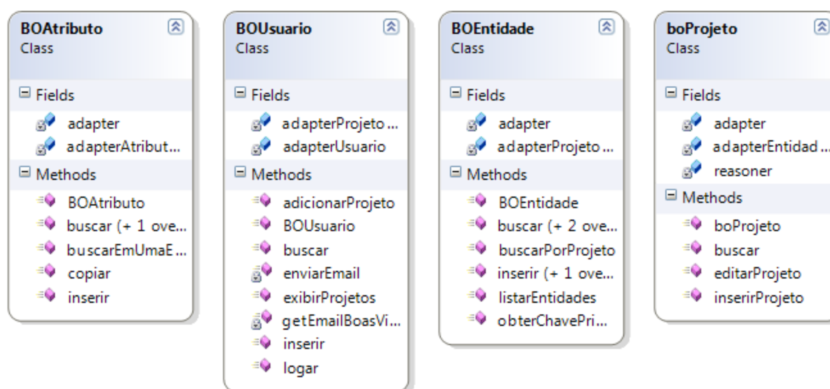
4.2.2 Camada de negócios

É na camada de negócios que encontra os motores do sistema onde estão contidas todas as classes de geradores, que transformam o PM do sistema a ser gerado em código e o motor semântico, que auxiliará na construção do PM. Essa camada está dividida em quatro principais namespaces: Arquivos, Grupo de namespaces Geradores, Entidades e Wizard, cada uma delas contendo um conjunto de classes responsáveis por realizar uma ação específica que ao atuar em conjunto representam toda a lógica de negócios do sistema.

4.2.2.1 Entidades

Na namespace Entidades estão presentes as classes que manipularão a ontologia e a estrutura DataSet e, como ilustrado na Fig. 4.6, contém quatro classes, cada uma responsável por manipular os conceito da ontologia: `BOAtributo`, `BOUsuário`, `BOEntidade`, `BOProjeto`.

Figura 4.6 – Classes Contida na namespace Entidades da Camada de Negócios



Fonte: Elaborada pelo autor

A classe `BOUsuário` é responsável por todas as manipulações que ocorram no conceito Usuário e é através dela que são inseridos novos usuário ao sistema mediante uma verificação na qual utiliza o campo chave para constatar se um usuário está cadastrado no sistema, ou seja, é nesta verificação que determina se um usuário é ou não único, neste caso por meio da informação que representa seu *login*, ou seja, o e-mail cadastrado para o usuário.

No sistema pode-se associar um projeto a um usuário, através do método `adicionarProjeto( )`, além de ser possível realizar uma buscar por um usuário. A classe `BOUsuário` se comunica diretamente com o DataSet através de dois objetos: `adapterProjeto` e `adapterUsuário`. Neles estão contidos todos os métodos necessários para manipulação do usuário diretamente na base de dados.

A classe `BOProjeto` contém os métodos de inserção, busca, edição, remoção e listar projetos dentro do sistema, sendo esses métodos todos executados por *queries* dentro dos DataSets e invocados pelo `BOProjeto`, todos os métodos são atômicos, isto é, realiza somente a ação referida e uma validação simples, a exemplo do método `editar( )`, ilustrado na Fig. 4.7, que contém uma validação na qual verifica se o projeto editado contém entidades não consistentes ou que não estejam presentes no sistema, estas entidades são criadas e depois são associados ao projeto.

Figura 4.7 – Método `editar( )` da Classe `BOProjeto`

```
adapter.Update(projeto.Nome, projeto.Descricao,
    projeto.DataCriacao, projeto.Linguagem, "",
    projeto.Owl, projeto.Id);

adapterEntidades.DeleteWhereProjeto(projeto.Id);

BO.entidade.BOEntidade boEntidades = new BO.entidade.BOEntidade();

foreach (VO.Entidade entidade in projeto.Entidades)
{
    if (entidade.Id == -1 || reasoner.isEntidadeVerificada(entidade))
        entidade.Id = boEntidades.inserir(entidade, projeto.Id);
    else
        boEntidades.inserir(projeto.Id, entidade.Id);
}

return projeto;
```

Fonte: Elaborada pelo autor

A classe `BOEntidade` manipula o conceito de entidades, presente na ontologia e é através dela que se pode buscar entidades no banco de dados, inserir entidades e exibir a chave primária de uma dada entidade, além do método `buscar( )`, ilustrado na Fig. 4.8, que recebe como parâmetro um `id` de uma entidade e retorna todos os seus dados, incluindo os atributos que pertencem àquela entidade.

Figura 4.8 – Método `buscarEntidade( )`

```
BO.entidade.BOAtributo boAtributo = new BOAtributo();

DataTable tabela = adapter.GetDataById(id);

VO.Entidade entidade = new VO.Entidade();
entidade.Descricao = tabela.Rows[0].Field<String>("descricao");
entidade.Id = tabela.Rows[0].Field<int>("id");
entidade.Nome = tabela.Rows[0].Field<String>("nome");

entidade.Atributos = boAtributo.buscarEmUmaEntidade(entidade.Id);

entidade.AtributosString = "";
if (entidade.Atributos != null)
    foreach (VO.Atributo atributo in entidade.Atributos)
        entidade.AtributosString += atributo.Nome + ";";
return entidade;
```

Fonte: Elaborada pelo autor

A classe `BOAtributo` contém todos os métodos necessários para manipular um conceito Atributo do sistema, contendo métodos para inserir e buscar um atributo no banco de dados, buscar um atributo dentro de uma data entidade e criar uma cópia do atributo, como ilustrado na Fig. 4.9, onde é possível observar a geração de que uma cópia de um dado atributo, dessa forma preservando o atributo original em algumas operações.

Figura 4.9 – Método `copiarAtributo( )`

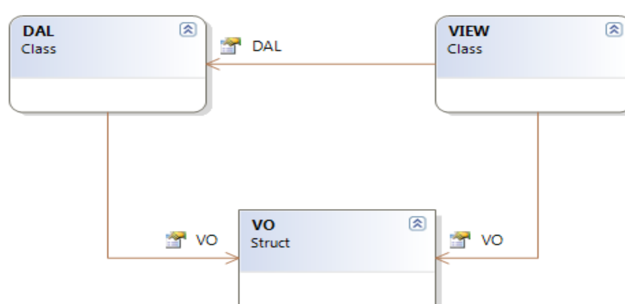
```
VO.Atributo novoAtributo = new VO.Atributo();
novoAtributo.ChavePrimaria = atributo.ChavePrimaria;
novoAtributo.DataCriacao = atributo.DataCriacao;
novoAtributo.Descricao = atributo.Descricao;
novoAtributo.Id = atributo.Id;
novoAtributo.MultiValorado = atributo.MultiValorado;
novoAtributo.Nome = atributo.Nome;
novoAtributo.Nulo = atributo.Nulo;
novoAtributo.Tamanho = atributo.Tamanho;
novoAtributo.TipoDado = atributo.TipoDado;
novoAtributo.TipoExibicao = atributo.TipoExibicao;
return novoAtributo;
```

Fonte: Elaborada pelo autor

4.2.2.2 Geradores

No grupo de namespaces Geradores estão contidos em todas as classes responsáveis pela geração de código que trabalham em conjunto com as classes da namespace Entidades para obter o sistema gerado que segue um padrão de arquitetura em duas camadas: *Data Acces Layer* (DAL) e *View*, exibidas na Fig. 4.10. Na camada DAL estão presentes todas as classes e informações relativas à persistência dos dados e na camada *View* estão todas as interfaces que servirão para entrada e saída de dados na arquitetura gerada, essas duas camadas se comunicam através das classes que implementa o padrão *Value Object* (VO) e toda a arquitetura é gerada pelas classes pertencentes ao grupo de namespaces Geradores baseado nas definições dadas pelo usuário, assim como implementa cinco funções básicas: remoção, inserção, busca, edição e listagem de dados.

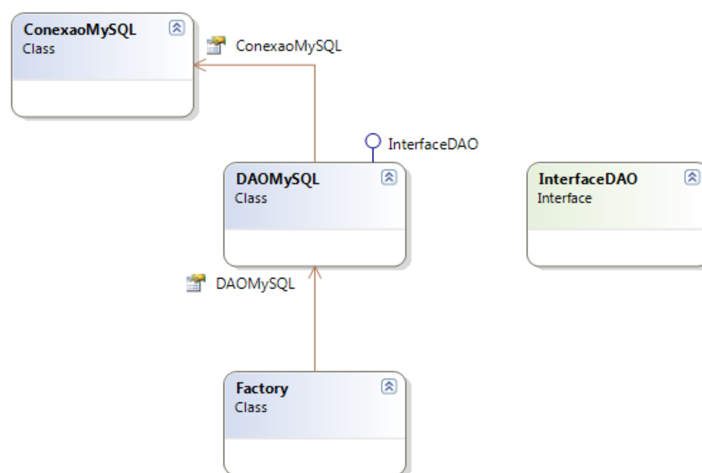
Figura 4.10 – Arquitetura do Sistema Gerado



Fonte: Elaborada pelo autor

A camada DAL está dividida em três conjuntos de classes: classe de conexão, classe DAO e a *Factory*, também é definida uma interface DAO, o que permite uma facilidade na implementação de diversas classes DAO, como ilustra a Fig. 4.11.

