UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FRAMEWORK PARA MAPEAMENTO OBJETO-RELACIONAL EM DELPHI

SAMUEL YURI DESCHAMPS

BLUMENAU
2010

2010/2-26

SAMUEL YURI DESCHAMPS

FRAMEWORK PARA MAPEAMENTO OBJETO-RELACIONAL EM DELPHI

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Jacques Robert Heckmann, Mestre - Orientador

BLUMENAU
2010

2010/2-26

FRAMEWORK PARA MAPEAMENTO OBJETO-RELACIONAL EM DELPHI

Por

SAMUEL YURI DESCHAMPS

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Jacques Robert Heckmann, Mestre – Orientador, FURB
Membro:	Prof. Adilson Vahldick, Mestre – FURB
Membro:	Prof. Marcos Rogério Cardoso, Especialista – FURB

Blumenau, 7 de dezembro de 2010

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, por de dar vida, força, saúde e disposição.

À minha família pelo apoio e incentivo.

Aos meus amigos mais próximos: André L. B. Rosa, João R. Rodrigues, Leandro da Cunha, Maicon R. Zatelli e Victor A. Muller, que me acompanharam desde o início do curso.

Aos amigos que de alguma forma me ajudaram, e gostaria de destacá-los aqui: André W. P. Hildinger, Fabiano Oss e Kelvin R. Stinghen.

À minha namorada, Paula G. Becher, pela compreensão e ajuda na revisão da monografia.

Ao meu orientador, Jacques R. Heckmann, por acreditar na minha capacidade e ser meu guia na realização deste trabalho.

Qualquer tolo poderia escrever código que um computador consegue entender. Bons programadores escrevem código que humanos conseguem entender.

Martin Fowler

RESUMO

Este trabalho apresenta o desenvolvimento de um *framework* de persistência de objetos que utiliza a técnica de Mapeamento Objeto-Relacional. Tal *framework* será útil na construção de sistemas em Delphi 2010 para Windows que interagem com bancos de dados Firebird ou MySQL, expansível a outros SGBDRs. A persistência é realizada a partir das classes de entidades da aplicação mapeadas através de anotações, uma abordagem mais eficaz que os mapeamentos baseados em arquivos XML. O *framework* utiliza recursos como RTTI, *Custom Attributes, Nullable* e técnicas de persistência especiais para Mapeamento Objeto-Relacional como OID, mapeamento de herança e de associações. Além disso, faz uso de padrões de projeto gerais e específicos, como *Lazy Load*. São explicados todos estes recursos, bem como a especificação do projeto com diagramas da UML e alguns detalhes de implementação. Ao final, são mostrados exemplos de utilização do *framework* para desenvolvimento de um sistema.

Palavras-chave: *Framework*. Delphi. Persistência. Banco de dados. Mapeamento objeto-relacional.

ABSTRACT

This work presents the development of an object persistence framework which uses the Object-Relational Mapping technique. Such framework will be useful to develop information systems in Delphi 2010 environment for Windows witch interact with Firebird or MySQL database systems, extensible to other RDBMSs. The persistence is realized from aplication's entity classes mapped by annotations. This is a more efficacious approach than XML-based mappings. The framework uses resources like RTTI, Custom Attributes, Nullable, Object-Relational Mapping special techniques like OID, inheritance mapping and association mapping. Also uses both general and specific design patterns, like Lazy Load. This document explains all these techniques, as well as the project specification with UML diagrams and some implementation details. At the end, there are shown examples of using the framework to develop a system.

Key-words: Framework. Delphi. Persistence. Database. Object-relational mapping.

LISTA DE ILUSTRAÇÕES

Figura 1 – Encaixe do ORM entre o paradigma relacional e o orientado a objetos 20
Figura 2 – Herança de tabela por classe concreta 21
Figura 3 – Herança de tabela única 23
Figura 4 – Herança de tabela por subclasse 24
Quadro 1 – Declaração de Atributos Customizados 27
Quadro 2 – Anotações com Atributos Customizados 27
Quadro 3 – Construtores em Atributos Customizados 27
Quadro 4 – Utilização dos construtores nos Atributos Customizados 28
Quadro 5 – Extração de Atributos Customizados 28
Figura 5 – Padrão *Identity Field* 31
Figura 6 – Padrão *Identity Map* 35
Quadro 6 – Implementação de métodos do *Identity Map* em Java 36
Quadro 7 – Implementação de métodos do *Identity Map* em Delphi 36
Figura 7 – Padrão *Strategy* 37
Figura 8 – Padrão *Singleton* 39
Figura 9 – Padrão *Adapter* por objeto 41
Figura 10 – Padrão *Adapter* por classe 42
Figura 11 – Padrão *Foreign Key Mapping* 42
Figura 12 – Padrão *Foreign Key Mapping* com coleções 43
Figura 13 – Padrão *Lazy Load* 46
Figura 14 – A comunicação dentro do MVC 50
Figura 15 – Estados dos objetos e suas transações no Hibernate 51
Figura 16 – Diagrama de casos de uso 55
Quadro 8 – Caso de uso `UC01 - Construir base de dados` 56
Quadro 9 – Caso de uso `UC02 - Persistir objeto de entidade` 56
Quadro 10 – Caso de uso `UC03 - Combinar objeto de entidade` 57
Quadro 11 – Caso de uso `UC04 - Atualizar objeto de entidade persistido` 58
Quadro 12 – Caso de uso `UC05 - Obter objeto de entidade` 58
Quadro 13 – Caso de uso `UC06 - Obter lista de objetos de entidades` 59