Figura 4.11 – Camada DAL da Arquitetura Gerada

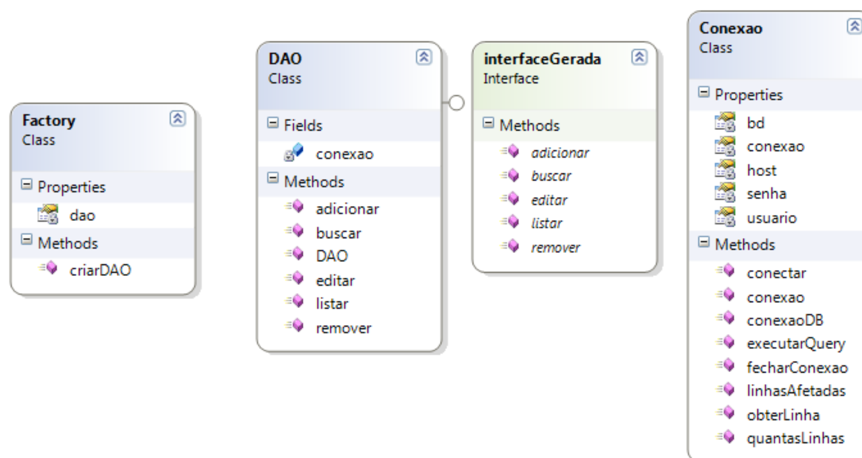


Fonte: Elaborada pelo autor

Na classe `ConexãoMySQL` estão presentes todas as informações necessárias para realizar a conexão com o banco de dados e todos os métodos para que se possa manipulá-lo, sendo necessário existir uma classe conexão para cada SGBD utilizado pelo sistema gerado. A interface é utilizada para criar um padrão de classes DAO, de forma a obedecer as regras delimitadas na arquitetura e assim facilitar a criação de novas classes DAO para outros SGBDs, também sendo necessário que os métodos `adicionar( )`, `editar( )`, `remover( )`, `buscar( )` e `listar( )` sejam implementados em cada classe DAO do sistema gerado. Cada classe DAO contém uma implementação da interface, de modo que a classe DAO é responsável por gerar as *queries* de manipulação do banco de dados e para cada conceito Entidade é necessário a criação de uma classe DAO, e para cada SGBD escolhido um conjunto de classes DAO é criado, por meio da classe de conexão do SGBD escolhido para poder executar as *queries* geradas.

A classe *Factory* é responsável por oferecer independência entre a camada DAL e a camada *View*, sendo nela implementada todas as instâncias das classes DAO necessárias e repassadas para a camada *View*, de forma que a camada *View* só informe qual SGBD quer utilizar e então a *Factory* instancia todas as classes DAO do SGBD escolhido e retorna à *View*, sem a necessidade de saber qual DAO foi instanciado. A Fig. 4.12 exibe claramente toda a estrutura gerada contendo uma *Factory*, uma interface, uma classe DAO e uma conexão.

Figura 4.12 – Camada DAL Arquitetura Gerada



Fonte: Elaborada pelo autor

As classes VO contidas na arquitetura gerada são criadas a partir da definição dada pelo usuário e estão contidas na estrutura composta pelo conjunto de atributos e entidades de um projeto, onde cada entidade representa uma classe gerada e cada atributo representará uma propriedade da classe gerada sendo utilizadas para armazenar os dados obtidos na interface gerada até a camada DAL e vice-versa.

As interfaces do sistema são geradas na linguagem HTML, por meio de telas para inserção, edição, remoção, busca e listagem de dados, permitindo que o recebimento dos dados ocorrida na camada *View* sejam transformados em um ou vários VOs através da linguagem escolhida para criação do sistema, e para gerar a arquitetura os códigos geradores estão divididos em quatro namespaces: geradorDAO, geradorSGBD, geradorVO e geradorVIEW. Cada um deles contém classes responsáveis por gerar cada uma das camadas da arquitetura gerada, funcionando assim como um verdadeiro motor de transformação do PM preenchido pelo usuário, com o auxilio de um motor semântico, em um sistema funcional e bem estruturado, pronto para gerir as informações que o usuário venha a precisar.

Todas as classes que fazem parte das namespaces geradoras e que são responsáveis por gerar o código de acordo com a linguagem escolhida implementam uma interface denominada `IGerador` permitindo que todos os geradores tenham a mesma definição da classe base garantindo aos geradores a obrigatoriedade de possuir os mesmos métodos, apesar da utilização de implementações diferentes, `getStringGerada( )` e `getNomeArquivo( )`, como ilustra a Fig. 4.13. O método `getStringGerada( )` deve ser implementado para que retorne uma String com o conteúdo do arquivo a ser gerado,

enquanto que o método `getNomeArquivo( )` deverá retornar o nome do arquivo com sua extensão.

Figura 4.13 – Interface `IGerador`

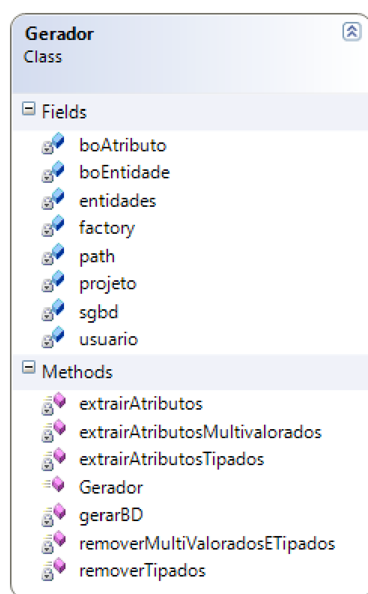
```
using System;
namespace LibraryGeradorSemantico.BO
{
    public interface IGerador
    {
        String getStringGerada();
        String getNomeArquivo();
    }
}
```

Fonte: Elaborada pelo autor

4.2.2.2.1 Gerador SGBD

O namespace geradorSGBD é responsável por gerar o *script* de criação das tabelas para o SGBD escolhido e está dividido em três estruturas: *Façade*, *Factory* e namespace contendo os geradores. A *Façade* é descrita pela classe `Gerador`, ilustrada na Fig. 4.14, e contém os métodos necessários para modelar um banco de dados relacional através da definição dada pelo usuário.

Figura 4.14 – Classe `Gerador` do namespace geradorSGBD



Fonte: Elaborada pelo autor

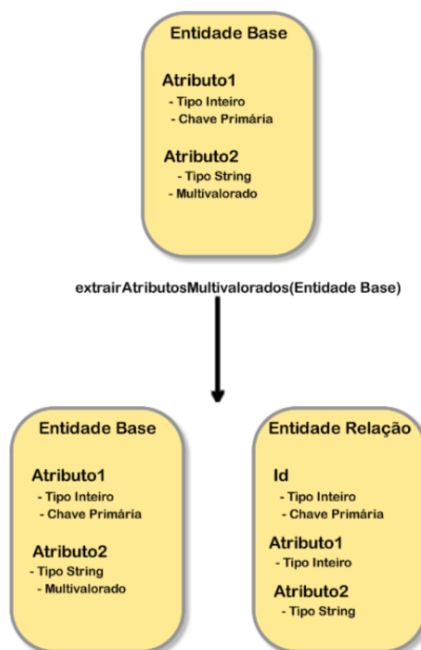
Para gerar um esquema entidade-relacional e modelar o PM específico para o SGBD através da definição do sistema, a classe `Gerador` precisa primeiro normalizar os dados obtidos através da verificação da análise e formatação dos mesmos, isso se torna necessário par que todos os dados definidos pelo usuário na descrição do sistema possam ser persistidos sem a ocorrência de inconsistência e/ou perda de informações.

Primeiro é verificado, para cada entidade, se algum de seus atributos possui alguma propriedade multivalorada, e caso possua é necessário aplicar a segunda forma normal, removendo assim todo atributo não-chave que independa da chave primária da entidade e criando uma nova entidade pra armazenar esse atributo não-chave através do método `extrairAtributosMultivalorados( )`, que é construído utilizando o seguinte algoritmo:

1. Cria-se uma nova entidade.
2. Preenche os dados dessa nova entidade para identificá-la como uma entidade de relação.
3. Adiciona-se um atributo à nova entidade.
    a. O atributo criado será a chave primária da entidade de relação, e que terá o nome ID do tipo inteiro.
4. Adiciona-se um segundo atributo à nova entidade.
    a. Esse atributo será a chave estrangeira, referenciando a entidade que possui o atributo multivalorado e terá o nome e o tipo do atributo que seja a chave primária da entidade base.
5. Adiciona-se um terceiro atributo à nova entidade.
    a. Esse atributo armazenará o valor do atributo multivalorado e receberá o nome e o tipo de dado do atributo multivalorado da entidade base.

A Fig. 4.15 ilustra o resultado obtido após a normalização de um atributo multivalorado.

Figura 4.15 – Resultado da Normalização para o Atributo Multivalorado Atributo2



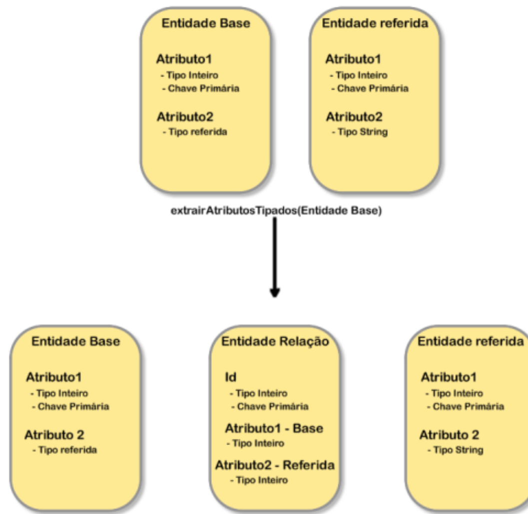
Fonte: Elaborada pelo autor

Para cada entidade, também é verificado se ela possui algum atributo tipado, ou seja, algum atributo cujo tipo de dado referencie alguma outra entidade e caso exista, é necessário criar uma entidade de relação, porém a entidade precisará conter os dados das duas entidades referenciadas e para isso, o método `extrairAtributosTipados( )` implementa o seguinte algoritmo:

1. Cria-se uma nova entidade.
2. Preenche os dados dessa nova entidade para identificá-la como uma entidade de relação.
3. Adiciona-se um atributo à nova entidade.
   a. O atributo criado será a chave primária da entidade de relação e terá o nome ID do tipo inteiro.
4. Adiciona-se um segundo atributo à nova entidade.
   a. Esse atributo será a chave estrangeira, referenciando a entidade que possui o atributo tipado e terá o nome e o tipo do atributo que seja a chave primária da entidade base.
5. Adiciona-se um terceiro atributo à nova entidade.
   a. Esse atributo será outra chave estrangeira, referenciando a entidade que foi referenciada pelo atributo tipado e receberá o nome e o tipo de dado do atributo multivalorado da entidade referenciada.

A Fig. 4.16 ilustra o resultado obtido após a normalização de um atributo tipado.

Figura 4.16 – Resultado da Normalização para o Atributo Tipado Atributo2



Fonte: Elaborada pelo autor

Após a extração das entidades de relação, é necessário aplicar a terceira forma normal na estrutura, transformando-a em uma estrutura relacional normalizada, para isso são retirados todos os atributos tipados ou multivalorados contidos nas entidades através do método `removerMultivalorado( )`, ilustrado na Fig. 4.17.

Figura 4.17 – Algoritmo para Retirar todos os Atributos Multivalorados ou Tipados

```
private void removerMultiValoradosETipados(VO.Entidade entidade)
{
    for (int i = 0; i < entidade.Atributos.Count; i++)
        if (entidade.Atributos[i].MultiValorado ||
             (
               !entidade.Atributos[i].TipoDado.Equals(Resources.Dados.tipoDeDados.String) &&
               !entidade.Atributos[i].TipoDado.Equals(Resources.Dados.tipoDeDados.Texto) &&
               !entidade.Atributos[i].TipoDado.Equals(Resources.Dados.tipoDeDados.Inteiro) &&
               !entidade.Atributos[i].TipoDado.Equals(Resources.Dados.tipoDeDados.Double) &&
               !entidade.Atributos[i].TipoDado.Equals(Resources.Dados.tipoDeDados.Data)
             )
            )
            entidade.Atributos.Remove(entidade.Atributos[i]);
}
```

Fonte: Elaborada pelo autor

Logo após realizar a normalização de todas as entidades da estrutura, existe agora uma nova coleção de entidades contendo, além das entidades definidas pelo usuário, as novas entidades de relação e excluídas todos os atributos multivalorados e tipados e dessa forma, agora todas as entidades do sistema estão mapeadas em um modelo entidade-relacional, formando um PM específico para o SGBD.

Com O PM específico gerado, é necessário gerar o *script* do banco de dados, contendo todas as suas tabelas, colunas e estruturas baseadas no PM. A *Factory*, ilustrada na Fig. 4.18, faz o papel de escolher qual gerador deverá ser usado levando em conta qual SGBD foi definido pelo usuário, de forma que a fachada nunca saiba qual gerador foi escolhido gerando um desacoplamento entre a fachada e os geradores de forma que possam ser criados vários outros geradores, um para cada SGBD, sem a necessidade de realizar alterações fachada.

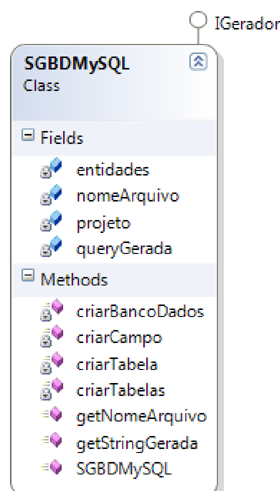
Figura 4.18 – *Factory* dos Geradores do SGBD

```
public IGerador obterBanco(VO.SGBD sgbd,List<VO.Entidade> entidades, String projeto)
{
    if (sgbd.Nome.Equals(Resources.Bancos.Bancos.MySQL))
    {
        try
        {
            return new SGBDMySQL(projeto,entidades);

        }
        catch (Exceptions.elementoIncompleto ex)
        {
            throw ex;
        }
    }

    throw new Exceptions.elementoNaoEncontrado("SGBD Não encontrado!");


}
```

Fonte: Elaborada pelo autor

O namespace Geradores contém todas as classes responsáveis por criar o *script* de criação do banco de dados, tabelas e colunas para cada SGBD suportado pelo sistema, como exemplificado na Fig. 4.19 no qual utiliza um gerador para o SGBD MySQL. Todas as classes implementam a interface `IGerador` e utilizam o PM específico gerado na etapa anterior para tal.

Figura 4.19 – Classe `SGBDMySQL`



Fonte: Elaborada pelo autor

A classe `Gerador` gera o *script* do banco de dados requerido através de uma sequência de algoritmos, que primeiro geram a estrutura do banco de dados, depois a estrutura de cada tabela e, por último, a estrutura de cada coluna de cada tabela. Para gerar a estrutura do banco de dados, o método `criarBancoDados( )`, ilustrado na Fig. 4.20, é invocado, gerando uma String contendo o SQL necessário para a criação do banco de dados utilizando a configuração do banco de dados charset utf8, no qual proporciona uma maior compatibilidade com o idioma português e tendo um nome igual ao do projeto, além disso elimina todos os espaços vazios e caso o nome do projeto seja vazio, uma exceção do tipo `elementoIncompleto` é levantada. A Figura 4.20 exibe o código para criação do banco de dados.

Figura 4.20 – Método `criarBancoDados( )`

```
private string criarBancoDados()
{
    if (projeto.Trim().Equals(""))
        throw new Exceptions.elementoIncompleto("Nome do projeto não pode ser vazio!");

    return "CREATE DATABASE " + projeto.Replace(" ", "") + " DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci;\n";
}
```

Fonte: Elaborada pelo autor

Depois de escrito o *script* de criação , a classe `SGBDMySQL` cria as tabelas que irão estar contidas dentro do banco de dados utilizando o método `criarTabela( )`, descrito na Fig. 4.21, no qual cada entidade do modelo PM passado para a classe `SGBDMySQL` servirá como base para criação de uma tabela no banco de dados recebendo o nome da entidade.

Figura 4.21 – Método `criarTabela( )`

```
private String criarTabela(VO.Entidade entidade)
{
    String query = "DROP TABLE IF EXISTS " + projeto.Replace(" ", "") + "." + entidade.Nome + ";\n";
    query += "CREATE  TABLE IF NOT EXISTS " + projeto.Replace(" ", "") + "." + entidade.Nome + " (";

    int i = 0;

    foreach (VO.Atributo atributo in entidade.Atributos)
    {
        if (!atributo.MultiValorado)
        {
            query += criarCampo(atributo);

            i++;
            if (i != entidade.Atributos.Count)
                query += ",";
        }

    }

    query.Remove(query.Length - 1);
    query += ")ENGINE=InnoDB;\n\n";

    return query;
}
```

Fonte: Elaborada pelo autor

Como ilustrado na Fig. 4.22, cada atributo da entidade virará uma coluna na tabela e será criado pelo método `criarCampo( )` que utiliza o seguinte algoritmo para criar a coluna:

1. Se o Atributo possuir a propriedade nulo como verdadeiro, adicione “NULL” ao script, caso contrário adicione “NOT NULL”.
2. Se o tipo de dado do atributo for inteiro e ele for uma chave primária, adicione “AUTO_INCREMENT” ao *script*.
3. Se o tipo de dado do atributo for Texto ou String adicione VARCHAR seguido do tamanho do atributo ao *script*. Caso contrário, adicione o tipo de dado do atributo.
4. Se o atributo for uma chave primária adicione “PRIMARY_KEY” ao *script*.

Figura 4.22 – Método `criarCampo( )`

```
private string criarCampo(VO.Atributo atributo)
{
    String nulo =
                atributo.Nulo == true ?
                "NULL" :
                "NOT NULL";
    String autoIncrement =
                    atributo.TipoDado == Resources.Dados.tipoDeDados.Inteiro &&
                    atributo.ChavePrimaria == true ?
                    "AUTO_INCREMENT" :
                    "";
    String tipoDado =
                    atributo.TipoDado == Resources.Dados.tipoDeDados.String ||
                    atributo.TipoDado == Resources.Dados.tipoDeDados.Texto ?
                    "VARCHAR(" + atributo.Tamanho + ")" :
                    atributo.TipoDado;

    String primaryKey =
                    atributo.ChavePrimaria == true ?
                    "PRIMARY KEY" :
                    "";

    String query = "`" + atributo.Nome + "` " + tipoDado + " " + nulo + " " +
                autoIncrement + " " + primaryKey + "\n";

    return query;
}
```

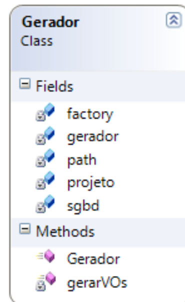
Fonte: Elaborada pelo autor

4.2.2.2.2 Gerador VO

O Namespace geradorVO é responsável por gerar todas as classes VO do sistema gerado a partir das entidades do sistema, e para serem gerados utilizam as três estruturas da namespace geradorVO: uma fachada, uma factory e uma subnamespace contendo os geradores. A Fachada é implementada pela classe Gerador, ilustrada na Fig. 4.23, e contém os métodos necessários para invocar os criadores de todos os VO`s do sistema gerado. O PM utilizado para gerar os VO´s é o mesmo que foi descrito pelo usuário do sistema, ou seja, o

PM é a própria ontologia de instância, sendo assim, a fachada só invoca os geradores responsáveis para a criação dos arquivos, sem nenhuma modelagem de um PM específico.

Figura 4.23 – Fachada Gerador VO



Fonte: Elaborada pelo autor

Já Factory, ilustrada na Fig. 4.24, faz o papel de construtora de geradores, de forma que a fachada nunca saiba qual construtor será instanciado, a partir da escolha da linguagem pelo usuário durante a criação do PM, transformando os geradores totalmente desacoplados da fachada.

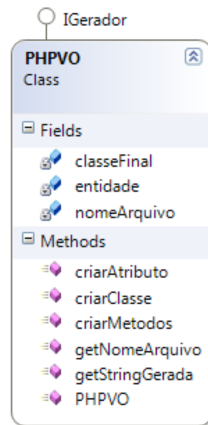
Figura 4.24 – Factory dos Geradores do VO.

```
class FactoryVO
{
    public IGerador obterVOs(VO.Entidade entidade, String linguagem)
    {
        if (linguagem.Equals(Resources.Linguagens.Linguagem.PHP))
        {
            try
            {
                return new vo.geradores.PHPVO(entidade);

            }
            catch (Exceptions.elementoIncompleto ex)
            {
                throw ex;
            }
        }

        throw new Exceptions.elementoNaoEncontrado("Linguagem Não suportada!");


    }
}
```

Fonte: Elaborada pelo autor

Existe um gerador de VOs para cada linguagem suportada, exemplificado na Fig. 4.25 pela classe PHPVO que gera VOs para a linguagem PHP, e esses geradores implementam a interface IGerador. Cada um dos geradores possui métodos específicos para criar a classe, métodos e atributos do VO para cada entidade dentro do PM, que se transformará em um VO no sistema gerado, e para cada atributo da entidade um atributo e seus respectivos métodos de alterações (*get* e *set*) serão também gerados.