Quadro 14 – Caso de uso UC07 - Remover objetos de entidades 60
Figura 17 – Diagrama de componentes 61
Figura 18 – Diagrama de classes do gerenciador de objetos 61
Figura 19 – Diagrama de classes dos aplicadores de comandos 63
Figura 20 – Diagrama de classes dos geradores de SQL 65
Figura 21 – Diagrama de classes do mediador de conexão 67
Figura 22 – Diagrama de classes do explorador de metadados 68
Figura 23 – Diagrama de seqüência do *framework* para o método Persist 71
Figura 24 – Classes do modelo de anotação 72
Quadro 15 – Exemplo de anotações em classe de entidade 76
Quadro 16 – Implementação do método Persist 80
Quadro 17 – Implementação do método GetChangerMembers 81
Quadro 18 – Implementação do método Delete 82
Quadro 19 – Implementação do método GenerateInsert 83
Quadro 20 – Classes de comandos nos geradores de SQL 84
Quadro 21 – Classes auxiliares nos geradores de SQL 85
Quadro 22 – Implementação da estrutura Nullable 88
Quadro 23 – Implementação da estrutura Proxy 89
Quadro 24 – Implementação do *Lazy Load* no *Proxy* 90
Figura 25 – Diagrama de classes do estudo de caso 91
Figura 26 – Criação de novo projeto no Delphi 91
Figura 27 – Implementação da classe TArtista 92
Figura 28 – Classe TArtista mapeada no *framework* 93
Figura 29 – Classe TObra mapeada no *framework* 94
Figura 30 – Implementação dos métodos de acesso da classe TObra 95
Figura 31 – Classe TAlbum mapeada no *framework* 96
Figura 32 – Classes TMusica e TClipe mapeadas no *framework* 97
Figura 33 – Tela de cadastro de músicas 98
Figura 34 – Implementação do carregamento dos componentes da tela 98
Figura 35 – Implementação do botão de salvar música 99
Figura 36 – Implementação do método para salvar música no controlador 100
Figura 37 – Configuração do objeto de conexão 101
Figura 38 – Exemplos de alteração e exclusão de objetos 102

Figura 39 – Construção da base de dados 103
Quadro 25 – Implementação das entidades utilizadas nos testes 104
Quadro 26 – SQL de construção da base de dados 105
Quadro 27 – Implementação do teste de persistência de objetos 106
Quadro 28 – Código SQL gerado para persistência de objetos 106
Quadro 29 – Implementação do teste de obtenção de objetos 107
Quadro 30 – Código SQL gerado para obtenção de objetos 107
Quadro 31 - Comparação entre características e funcionalidades dos *frameworks* 110

LISTA DE TABELAS

Tabela 1 – Medição de tempos de persistência e obtenção de objetos.................................. 108