Figura 4.25 – Classe Geradora PHPVO

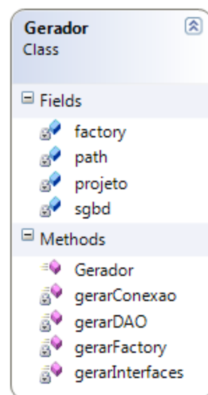


Fonte: Elaborada pelo autor

4.2.2.2.3 GeradorDAO

O namespace GeradorDAO contém todas as classes responsáveis para gerar a camada DAL do sistema gerado de forma a sempre manter a arquitetura explicada na seção anterior utilizando, além das classes *Factory* e Fachada (aqui denominada `Gerador` e exibida na Fig. 4.26), o namespace geradores que contém uma classe para gerar cada uma das classes que fazem parte da camada DAL. A Fachada `Gerador` tem o papel de invocar todos as classes que irão gerar a camada DAL, ou seja, executar todas as classes da namespace geradores.

Figura 4.26 – Gerador DAO

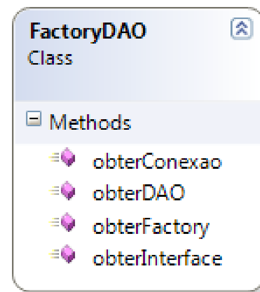


Fonte: Elaborada pelo autor

A *Factory*, como exibida na Fig. 4.27, tem um papel de instanciar todos os geradores de acordo com as definições dadas pelo usuário no PM, como, por exemplo, se o usuário

definiu a linguagem como PHP e o banco de dados como MySQL a *Factory* irá instanciar os geradores para a linguagem PHP e MySQL sem que a camada geradora saiba qual classe gerador foi invocado e dessa forma, a camada geradora somente obterá um resultado, sem saber quem está fazendo o código.

Figura 4.27 – Factory DAO



Fonte: Elaborada pelo autor

A namespace geradores contém as classes que são responsáveis por gerar as classes Conexão, *Factory*, a interface DAO e o DAO para cada linguagem, e para cada SGBD suportados pelo sistema existe um conjunto de classes geradoras de conexões, *Factory*, interface DAO e DAO onde todas essas classes implementam a interface `IGerador` e são instanciadas pela *Factory*.

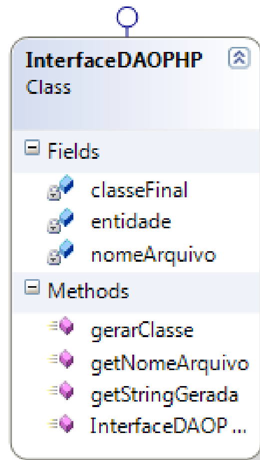
A classe que gera a Conexão com o SGBD escolhido é responsável por criar um arquivo de conexão que será utilizado pela classe DAO e é criado utilizando todas as informações contidas no PM que se refiram ao SGBD. A Fig 4.28 exemplificada uma classe Conexão com o SGBD MySQL.

Figura 4.28 – Classe Geradora Conexão

Fonte: Elaborada pelo autor

Para gerar a interface, a classe geradora Interface, exemplificada na Fig. 4.29 no exemplo de uma classe geradora de uma interface PHP, utiliza o PM e cria uma interface contendo a assinatura dos métodos `adicionar( )`, `editar( )`, `remover( )`, `buscar( )` e `listar( )` para cada entidade do sistema escrita na linguagem definida pelo usuário.

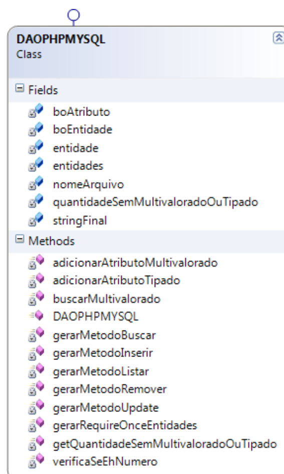
Figura 4.29 – Classe Geradora Interface



Fonte: Elaborada pelo autor

Cada classe DAO da camada DAL do sistema gerado deve ser responsável por ter as *queries* de manipulação com o banco de dados, dessa forma é gerada uma classe DAO para cada entidade do sistema que é escrita na linguagem e no SGBD escolhidos pelo usuário, definido pelo PM do sistema a ser gerado, a partir de uma classe geradora DAO que implementa a interface `IGerador`, como exemplificado na Fig. 4.30 através de um gerador para linguagem PHP e SGBD MySQL e é invocada pela *Factory*.

Figura 4.30 – Classe Geradora DAO



Fonte: Elaborada pelo autor

Como a classe DAO gerada interage diretamente com o SGBD gerado são necessários alguns ajustes ao PM que são feitos diretamente no código gerado, de forma que não é necessária uma re-modelagem do PM como acontece nos geradores do SGBD, dessa forma nos métodos `gerarMetodoInserir( )` e `gerarMetodoBuscar( )` são responsáveis por escrever também as *queries* que atendam as atribuições de uma entidade que possua atributos multivalorados ou tipados.

O método `gerarMetodoInserir( )` cria uma *query* para inserir valores a todos os atributos de uma entidade. Para tanto, ele implementa o seguinte algoritmo:

1. Escrever o código para estabelecer uma conexão.
2. Começar a escrever a *query* de inserção.
3. Para cada atributo da entidade, insira-o na *query* de inserção, exceto se esse atributo for multivalorado.
4. Escrever o final da *query* se inserção e o código para executá-la.
5. Para cada atributo multivalorado ou tipado, inserir uma *query* de inserção para a tabela normalizada.

Para executar o algoritmo acima descrito, o método `gerarMetodoInserir( )` utiliza dois métodos auxiliares: `adicionarAtributosMultivalorado( )` e `adicionarAtributosTipado( )`, onde este último é ilustrado na Fig 4.31 e que é possível observar uma busca nas entidades do PM para encontrar qual a entidade referida e sua chave primária, de forma a criar uma *query* que insira os dados necessários na tabela de relação, criada pelos geradores do SGBD.

Figura 4.31 – Método `adicionarAtributoTipado( )`

```
private String adicionarAtributoTipado(VO.Atributo atributo)
{

    VO.Entidade entidadeReferenciada = boEntidade.buscar(atributo.TipoDado, entidades);
    VO.Atributo atributoPKEntidade = boAtributo.copiar(boEntidade.obterChavePrimaria(entidade));
    VO.Atributo atributoPKEntidade2 = boAtributo.copiar(boEntidade.obterChavePrimaria(entidadeReferenciada));

    String funcao = "$" + atributo.Nome + "s = $" + entidade.Nome + "->get" + atributo.Nome + "();\n\n";

    funcao += "for($i=0;$i<sizeof($" + atributo.Nome + "s);$i++){ \n" +
              "$sql = \"INSERT INTO " + entidade.Nome + atributo.Nome + " VALUES(0," +
                verificaSeEhNumero(atributoPKEntidade.TipoDado) +
             "\";\n" +
            "$sql .=  $" + entidade.Nome + "->get" + atributoPKEntidade.Nome + "().\"" +
             verificaSeEhNumero(atributoPKEntidade.Ti (local variable) VO.Atributo atributoPKEntidade
             "," +
            verificaSeEhNumero(atributoPKEntidade2.TipoDado) +
            "\";\n" +
            "$sql .=  $" + atributo.Nome + "s[$i].\"" +
            verificaSeEhNumero(atributoPKEntidade2.TipoDado) +
            "\";\n" +
            "$sql.= \")\";\n" +
            "$query = $this->conexao->executarQuery($sql);\n}\n\n";
    return funcao;

}
```

Fonte: Elaborada pelo autor

O método `adicionarAtributoMultivalorado( )`, ilustrado na Fig. 4.32, tem um funcionamento parecido com o método `adicionarAtributoTipado( )`, já que ele também cria a *query* de inserção para a tabela de relação, porém ele não utiliza nenhuma entidade de referencia.

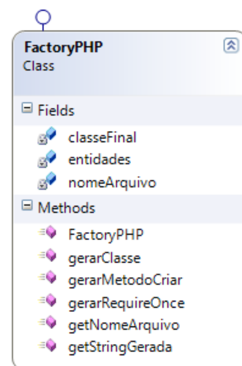
Figura 4.32 – Método `adicionarAtributoMultivalorado( )`

```
private String adicionarAtributoMultivalorado(VO.Atributo atributo)
{

    if (
                      !atributo.TipoDado.Equals(Resources.Dados.tipoDeDados.String) &&
                      !atributo.TipoDado.Equals(Resources.Dados.tipoDeDados.Texto) &&
                      !atributo.TipoDado.Equals(Resources.Dados.tipoDeDados.Inteiro) &&
                      !atributo.TipoDado.Equals(Resources.Dados.tipoDeDados.Double) &&
                      !atributo.TipoDado.Equals(Resources.Dados.tipoDeDados.Data)
        )
    {
        return adicionarAtributoTipado(atributo);
    }

    VO.Atributo atributoPKEntidade = boAtributo.copiar(boEntidade.obterChavePrimaria(entidade));
    String funcao = "$" + atributo.Nome + "s = $" + entidade.Nome + "->get" + atributo.Nome + "();\n\n";

    funcao += "for($i=0;$i<sizeof($" +atributo.Nome+"s);$i++){ \n" +
              "$sql = \"INSERT INTO " + entidade.Nome+atributo.Nome + " VALUES(0," +
                verificaSeEhNumero(atributoPKEntidade.TipoDado) +
             "\";\n" +
            "$sql .=  $" + entidade.Nome + "->get" + atributoPKEntidade.Nome + "().\"" +
             verificaSeEhNumero(atributoPKEntidade.TipoDado) +
             "," +
            verificaSeEhNumero(atributo.TipoDado) +
            "\";\n" +
            "$sql .=  $"+atributo.Nome+"s[$i].\"" +
            verificaSeEhNumero(atributo.TipoDado)+
            "\";\n" +
            "$sql.= \")\";\n" +
            "$query = $this->conexao->executarQuery($sql);\n}\n\n";
    return  funcao;
}
```

Fonte: Elaborada pelo autor

A classe geradora `Factory`, exemplificada pela Fig. 4.33 em uma classe `Factory` para a linguagem PHP, é responsável por gerar a *Factory* que vai instanciar as classes `DAOGeradas`, ela permite que o usuário possa criar mais DAO´s em seu código gerado no futuro, se assim desejar, obtendo uma total independência entre a camada gerada *View* e camada DAL.