LISTA DE SIGLAS

API - *Application Programming Interface*

BDE - Borland *Database Engine*

BLOB - *Binary Large OBject*

CPF - Cadastro de Pessoa Física

DAO - *Data Access Object*

DDL - *Data Definition Language*

DML - *Data Manipulation Language*

GUID - *Global Unique IDentifier*

HQL - Hibernate *Query Language*

IDE - *Integrated Development Environment*

IQL - *Instant Query Language*

JPA - Java *Persistence API*

MVC - *Model View Controller*

OID - *Object IDentifier*

OPF - *Object Persistence Framework*

ORM - *Object-Relational Mapping*

RAD – *Rapid Application Development*

RDBMS - *Relational DataBase Management System*

RMI - *Remote Method Invocation*

RTL - *RunTime Library*

RTTI - *RunTime Type Information*

SGBD - Sistema Gerenciador de Bancos de Dados

SGBDR - Sistema Gerenciador de Bancos de Dados Relacional

SQL - *Structured Query Language*

UML - *Unified Modeling Language*

VCL - *Visual Component Library*

VMT - *Virtual Method Table*

XML - *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO 16
1.1 OBJETIVOS DO TRABALHO 17
1.2 ESTRUTURA DO TRABALHO 17
2 FUNDAMENTAÇÃO TEÓRICA 18
2.1 O PAPEL DO *FRAMEWORK* 18
2.2 MAPEAMENTO OBJETO-RELACIONAL 19
2.2.1 Mapeamento de herança 21
2.2.1.1 Herança de tabela por classe concreta 21
2.2.1.2 Herança de tabela única 22
2.2.1.3 Herança de tabela por subclasse 23
2.3 RUNTIME TYPE INFORMATION (RTTI) 25
2.4 ATRIBUTOS CUSTOMIZADOS 26
2.4.1 Declaração de Atributos Customizados 26
2.4.2 Extração de Atributos Customizados 28
2.5 TIPOS ANULÁVEIS 29
2.6 PADRÕES DE PROJETO 30
2.6.1 Padrão *Data Access Object* (DAO) 30
2.6.2 Padrão *Identity Field* 30
2.6.2.1 Chaves com ou sem significado 31
2.6.2.2 Chaves simples *versus* chaves compostas 32
2.6.2.3 O tipo da chave 32
2.6.2.4 Obtendo uma nova chave 32
2.6.2.5 Trabalhando com herança 34
2.6.3 Padrão *Identity Map* 34
2.6.3.1 Funcionamento do Mapa de Identidade 34
2.6.4 Padrão *Strategy* 36
2.6.5 Padrão *Singleton* 38
2.6.6 Padrão *Adapter* 40
2.6.7 Padrão *Foreign Key Mapping* 42
2.6.7.1 Mapeamento de coleções 43
2.6.7.2 Tratamento de ciclos 45

2.6.7.3 Onde não utilizar 45
2.6.8 Padrão *Lazy Load* 46
2.6.8.1 Inicialização tardia 47
2.6.8.2 *Proxy* virtual 47
2.6.8.3 Armazenador de valor 48
2.6.8.4 Fantasma 48
2.6.8.5 Programação orientada a aspectos 48
2.6.8.6 Escolhendo a melhor estratégia 49
2.6.9 Padrão *Model View Controller* (MVC) 49
2.7 TRABALHOS CORRELATOS 50
2.7.1 Hibernate 50
2.7.2 Instant Objects 51
2.7.3 Ferramenta para aplicação do DAO em sistemas desenvolvidos em Delphi 52
3 DESENVOLVIMENTO DO *FRAMEWORK* 53
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 53
3.2 ESPECIFICAÇÃO 54
3.2.1 Comportamento do *framework* 54
3.2.1.1 UC01 – Construir base de dados 55
3.2.1.2 UC02 – Persistir objeto de entidade 56
3.2.1.3 UC03 – Combinar objeto de entidade 57
3.2.1.4 UC04 - Atualizar objeto de entidade persistido 57
3.2.1.5 UC05 – Obter objeto de entidade 58
3.2.1.6 UC06 – Obter lista de objetos de entidades 59
3.2.1.7 UC07 – Remover objeto de entidade 59
3.2.2 Estrutura do *framework* 60
3.2.2.1 Gerenciador de objetos 61
3.2.2.2 Aplicadores de comandos 62
3.2.2.3 Geradores de SQL 64
3.2.2.4 Mediador de conexão 66
3.2.2.5 Explorador de metadados 68
3.2.3 Interação entre as classes 70
3.3 MODELO DE ANOTAÇÃO PARA MAPEAMENTO 71
3.4 IMPLEMENTAÇÃO 77
3.4.1 Técnicas e ferramentas utilizadas 77

3.4.2 Convenções seguidas 78
3.4.3 Implementação dos componentes do *framework* 79
3.4.4 Funcionalidades e estratégias implementadas.. 85
3.4.4.1 Mapeamento de herança 85
3.4.4.2 Mapeamento de associações 86
3.4.4.3 *Object Identifier* 87
3.4.4.4 Nullable.. 87
3.4.4.5 Lazy Load.. 89
3.4.5 Operacionalidade da implementação 90
3.5 RESULTADOS E DISCUSSÃO 103
3.5.1 Análise do SQL gerado 103
3.5.1.1 Teste 1 – Construção da base de dados... 104
3.5.1.2 Teste 2 – Persistência de objetos 105
3.5.1.3 Teste 3 – Obtenção de objetos 107
3.5.2 Análise de desempenho.. 108
3.5.3 Comparações com trabalhos correlatos.. 109
4 CONCLUSÕES.. 113
4.1 EXTENSÕES 114
REFERÊNCIAS BIBLIOGRÁFICAS 116

1 INTRODUÇÃO

Com o aumento da quantidade de sistemas de informação sendo construídos nos últimos anos, surgiram *frameworks*[1] para facilitar e organizar o seu desenvolvimento. O seu objetivo é abstrair a implementação das partes mais repetitivas da construção, que normalmente são comuns a qualquer sistema.