Figura 4.33 – *Factory* Gerada

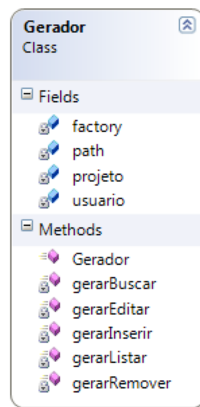


Fonte: Elaborada pelo autor

4.2.2.2.4 GeradorView

Toda a camada *View* do sistema gerado é criada pelas classes contidas no namespace geradorView e possuem uma estrutura bastante similar a todos os outros, formado por uma Fachada, Gerador, uma *Factory* e um namespace que contém as classes geradoras, onde todas elas implementam a interface `IGerador`, onde a Fachada, ilustrada na Fig. 4.34, realiza somente a chamada da *Factory*, esperando somente o resultado dos geradores para então criar o código gerado.

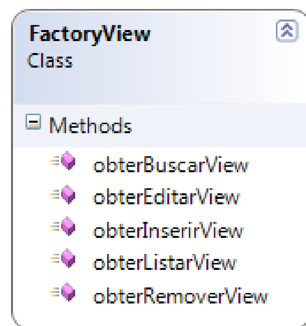
Figura 4.34 – Fachada Gerador do `geradorView`



Fonte: Elaborada pelo autor

A classe `Factory`, ilustrada na Fig. 4.35, é a responsável por instanciar todos os geradores, de forma que a fachada não saiba quem gerou o código através da escolha de quem vai gerar o código de acordo com a linguagem definida pelo usuário no PM.

Figura 4.35 – Factory do geradorView

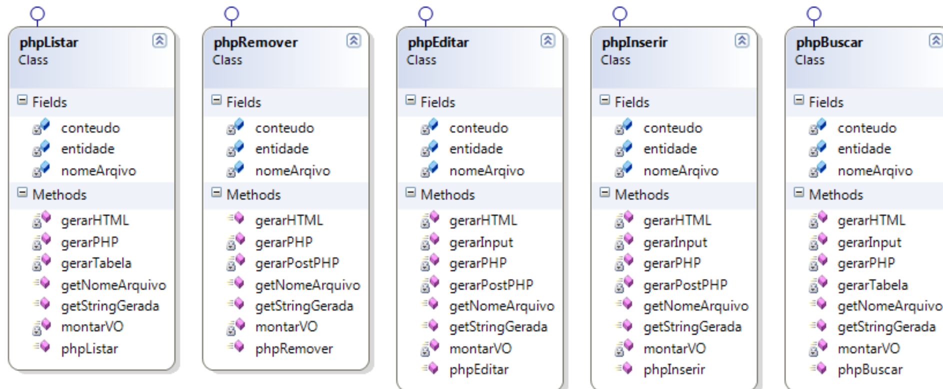


Fonte: Elaborada pelo autor

A namespace Geradores contém todos os geradores que implementam a interface `IGerador` sendo que para cada linguagem escolhida pelo usuário no PM um conjunto de

geradores é responsável por gerar toda a interface do sistema, na linguagem HTML e todos os geradores possuem métodos parecidos, e são exibidos na Fig. 4.36.

Figura 4.36 – Geradores do geradorView



Fonte: Elaborada pelo autor

Os geradores de inserir e editar possuem métodos específicos para criar os campos responsáveis por captarem os dados do usuário que podem variar de acordo com as propriedades escolhidas nos atributos das classes, como exemplificado na Fig. 4.37 na qual exibe o método que gera esses campos, de acordo com a propriedade `tipoExibição` de um atributo de uma classe.

Figura 4.37 – Método `gerarInput( )`

```
private String gerarInput(VO.Atributo atributo)
{

    String input = "<p>" + atributo.Nome;

    if (atributo.TipoExibicao.Equals(Resources.View.View.Valor))
        input += "<input type='text' name='txt" + atributo.Nome + "' />\n";
    else if (atributo.TipoExibicao.Equals(Resources.View.View.Texto))
        input += "<textarea name='txt" + atributo.Nome + "'> </textarea>\n";
    else if (atributo.TipoExibicao.Equals(Resources.View.View.Imagem))
        input += "<input type='file' name='img"+atributo.Nome+"'/>\n";
    else if (atributo.TipoExibicao.Equals(Resources.View.View.Arquivo))
        input += "<input type='file' name='arquivo"+atributo.Nome+"'/>\n";

    input += "</p>";
    return input;

}
```

Fonte: Elaborada pelo autor

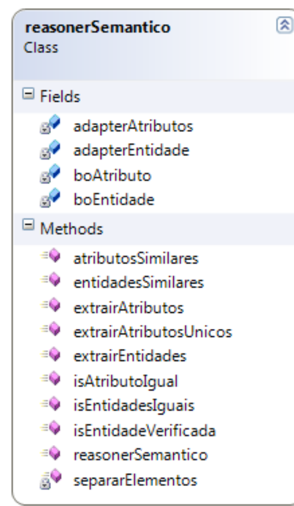
4.3 Wizzard

A camada Wizzard é responsável por toda a área semântica de forma a auxiliar a captação dos dados para gerar o PM, de forma que ela é de primordial importância neste

trabalho por ser exatamente o motor semântico de todo o sistema de captação de dados. A Wizzard tem como função mostrar ao usuário que ele pode melhorar a criação do seu PM utilizando definições de entidades que se encaixem melhor em seu projeto através de uma análise semântica seguindo as regras definidas na ontologia do sistema.

A análise semântica é feita através das seguintes atividades: Semelhança entre entidades e atributos, Igualdade entre entidades e atributos, Validação de entidades e Extração de entidades e atributos. Todas essas diretrizes estão descritas em uma única classe, o *reasoner* semântico. A Fig. 4.38 exibe o *reasoner* semântico.

Figura 4.38 – Reasoner Semântico



Fonte: Elaborada pelo autor

Para verificar quais as entidades que sejam semelhantes a uma entidade dada, ou seja, àquelas que se encaixam no axioma Semelhança, o método `entidadesSimilares( )` realiza o seguinte algoritmo:

1. Obtém-se a lista de entidades que possuam o nome igual ou composto pelo nome da entidade dada.
2. Para cada entidade na lista, verifique:
   a. Se as duas entidades possuem atributos.
      i. Caso possuam, verifique se a quantidade de atributos iguais das duas classes é igual ou superior a uma porcentagem determinada da quantidade de atributos da entidade que possui mais atributos.
   b. Caso a quantidade de atributos seja maior que certa porcentagem ou caso uma das entidades não possuir atributos, adicione-a na lista de entidades semelhantes.

c. Caso contrário, não adicione.

A Fig. 4.39. ilustra o método `entidadesSimilares( )`, que implementa o algoritmo acima.

Figura 4.39 – Método `entidadesSimilares( )`

```
public List<VO.Entidade> entidadesSimilares(VO.Entidade entidade)
{
    DataTable tabela = adapterEntidade.GetDataByNome(entidade.Nome);
    List<VO.Entidade> retorno = new List<VO.Entidade>();
    foreach (DataRow r in tabela.Rows)
    {
        try{
            VO.Entidade entidadeBuscada = boEntidade.buscar(r.Field<int>("id"));
            double atributosIguais = 0;
            bool entidadeNaoPossuiAtributos = false;
            if (!(entidade.Atributos.Count == 0 || entidadeBuscada.Atributos.Count == 0))
            {
                foreach (VO.Atributo atributo in entidade.Atributos)
                    foreach (VO.Atributo atributoBuscado in entidadeBuscada.Atributos)
                        if (isAtributoIgual(atributo, atributoBuscado))
                            atributosIguais++;
                int maiorQuantidadeAtributos = 0;
                if (entidade.Atributos.Count >= entidadeBuscada.Atributos.Count)
                    maiorQuantidadeAtributos = entidade.Atributos.Count;
                else
                    maiorQuantidadeAtributos = entidadeBuscada.Atributos.Count;
                atributosIguais = (atributosIguais / maiorQuantidadeAtributos) * 100;
            }
            else
                entidadeNaoPossuiAtributos = true;
            if(atributosIguais >= 0.5 || entidadeNaoPossuiAtributos)
                retorno.Add(entidadeBuscada);
        }
         catch (Exceptions.elementoNaoEncontrado ex)
        {
        }
    }
    return retorno;
}
```

Fonte: Elaborada pelo autor

Para verificar quais as atributos que sejam semelhantes a um atributo dado, ou seja, àquelas que se encaixam no axioma Semelhança, o método `atributosSimilares( )` realiza o seguinte algoritmo:

1. Obtém-se a lista de atributos que possuam o nome igual ou composto pelo nome do atributo dado.
2. Para cada atributo na lista, verifique:
    a. Se possui pelo menos duas propriedades iguais (`chavePrimaria`, `multivalorado`, `nulo`, `tamanho`, `tipoDado` e `tipoExibição`) ou um dos atributos não possuam as propriedades preenchidas, adicione-o na lista de semelhantes.
    b. Caso contrário, não adicione.

A Fig. 4.40. ilustra o método `atributosSimilares( )`, que implementa o algoritmo acima.

Figura 4.40 – Método `atributosSimilares( )`

```
public List<VO.Atributo> atributosSimilares(VO.Atributo atributo)
{
    DataTable tabela = adapterAtributos.GetDataByNome(atributo.Nome);
    List<VO.Atributo> retorno = new List<VO.Atributo>();
    foreach (DataRow r in tabela.Rows)
    {
        VO.Atributo atributoBuscado = boAtributo.buscar(r.Field<int>("id"));
        try
        {
            bool atributoSemelhante = false;
            if (atributo.Id == -1)
                atributoSemelhante = true;
            else
            {
                int propriedadesIguais=0;
                if(atributo.ChavePrimaria.Equals(atributoBuscado.ChavePrimaria))
                    propriedadesIguais++;
                if(atributo.MultiValorado.Equals(atributoBuscado.MultiValorado))
                    propriedadesIguais++;
                if(atributo.Nulo.Equals(atributoBuscado.Nulo))
                    propriedadesIguais++;
                if(atributo.Tamanho.Equals(atributoBuscado.Tamanho))
                    propriedadesIguais++;
                if(atributo.TipoDado.Equals(atributoBuscado.TipoDado))
                    propriedadesIguais++;
                 if(atributo.TipoExibicao.Equals(atributoBuscado.TipoExibicao))
                    propriedadesIguais++;
                if(propriedadesIguais >= 2)
                    atributoSemelhante = true;
            }
            if(atributoSemelhante)
                retorno.Add(atributoBuscado);
        }
        catch (Exceptions.elementoNaoEncontrado ex)
        {
        }
    }
    return retorno;
}
```

Fonte: Elaborada pelo autor

Para verificar se uma entidade é igual à outra, o algoritmo abaixo é implementado:

1. Verifica se as duas entidades possuem a mesma quantidade de atributos.
    a. Caso possuam, verifica se todos os atributos são iguais, caso sejam, retorne verdadeiro, caso contrário, retorne falso.
2. Caso não possuam, retorne falso.

A Fig. 3.41 ilustra o algoritmo acima descrito implementado.

Figura 4.41 – Método `isEntidadeIgual( )`

```
public bool isEntidadesIguais(VO.Entidade entidade1, VO.Entidade entidade2)
{
    bool retorno = true;
    if (entidade1.Atributos.Count != entidade2.Atributos.Count)
        retorno = false;
    if(entidade1.AtributosString.Equals(entidade2.AtributosString))
        retorno = false;
    if (retorno)
    {
        foreach (VO.Atributo atributo1 in entidade1.Atributos)
        {
            bool encontrado = false;
            foreach (VO.Atributo atributo2 in entidade2.Atributos)
                if (isAtributoIgual(atributo1, atributo2))
                {
                    encontrado = true;
                    break;
                }

            if (!encontrado)
            {
                retorno = false;
                break;
            }
        }
    }
    return retorno;
}
```

Fonte: Elaborada pelo autor

Para verificar se um atributo é igual a outro, o algoritmo abaixo é implementado:

1. Verifica se todas essas propriedades são iguais: `chavePrimaria`, `multivalorado`, `nulo`, `tamanho`, `tipoDado` e `tipoExibição`. Se forem iguais, então os atributos são considerados iguais.

A Fig. 4.42 ilustra a implementação do algoritmo acima descrito.

Figura 4.42 – Método `isAtributoIgual( )`

```
public bool isAtributoIgual(VO.Atributo atributo1, VO.Atributo atributo2)
{
    bool retorno = true;

    if (!atributo1.ChavePrimaria.Equals(atributo2.ChavePrimaria) ||
        !atributo1.MultiValorado.Equals(atributo2.MultiValorado) ||
        !atributo1.Nulo.Equals(atributo2.Nulo) ||
        !atributo1.Tamanho.Equals(atributo2.Tamanho) ||
        !atributo1.TipoDado.Equals(atributo2.TipoDado) ||
        !atributo1.TipoExibicao.Equals(atributo2.TipoExibicao)
        )
        retorno = false;


    return retorno;
}
```

Fonte: Elaborada pelo autor

Para validar uma entidade é necessário verificar se todos os seus atributos são válidos, isto é, eles estão descritos na lista de atributos de uma entidade e também na descrição dos atributos contida na propriedade `atributoString` da entidade. A Fig. 4.43 ilustra a codificação do método `isEntidadeVerificada( )`.

Figura 4.43 – Método isEntidadeVerificada( )

```
public bool isEntidadeVerificada(VO.Entidade entidade)
{
   bool retorno = false;

    String atributos = "";

    foreach(VO.Atributo atributo in entidade.Atributos)
        atributos += atributo.Nome+";";

    if(atributos.Equals(entidade.AtributosString))
        retorno = true;

    return retorno;
}
```

Fonte: Elaborada pelo autor

A Extração de uma entidade nada mais é do que dada uma lista de nomes de entidades, é criada uma nova lista de entidades sem atributos e com as propriedades vazias, de forma que sejam preenchidas pelo usuário. A Fig. 4.44 ilustra o método `extrairEntidades( )`.

Figura 4.44 – Método extrairEntidades( )

```
private List<VO.Entidade> extrairEntidades(String texto)
{
    List<VO.Entidade> entidades = new List<VO.Entidade>();

    foreach (String e in separarElementos(texto))
    {
        VO.Entidade entidade = new VO.Entidade();
        entidade.Id = -1;
        entidade.Nome = e;
        entidade.Descricao = "";
        entidades.Add(entidade);
    }
    return entidades;
}
```

Fonte: Elaborada pelo autor

Para extrair atributos de uma dada lista contendo somente o nome dos atributos requeridos, o *reasoner* instancia uma lista de atributos com propriedades vazias, que serão preenchidas pelo usuário. A Fig. 4.45 ilustra o método `extrairAtributos( )`.

Figura 4.45 – Método `extrairAtributos( )`.

```
private List<VO.Atributo> extrairAtributos(String texto)
{
    List<VO.Atributo> atributos = new List<VO.Atributo>();

    foreach (String e in separarElementos(texto))
    {
        VO.Atributo atributo = new VO.Atributo();
        atributo.ChavePrimaria = false;
        atributo.DataCriacao = DateTime.Now;
        atributo.Descricao = "";
        atributo.Id = 0;
        atributo.MultiValorado = false;
        atributo.Nome = e;
        atributo.Nulo = false;
        atributo.Tamanho = 100;
        atributo.TipoDado = Resources.Dados.tipoDeDados.String;
        atributo.TipoExibicao = Resources.View.View.Texto;
        atributos.Add(atributo);
    }

    return atributos;
}
```

Fonte: Elaborada pelo autor

4.4 Resumo do capítulo

Neste capítulo foi explicada toda a arquitetura do *framework* proposto, exibindo sua estrutura de organização em camadas, seus geradores e seu motor semântico, todos eles escritos na linguagem C# e utilizando a plataforma Microsoft .Net *Framework* 4.0. Foi discutido também a arquitetura do sistema gerado e como os geradores a criam, utilizando técnicas de MDA para obter todo o código, as camadas de interface e até mesmo o *script* de criação do banco de dados na linguagem e SGBD escolhidos pelo usuário, assim como abordado o motor semântico, exibindo como o *framework* faz para auxiliar o usuário a construir seu sistema.

No capítulo seguinte será destacado o uso do *framework* em uma aplicação web denominada projeto PaMDA.

5 Estudo de Caso

Neste capítulo é apresentado o estudo de caso para o sistema proposto. Na seção 5.1 é mostrado o projeto PaMDA e a seção 5.2 é apresentada um breve resumo sobre o estudo de caso.

5.1 Introdução

A arquitetura descrita no Capítulo 4 foi aplicada em um sistema web, denominado Projeto PaMDA, que permite a criação de sistemas baseados na descrição do usuário na linguagem PHP e com o SGBD MySQL, e para sua construção foi utilizada a linguagem ASP.NET, de forma que o sistema possa ser acessível por qualquer pessoa independentemente de plataforma ou localidade geográfica.

O Projeto PaMDA se baseia na ontologia descrita, que define o PM do sistema, de forma que para acessá-lo é necessário possuir uma conta, a qual será associada todos os projetos criados pelo usuário. O Projeto PaMDA pode ser visualizado no Anexo A, a partir de um vídeo demonstrativo.

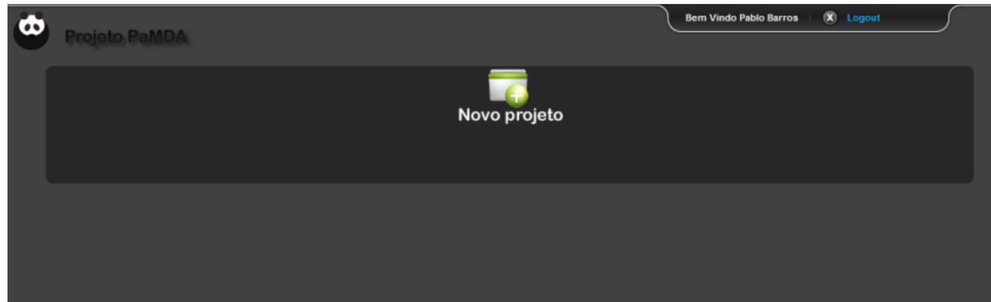
O *reasoner* semântico funciona sobre os projetos cadastrados no sistema, mas sem identificá-los, de forma que um usuário pode usar algumas entidades e/ou atributos criados por outro usuário, mas sem precisar conhecer sobre o projeto original nos quais essa entidade e/ou atributo pertencem. A Fig. 5.1 exibe a tela inicial deste estudo de caso.

Figura 5.1 – Tela Incial Projeto PaMDA

Fonte: Elaborada pelo autor

A partir de um cadastro, um usuário pode criar vários projetos, seguindo sempre a estrutura designada no PM. Para tanto, ele deve efetuar três etapas: descrever o projeto, descrever as entidades e descrever os atributos. Para descrever o projeto, o usuário acessa o link Novo projeto, na tela principal do sistema, exibida na Fig. 5.2.