Em um sistema de informação que utilize um Sistema Gerenciador de Bancos de Dados (SGBD) para persistir as informações, uma parte do desenvolvimento que normalmente se torna repetitiva é a implementação de rotinas que interagem com o SGBD. Em especial, quando se trata de um Sistema Gerenciador de Bancos de Dados Relacional (SGBDR), esta parte geralmente é implementada utilizando a *Structured Query Language* (SQL), que é embutida no código da linguagem de programação do sistema.

Esta prática pode tornar-se pouco produtiva, pois há a necessidade de se trabalhar ao mesmo tempo com duas linguagens de diferentes paradigmas: o da orientação a objetos e o relacional. Não é interessante preocupar-se com detalhes específicos do banco de dados e ao mesmo tempo preocupar-se com regras de negócio da aplicação. Além disso, o código SQL é validado somente em tempo de execução, aumentando a possibilidade de *bugs*.

O sistema pode ser organizado em camadas para separar os diferentes detalhes de implementação. Uma delas seria a camada de persistência, que se encarregaria de resolver todos os problemas relativos à geração de SQL, conexão com o banco e detalhes específicos do SGBD. As responsabilidades desta camada podem ser delegadas a um *framework*.

Frente a isso, o presente trabalho propõe o desenvolvimento de um *Object Persistence Framework* (OPF) que utilize a técnica de *Object-Relational Mapping* (ORM). Com a sua utilização, o programador será poupado da tarefa de implementar comandos em SQL para ler e gravar informações no banco de dados. O *framework* será voltado para o desenvolvimento de sistemas orientados a objetos em Delphi para Windows, linguagem e plataforma nas quais será implementado. Esta escolha foi tomada tendo em vista que é a linguagem de maior domínio do autor há anos, conhecendo também empresas da região de Blumenau que a utilizam. Para justificar a relevância desta linguagem, Tiobe Software (2010) aponta Delphi entre as dez linguagens mais populares no ano de 2010.

---

[1] "Um *framework* é um conjunto de classes cooperantes que constroem um projeto reutilizável para uma determinada categoria de software." (GAMMA et al., 2000, p. 41).

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um OPF em Delphi que utilize a técnica de ORM para fornecer a camada de persistência a uma aplicação.

Os objetivos específicos do trabalho são:

a) definir um modelo de anotação específica para o ORM em Delphi;
b) implementar os componentes que irão compor o motor de persistência, que será responsável por gerar comandos `select`, `insert`, `update` e `delete` na linguagem SQL e executá-los para ler ou gravar objetos de entidades no banco de dados;
c) implementar os componentes que irão compor a infraestrutura de manipulação dos objetos de entidades na aplicação, controlando o seu ciclo de vida. Tais componentes deverão permitir manter o *cache* dos objetos em memória local para otimização de desempenho;
d) fazer com que o *framework* suporte alguns SGBDs de mercado, a saber, os SGBDs MySQL e Firebird.

1.2 ESTRUTURA DO TRABALHO

O trabalho divide-se em quatro capítulos. No segundo capítulo é apresentada a fundamentação teórica necessária para o entendimento do trabalho, onde são mencionadas as principais características de um *framework* de mapeamento objeto-relacional, também os conceitos, técnicas e padrões de projeto que podem ser utilizados no seu desenvolvimento. Também nesse capítulo, em trabalhos correlatos, serão apresentados dois *frameworks* de mercado com funcionalidades semelhantes ao *framework* desenvolvido neste trabalho e também uma ferramenta desenvolvida em trabalho de conclusão de curso relacionada com o tema. No terceiro capítulo será exposto o desenvolvimento do *framework*. Nesse capítulo são listados os requisitos que o mesmo se propõe a atender e a especificação do trabalho, bem como os detalhes de implementação e utilização de padrões. Por fim, no quarto capítulo são mostradas as conclusões e as extensões sugeridas para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo é apresentada a fundamentação teórica que foi tomada como base para o desenvolvimento deste trabalho, para que se possa entender como o *framework* foi desenvolvido. Em primeiro lugar será comentado qual o papel de um *framework*. Em seguida será apresentada a técnica de mapeamento objeto-relacional, explicando algumas diferenças entre o paradigma da orientação a objetos e o paradigma relacional. Logo após, serão abordados alguns recursos da linguagem Delphi que foram utilizados, como *RunTime Type Information* (RTTI) e Atributos Customizados. Também será abordado o conceito de Tipos Anuláveis, recurso que foi implementado como parte do desenvolvimento deste trabalho para ser utilizado nas classes de entidades mapeadas. Em seguida, serão apresentados todos os padrões de projeto que foram utilizados no desenvolvimento do *framework*. Ao final, serão apresentados os trabalhos correlatos e suas características.

2.1 O PAPEL DO *FRAMEWORK*

Gamma et al. (2000, p. 41-42) esclarece que o *framework* dita a arquitetura de uma aplicação. É o *framework* que define sua estrutura geral, sua divisão em classes e objetos e em conseqüência as responsabilidades-chave das classes e objetos, como estas colaboram, e o fluxo de controle. Um *framework* predefine estes parâmetros de projeto, de maneira que o projetista ou programador da aplicação possa concentrar-se nos aspectos específicos da aplicação.

Um *framework* disponibiliza um conjunto de componentes ou classes para serem utilizadas ou estendidas por uma aplicação. Neste último caso, o *framework* é customizado para uma aplicação específica através da criação de subclasses para a aplicação, derivadas das classes abstratas do *framework*. Ao fazer isso, o programador reutiliza o corpo principal e escreve o código que este chama, escrevendo operações com nomes e convenções de chamada já especificadas. Isso reduz as decisões de projeto que será preciso tomar. Como resultado, é possível não somente construir aplicações mais rapidamente, como também construí-las em estruturas similares. Tais aplicações serão mais fáceis de manter e parecem mais consistentes para seus usuários. Por outro lado, perde-se alguma liberdade criativa, uma vez que muitas

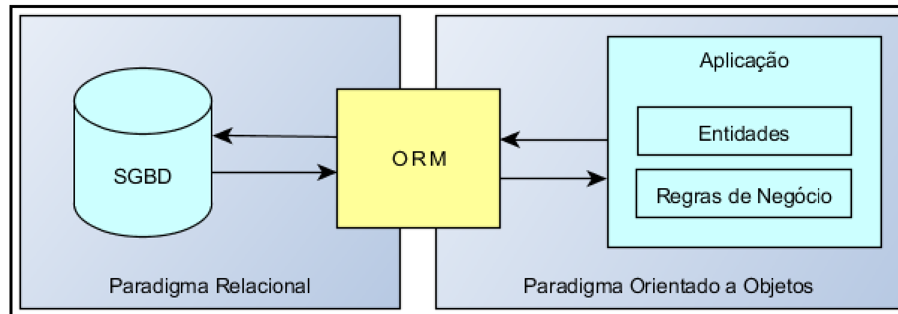
decisões de projeto já terão sido tomadas (GAMMA et al., 2000, p. 41-42).

2.2 MAPEAMENTO OBJETO-RELACIONAL

Bernardi (2006, p. 46-51) explica que apesar de o paradigma orientado a objetos estar sendo cada vez mais difundido no processo de desenvolvimento de *software*, não existem hoje soluções comerciais robustas e amplamente aceitas neste paradigma para a persistência de dados. Com o surgimento de bancos de dados pós-relacionais, orientados a objetos e outras alternativas de persistência de objetos, este cenário pode mudar, mas atualmente o mercado de aplicações comerciais ainda é dominado pelos SGBDs relacionais.

Segundo Fowler (2006, p. 170), não existe um encaixe perfeito entre os dois paradigmas. Objetos e bancos de dados relacionais utilizam mecanismos diferentes para estruturar os dados. Muitas partes integrantes de um objeto, tais como coleções e herança, não estão presentes em bancos de dados relacionais. Além disso, enquanto a orientação a objetos é baseada em princípios da engenharia de *software* onde os objetos são abstrações do mundo real, o paradigma relacional é baseado em princípios matemáticos, onde as informações de uma base de dados são consideradas relações matemáticas e estão representadas de maneira uniforme com o uso de tabelas bidimensionais. Quando se cria um modelo de objetos com muita lógica de negócio, o uso de um mecanismo de ORM é valioso para melhor organizar os dados e o comportamento associado.

“O mapeador de dados é uma camada de *software* que separa os objetos na memória do banco de dados. Sua responsabilidade é transferir dados entre os dois e também isolá-los um do outro.” (FOWLER, 2006, p. 170). Com o mapeador de dados, os objetos na memória não precisam nem mesmo saber que existe um banco de dados. Eles não precisam conter comandos SQL e certamente não precisam ter nenhum conhecimento do esquema do banco de dados. A Figura 1 mostra como o ORM encaixa-se entre os dois paradigmas.



Fonte: adaptado de Bernardi (2006, p. 48).

Figura 1 – Encaixe do ORM entre o paradigma relacional e o orientado a objetos

Um *framework* de persistência de objetos permite a realização do armazenamento e manutenção do estado dos objetos em algum meio não-volátil, como um banco de dados, de forma transparente. Uma de suas vantagens é que o analista/programador pode trabalhar como se estivesse em um sistema completamente orientado a objetos. Outra vantagem é a centralização do canal de acesso ao SGBD em um só ponto, inacessível ao programador, tornando a aplicação e o SGBD desacopláveis e dando a possibilidade de troca do SGBD ou alterações na base de dados sem muito transtorno (BERNARDI, 2006, p. 46-51).