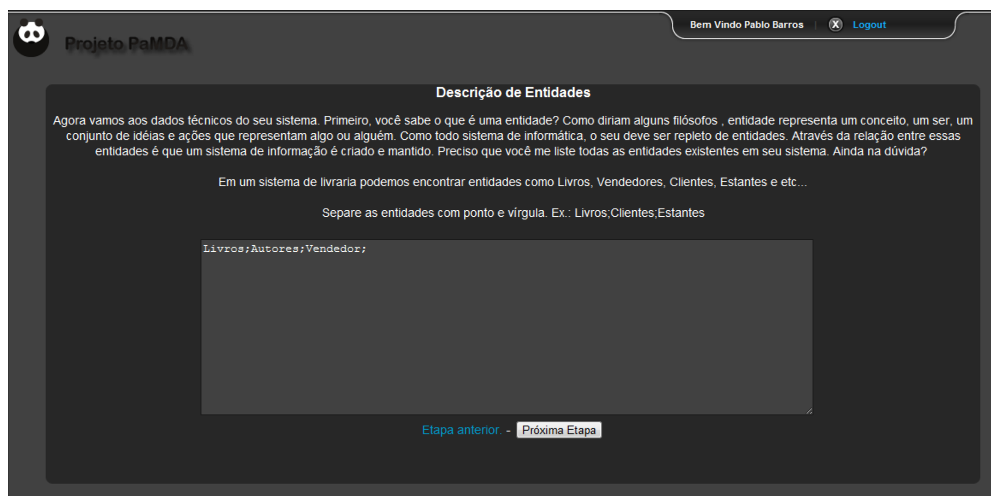
Figura 5.2 – Tela Principal



Fonte: Elaborada pelo autor

Depois de criado o projeto, a etapa de descrição do projeto tem inicio, através da escolha do nome do projeto e sua descrição, dada pelo próprio usuário. Após isso, cabe ao usuário identificar quais as entidades que irão conter o projeto, de forma que todas as entidades sejam explicitadas a partir de seu nome, separando-se por um ponto-e-vírgula, como ilustrado na Fig. 5.3, na última fase da etapa de criação de um projeto.

Figura 5.3– Tela da Etapa de Informar as Entidades



Fonte: Elaborada pelo autor

Finalizada a primeira etapa, o PaMDA se encarrega de criar um projeto com os dados inseridos pelo usuário, como ilustra a Fig. 5.4, porém sem conter nenhuma entidade, nenhum SGBD e nenhuma linguagem definida, já que essas informações serão passadas em etapas futuras. Este projeto é adicionado ao usuário, de forma que crie a ligação entre o projeto e o usuário, fazendo com que essas informações sejam persistidas na base de dados.

Figura 5.4 – Método para Criar um Novo Projeto

```
protected void Unnamed4_Click(object sender, EventArgs e)
{
    LibraryGeradorSemantico.VO.Projeto projeto =
                                new LibraryGeradorSemantico.VO.Projeto();
    projeto.DataCriacao = DateTime.Now;
    projeto.Descricao = txtDescricao.Text;
    projeto.Id = 0;
    projeto.Nome = txtNome.Text;
    projeto.Linguagem = "";
    projeto.Owl = "";
    projeto.SGBDS = new List<LibraryGeradorSemantico.VO.SGBD>();
    LibraryGeradorSemantico.BO.entidade.BOUsuario boUsuario =
                    new LibraryGeradorSemantico.BO.entidade.BOUsuario();
    int id = boUsuario.adicionarProjeto(projeto, usuario);
    projeto.Id = id;

    Session["projeto"] = projeto;
    Session["entidades"] = txtDescricao.Text;
    Response.Redirect("entidadesProjeto.aspx");
}
```

Fonte: Elaborada pelo autor

Criado o novo projeto, o PaMDA executa agora uma atividade semântica que consiste na extração das entidades a partir das informações dadas pelo usuário, utilizando a funcionalidade de extrair entidades do motor semântico da Wizzard, e então é montada uma interface personalizada para a construção das entidades, onde permite que o utilizador preencha os dados da entidade e terá a sua disposição entidades semelhantes que possam ser utilizadas na construção do seu projeto. Para tanto, quando a página `entidadesProjeto.aspx` é exibida realiza uma checagem na sessão, de forma a delimitar se é necessário ou não extrair as entidades, como é ilustrado na Fig. 5.5, e caso seja necessário, as novas entidades são criadas e anexadas ao projeto do usuário.

Figura 5.5 – Extração de Entidades na Página `entidadesProjeto.aspx`

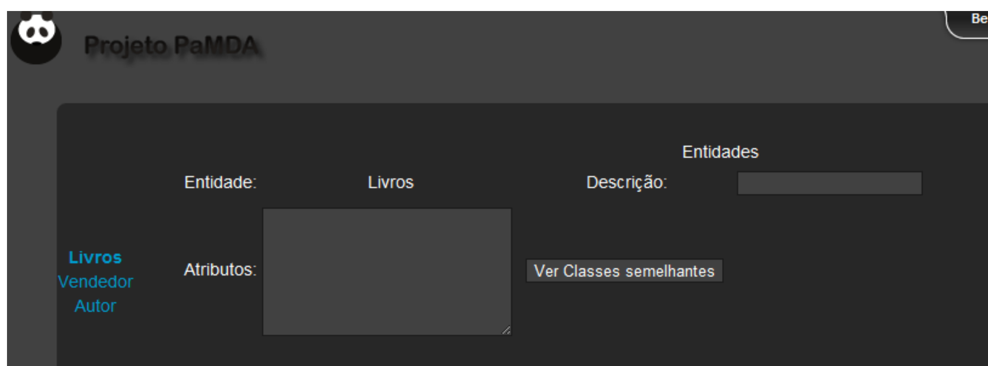
```
protected void Page_Init(object sender, EventArgs e)
{
    if (Session["projeto"] == null)
    {
        Response.Redirect("principal.aspx");
    }

    projeto = (LibraryGeradorSemantico.VO.Projeto)Session["projeto"];
    reasoner =
            new LibraryGeradorSemantico.BO.wizzard.reasoner.reasonerSemantico();
    if (Session["entidades"] != null)
    {
        List<LibraryGeradorSemantico.VO.Entidade> entidades =
                            reasoner.extrairEntidades((String)Session["entidades"]);
        projeto.Entidades = entidades;
        Session["entidades"] = null;
        Session["projeto"] = projeto;
    }

    gerarWizzard();
}
```

Fonte: Elaborada pelo autor

Realizada a extração e inserção das entidades ao projeto, é hora de realizar a criação da interface personalizada, ilustrada na Fig. 5.6, para preenchimento dos dados da entidade, isso é feito através do método `gerarWizzard( )`, que cria uma estrutura do ASP.NET

denominada *Step Wizzard*, preenchendo-a com os campos necessários para que cada entidade seja descrita pelo usuário e listando quais os atributos que pertencem àquela entidade, todos eles separados por ponto-e-vírgula, além de criar a estrutura responsável por invocar o motor semântico para exibição das entidades semelhantes à entidade a ser descrita.

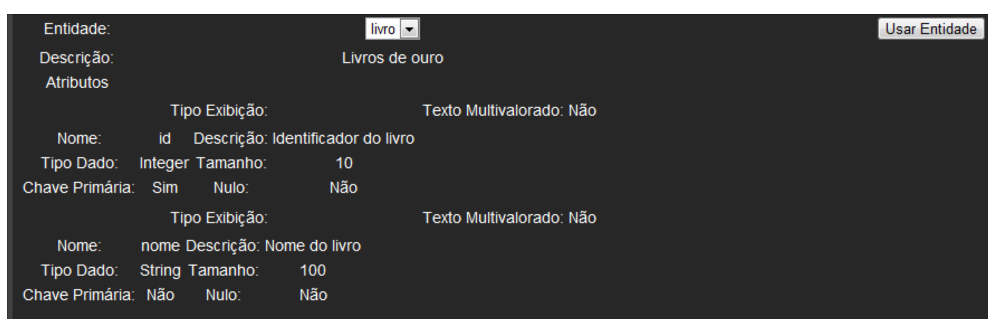
Figura 5.6 – Tela Personalizada para Criação de Interfaces



Fonte: Elaborada pelo autor

Ao se clicar no botão Ver Classes Semelhantes, uma lista de classes semelhantes à entidade a ser preenchida é exibida, como ilustra a Fig. 5.7. Essa lista é dada de acordo com a funcionalidade de encontrar entidades semelhantes delimitada no axioma da ontologia e implementada pelo motor semântico, e ao se escolher uma entidade semelhante, a entidade escolhida substitui a entidade que primeiro foi descrita pelo usuário, de forma que agora a entidade semelhante escolhida faça parte do projeto, enquanto a entidade que foi substituída será desvinculada do mesmo.

Figura 5.7 – Tela Ilustração da Consulta a Classes Semelhantes



Fonte: Elaborada pelo autor

Após escolher todas as entidades que irão ser utilizadas no seu sistema, o usuário é redirecionado para a página `AtributosEntidades.aspx`, e todas as entidades descritas pelo usuário são adicionadas ao projeto. Caso a entidade não exista no sistema, isto é, caso não exista nenhuma entidade que seja igual à entidade criada, seguindo as regras delimitadas

pelo axioma de igualdade, a entidade é criada e persistida na base de dados, caso a entidade já exista ela é somente anexada ao projeto, como delimitado pelo BOProjeto.

A tela `AtributosEntidades.aspx` é bem similar a anterior, como ilustra a Fig. 5.8, onde o usuário tem o poder de descrever todos os atributos de todas as entidades do sistema, preenchendo todas as propriedades definidas na ontologia, para então armazená-los no PM e a partir daí gerar o código do sistema. Essa é a última etapa da criação de um projeto e uma das mais importantes pois todo o sistema funcionará de acordo com os atributos das entidades, por isso todos os atributos são obrigatórios e devem ser preenchidos com muito cuidado pelo usuário.

Figura 5.8 – Tela Ilustração a AtributosEntidades.aspx

Fonte: Elaborada pelo autor

Escolhidos os atributos, o usuário é enviado a uma tela que exibe o resumo do projeto e a opção de escolher uma linguagem e os SGBD que o sistema gerado será escrito, como exibido na Fig. 5.9. Após realizar essa escolha o usuário pode então selecionar o botão Gerar Código, o que permite ao PaMDA começar a escrever o sistema gerado através do conjunto de geradores descritos na arquitetura do sistema.

Figura 5.9 – Tela de Resumo de Projeto

Fonte: Elaborada pelo autor

Ao serem invocados, os geradores começam suas interações até obterem todos os arquivos necessários para que o sistema gerado funcione, incluindo todas as classes do sistema gerado, todos os arquivos HTML e o *script* do SGBD escolhido. A Fig. 5.10 exibe os arquivos gerados pelo sistema.

Figura 5.10 – Arquivos Gerados

Nome	Data de modificaç...	Tipo	Tamanho
DAO	24/02/2011 15:56	Pasta de arquivos	
View	04/03/2011 15:00	Pasta de arquivos	
VO	24/02/2011 15:56	Pasta de arquivos	
BD.sql	28/02/2011 17:58	Arquivo SQL	1 KB

Fonte: Elaborada pelo autor

Após todos os arquivos serem gerados, o usuário poderá publicá-los em um servidor que suporte a linguagem PHP e o SGBD MySQL para utilizá-lo, ou pode ainda adicionar todas as funcionalidades que queira, seguindo os padrões de desenvolvimento já existentes no código gerado.

5.2 Resumo do capítulo

Neste capítulo foi explicado o estudo de caso Projeto PaMDA que utiliza toda a arquitetura do *framework* proposto para modelar um sistema gerador de código com caracteristicas semânticas que funciona sob a web e que qualquer usuário possa utiliza-lo.

No capítulo seguinte serão destacadas as considerações finais sobre esse trabalho.

6 Conclusão

Neste capítulo apresentam-se as considerações finais sobre o trabalho desenvolvido nesta monografia. Na seção 6.1 são apresentas as considerações finais sobre o trabalho. Na seção 6.2 são descritas as contribuições desta monografia e na seção 6.3 algumas propostas para trabalhos futuros.

6.1 Considerações finais

A arquitetura proposta neste trabalho gera um *framework* robusto e totalmente desacoplável para a criação de sistemas, sempre utilizando como base um motor semântico que auxilia o usuário na definição do seu sistema, baseando sua inferência nos dados captados de todos os projetos já realizados, e sua estrutura permite sua expansão de forma prática, e seu uso é facilitado por uma interface visual, o Projeto PaMDA, disponível na WEB, de forma que qualquer um com acesso a internet possa fazer uso da arquitetura, e qualquer pessoa com conhecimento em programação orientada a objetos possam realizar as alterações necessárias para adaptar a arquitetura proposta a seu próprio uso.

A utilização do *framework* proposto permite uma otimização no reuso de componentes, técnica bastante utilizada em Fábricas de *Softwar*e para agilizar seu processo produtivo, além de permitir uma captação de dados mais apurada através do uso do motor semântico do sistema, facilitando ainda mais a criação de operações consideradas básicas em sistemas de informação.

6.2 Constribuições deste trabalho

Como contribuição principal deste trabalho estão a definição de uma ontologia genérica para a modelagem de *softwares*, a qual é baseada toda a estruturação lógica e semântica do sistema, a integração de uma ontologia com técnicas de MDA, uma arquitetura implementada em camadas que permite a construção de sistemas, algoritmos para modelagem

de sistemas utilizando a ontologia, além da criação de um modelo relacional baseado também na ontologia.

A arquitetura geradora está desenvolvida de forma que possa ser adaptada, acoplada ou desligada caso necessário, fazendo deste um trabalho descritivo de uma arquitetura independente e que utiliza padrões de desenvolvimento semântico para automatizar algumas tarefas como a normalização de dados e entidades e validação semântica de conteúdo.

6.3 Proposta para trabalhos futuros

Como continuidade deste trabalho, a partir da arquitetura proposta, poderá ser desenvolvida uma nova versão de geradores a serem utilizados no desenvolvimento de sistemas em outras linguagens e/ou SGBDs ou a utilização de arquiteturas personalizadas para o sistema gerado.

O aprimoramento do motor semântico pode ser utilizado como um campo de pesquisa amplo, de forma a possibilitar a utilização na geração de conteúdo a partir da descrição de alto nível dada pelo usuário utilizando técnicas de processamento de linguagem natural.

REFERÊNCIAS BIBLIOGRÁFICAS

ACKOF, R. L. **From Data to Wisdom.** In: Journal of Applies Systems Analysis, v. 16, p. 3-9, 1989.

ADAPTIVEGLUE. **Adaptive Glue.** Disponível em: <http://getglue.com/>. Acesso em: 10 mai. 2011.

ALMEIDA, M. B.; BAX, M. P. **Uma visão geral sobre ontologias: pesquisa sobre definições, tipos, aplicações, métodos de avaliação e de construção**. In: Ciência da Informação, Brasília, v. 32, n. 3, 2003.

AMAZON. **Amazon.com**. 2011 Disponível em: <http://www.amazon.com>. Acesso em: 15 jun. 2011.

BERNARAS, A.; LARESGOITI, I.; CORERA, J. **Building and reusing ontologies for electrical network applications**. In: Proceedings of the European Conference on Artificial Intelligence, 1996.

BERNES-LEE, TIM. **Weaving The Web,** Haper San Francisco. São Fransico, 2001.

BLACKBURN, S.; MARCONDES, A. **Dicionário Oxford de Filosofia.** Tradução D. Murcho et al. Rio de Janeiro: Jorge Zahar, 1997.

BORST, W. N. **Construction of engineering ontologies.** Tese de Doutorado, Enschede, Países Baixos, 1997.

CHAUVEL, F.; JÉZÉQUEL, JEAN-MARC. **Code generation from UML Models with semantic variation points**. Trabalho de Conlcusão de Curso, Université Rennes, 2005.

CORCHO, O.; FERNÁNDEZ, M.; GÓMEZ-PÉREZ, A.; LÓPEZ-CIMA, A. **Building Legal Ontologies with METHONTOLOGY and WebODE**, *Springer-Verlag,* 2005.

DESTRO, D. **Implementando Design Patterns com Java**. Disponível em: <http://www.guj.com.br/java.artigo.137.2.guj> . Acesso em: 19 jun. 2010

DING, L.; FINNIN, T.; JOSHI, A.; PENG, Y.; **Swoogle: A Semantic Web Search Metadata Engine**. University of Maryland Baltimore County, 2005.

DINUCCI, D. **Fragmented Future.** In: *IBM Print,* p. 53-54, 1999.

DOMÍNGUEZ, K.; PEREZ, M.; MENDOZA, L.; GRIMANN, A. **Hacia una Ontologia para Fabricas de *Software*,** Universidad Simon Bolivar, Caracas, 2009.

GAMMA, E., HELM, R.; JOHNSON, R.; JOHN, V. **Design Patterns**: **Elements of Reusable Object-Oriented Software**. 1994.

GRUNINGER, M.; FOX, M. S. **Methodology for the design and evaluation of ontologies.** Disponível em: <http://citeseer.ist.psu.edu/grninger95methodology.html>. Acesso em: 10 mai 2011.

FERNÁNDEZ-LÓPEZ, M. **Building a chemical ontology using methotology and the ontology design environment.** IEEE Intelligent Systems & their Applications, p. 37-46, 1999.

GOUVEIA, **Arquitectura Semântica para a Integração de Sistemas no Domínio do Turismo,** Monografia, Universidade da Madeira, 2007.

JUNIOR, P. J. **Padrões de Projeto em Java - Reutilizando o projeto de *softwar*e. Mundo Java**, 6. ed. 2004.

LASSILA, O.; SWICK, R. **Resource Description Framework (RDF) model and syntax specification.** Disponível em: <http://www.w3.org/TR/REC-rdf-syntax>. Acesso em: 01 set. 2010.

LERDOF, R. **PHP Pocket References.** 1. ed. O'Reilly Media, Inc, 2000. 120p.

OMG. **Model Driven Architecture**. Disponível em: <http://www.omg.org/mda/>. Acesso em: 15 abr. 2011.

PERÉZ, A. G.; CORCHO, O. **Ontology Languages for the Semantic Web**. IEEE Inteligent Systems. v. 13, p. 54-60, 2002.

PLATÃO, **A República**. Livro VII.

REED, S.L.; LENAT, D.B. **Mapping Ontologies into Cyc.** Disponível em: <http://www.cyc.com/doc/white_papers/mapping-ontologies-into-cyc_v31.pdf>. Acesso em: 20 mai 2011.

RHIND-TUTT, S. **Between Now and 2020**, Libraries Should - ALCTS Midwinter Symposium, 2010.

SILVA, D. L.; SOUZA, R. R.; ALMEIDA, M. B. **Ontologias e vocabulários controlados**: **comparação de metodologias para construção**. In: Ciência da Informação [online], v. 37, n. 3, p. 60-75, 2008.

SPIVAK, N. **Web Illustration**. Disponível em: <http://www.concinnity.dk/web-2-0/nova-spivack-web-0-overview-illustration/>. Acesso em: 01 jun. 2011.

SWARTOUT, B.; RAMESH, R.; KNIGHT, K.; RUSS, T. **Toward distributed use of large-scale ontologies**. In: Proceedings of aaai97 spring symposium series workshop on ontological engineering, 1997.

THE WORLD FACTBOOK. **CIA World FactBook.** Disponível em: <https://www.cia.gov/library/publications/the-world-factbook/>. Acesso em: 02 jun. 2011.

Total Code Generator. **Total Code Generator.** Disponível em: <http://www.totalcodegenerator.com.br/total/home.aspx>. Acesso em: 15 jun. 2011.

USCHOLD, M.; KING, M. **Towards a Methodology for Building Ontologies**. 1995. Disponível em: <http://citeseer.ist.psu.edu/uschold95toward.html> Acesso em: 18 mai. 2011

W3C. **Architecture of the world wide web, volume one**. Disponível em: <http://www.w3.org/tr/2004/rec-webarch-20041215/>. Acesso em: 23 mai. 2010.

W3C. **OWL Web Ontology Language Guide**. Disponível em: <http://www.w3.org/TR/owl-guide>. Acesso em: 02 set. 2010.

WATSON, A. **MDA Guide Version.** Disponível em: <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>. Acesso em: 30 mai. 2011.

WIKIPEDIA. **Wikipedia.** Disponível em: <http://en.wikipedia.org/>. Acesso em: 10 mai. 2011.

ZHANG, Y.; VASCONCELOS, W.; SLEEMAN, D. **OntoSearch: An Ontology Search Engine**. University of Aberdeen, 2005.

ANEXO A – PROJETO PAMDA

O projeto PaMDA consiste em um sistema WEB que possibilita a criação de sistemas de informação capazes de manter dados a partir de uma descrição dada pelo usuário utilizando um motor de inferência semântica para otimizar o processo de captação de dados e técnicas de *Model Driven Architecture* para gerar um código limpo, independente e de fácil compreensão.

Este anexo contém um CD-ROM com código fonte do projeto PaMDA assim como screen shot vídeo demonstrando sua utilização .