Segundo Bernardi (2006, p. 46-51), como regras de mapeamento, pode-se estabelecer uma analogia entre objetos e tabelas, e entre atributos e colunas. Ao se realizar uma contraposição entre os dois modelos é importante levar em consideração, com certa flexibilidade, as seguintes regras:

a) todas as tabelas devem ter uma chave primária e, no sistema orientado a objetos, cada objeto deve ser único. Esta unicidade é garantida através da introdução de um *Object IDentifier* (OID), que deve ser gerado pelo sistema, não pode ser alterado pelo usuário e não pode depender de outros atributos do objeto;
b) os objetos podem possuir atributos simples, que são mapeados para apenas uma coluna, atributos compostos, que são mapeados a outra tabela, e atributos multivalorados, que também são mapeados a outra tabela e representam coleções, como por exemplo o atributo `ListaTelefones` de uma classe `Cliente`;
c) a herança entre classes pode ser mapeada de três formas: criar apenas uma tabela para todas as classes da hierarquia, criar uma tabela para cada classe, ou criar uma tabela para cada classe não-abstrata. Na primeira forma é criada uma coluna que identifique, para cada registro da tabela, a classe à qual pertence. Nas duas últimas formas é criada nas classes filhas uma coluna que referencie a tabela pai através de uma chave estrangeira;
d) o mapeamento deve levar em consideração a existência dos tipos de associações

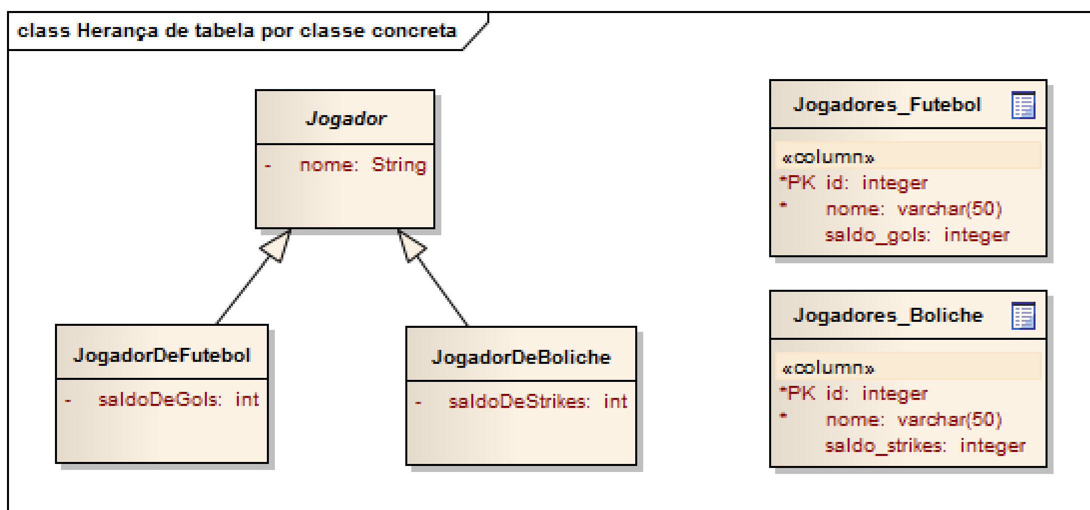
muitos-para-muitos, muitos-para-muitos com classe de associação, um-para-muitos, um-para-muitos com classe de associação e um-para-um.

2.2.1 Mapeamento de herança

Fowler (2006, p. 283) afirma que como os bancos de dados relacionais não suportam herança segundo os conceitos da orientação a objetos, é necessário mapear a herança para o paradigma relacional. A seguir são explicadas com mais detalhes as três estratégias de mapeamento de herança para que se possa entendê-las com maior clareza.

2.2.1.1 Herança de tabela por classe concreta

Segundo Fowler (2006, p. 283), esta estratégia utiliza uma tabela do banco de dados para cada classe concreta da hierarquia. Cada tabela contém colunas para a classe concreta e todos os seus ancestrais, de modo que qualquer campo na classe pai é duplicado pelas tabelas das classes filhas, conforme se pode ver na Figura 2.



Fonte: adaptado de Fowler (2006, p. 283).

Figura 2 – Herança de tabela por classe concreta

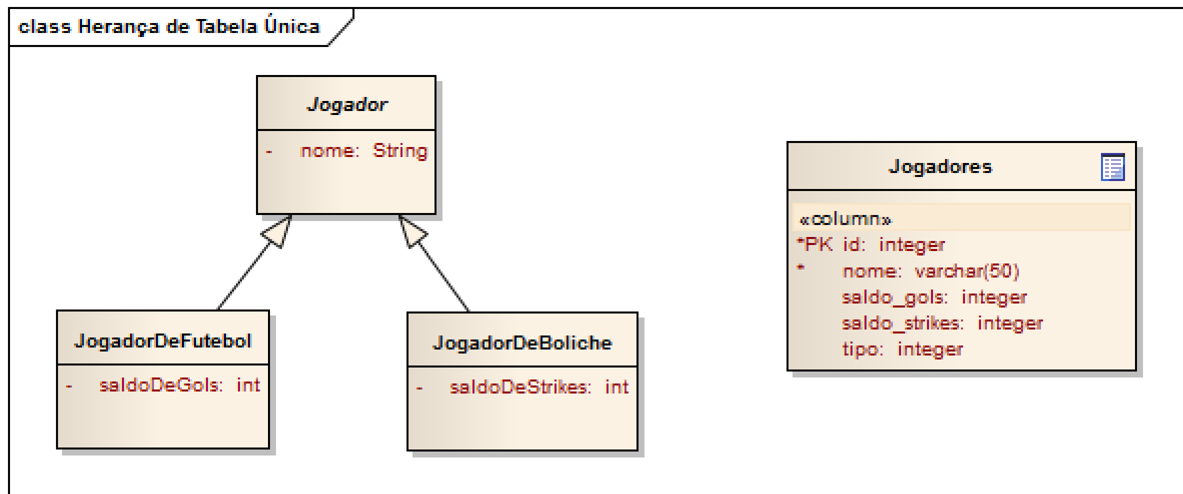
Fowler (2006, p. 285) afirma que uma das vantagens desta estratégia é que cada tabela é auto-contida e não possui campos irrelevantes. A conseqüência é que estas tabelas farão bastante sentido quando utilizadas por outras aplicações que não as estejam mapeando para objetos. Além disso, não há junções a realizar durante a leitura dos dados das classes concretas.

Um dos problemas desta estratégia, conforme Fowler (2006, p. 283), é garantir que a chave primária seja única não apenas para uma tabela, mas para todas as tabelas da hierarquia. Isso deve ser feito para que se possa realizar consultas polimórficas que obtém registros de várias destas tabelas, mas cada objeto tenha um identificador distinto. Por exemplo, uma consulta para retornar um `Jogador` com `id` igual a `10`. Se forem utilizadas *sequences* no banco de dados este problema fica mais fácil de resolver, basta utilizar a mesma *sequence* para gerar as chaves primárias de todas estas tabelas.

Fowler (2006, p. 285) ainda explica que um ponto fraco desta estratégia é a impossibilidade de forçar relacionamentos no banco de dados para classes abstratas. Usando o exemplo citado no diagrama, não seria possível criar uma chave estrangeira no banco de dados que apontasse para um `Jogador`. Alguma lógica de negócio orientada a objetos poderia ter esta necessidade. Outro ponto fraco é a refatoração nas classes superiores. Cada vez que é alterado um campo na classe superior, devem ser alteradas todas as tabelas que contenham este campo. Finalmente, mais uma desvantagem é que cada consulta polimórfica geraria consultas em várias tabelas, levando a múltiplos acessos ao banco de dados. Bauer e King (2007, p. 195) afirmam que este problema pode ser minimizado utilizando *unions*.

2.2.1.2 Herança de tabela única

Conforme Bauer e King (2007, p. 199), uma hierarquia inteira de classes pode ser mapeada a uma única tabela. Esta tabela inclui colunas para todas as *properties* de todas as classes da hierarquia. A subclasse concreta é representada por uma linha particular na tabela que é identificada pelo valor presente em uma coluna especial, o discriminador do tipo da classe, conforme a Figura 3.



Fonte: adaptado de Fowler (2006, p. 269).

Figura 3 – Herança de tabela única

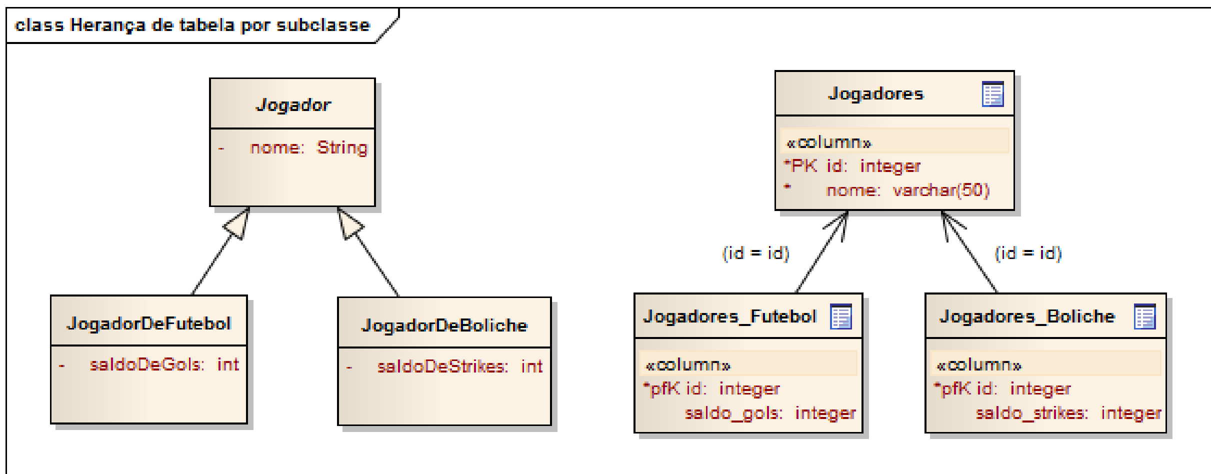
Segundo Fowler (2006, p. 269), ao mapear para um banco de dados relacional, tenta-se minimizar a quantidade de *joins* que pode crescer rapidamente ao processar uma estrutura de herança em diversas tabelas. Bauer e King (2007, p. 199) explicam que esta estratégia de mapeamento é a vencedora em termos de desempenho e simplicidade. Os dados podem ser obtidos sem *joins* nem *unions* complexas. Além disso, é a melhor forma de representar polimorfismo. Tanto consultas polimórficas quanto não-polimórficas funcionam bem. Fowler (2006, p. 270) diz também que a refatoração se torna mais fácil quando é necessário mover campos para cima ou para baixo na hierarquia, pois não requer nenhuma alteração no banco de dados.

Bauer e King (2007, p. 200) afirmam que a maior desvantagem desta estratégia é que as colunas das classes filhas devem ser declaradas como anuláveis. Se cada uma das classes filhas declara vários campos obrigatórios, a perda das construções de `not null` pode ser um problema para a integridade dos dados. Fowler (2006, p. 270) também aponta outros pontos fracos, como desperdício de espaço no banco de dados se a mesma tabela representar uma grande quantidade de classes e resultar em muitos campos com o valor `null`. O desempenho do sistema também pode ser prejudicado se a tabela for grande demais e precisar de muitos índices, ou então problemas relacionados a bloqueios freqüentes nesta tabela.

2.2.1.3 Herança de tabela por subclasse

Conforme Bauer e King (2007, p. 203), esta estratégia representa relacionamentos de herança como chaves estrangeiras no banco de dados. Cada classe ou subclasse que declara

campos persistentes, incluindo classes abstratas, possui sua própria tabela. A Figura 4 exemplifica como ficariam as tabelas no banco de dados.



Fonte: adaptado de Bauer e King (2007, p. 204).

Figura 4 – Herança de tabela por subclasse

Bauer e King (2007, p. 203) explicam que nesta estratégia, a tabela contém campos apenas para as *properties* declaradas na própria classe, pois as *properties* herdadas já se tornaram campos na tabela que representa a classe pai. Além disso, nas tabelas filhas, o mesmo campo que representa a chave primária, é também uma chave estrangeira para a tabela pai.

Segundo Bauer e King (2007, p. 204), a principal vantagem desta estratégia é que o esquema das tabelas é normalizado[2]. A evolução do esquema e a integridade das construções são corretas e o modelo relacional fica bem parecido com o modelo das classes. Atende ao mesmo tempo as convenções do paradigma relacional quanto orientado a objetos. Uma associação de uma classe para a classe pai pode ser representada no banco como uma chave estrangeira para a tabela pai e uma associação para a classe filha pode ser representada como uma chave estrangeira para a tabela filha. Fowler (2006, p. 277) também afirma que todas as colunas são relevantes para todas as linhas, de modo que as tabelas são mais fáceis de compreender e não desperdiçam espaço.

Dentre as desvantagens, segundo Fowler (2006, p. 277), estaria o desempenho das consultas, já que várias tabelas precisam ser acessadas para carregar um único objeto. As tabelas das classes superiores podem se tornar um gargalo, pois elas precisam ser acessadas com mais freqüência. Além disso, quando é necessário refatorar uma classe movendo campos para cima ou para baixo na hierarquia, é necessário alterar mais de uma tabela no banco.

[2] Segundo Heuser (2000), normalização define regras que devem ser obedecidas para que o esquema do banco de dados seja considerado bem projetado. Existe um conjunto numerado de formas normais para verificar tabelas em banco de dados relacionais.

2.3 RUNTIME TYPE INFORMATION (RTTI)

Segundo Martins (2003, p. 6-11), RTTI é um poderoso recurso do Delphi que permite obter informações de tipos em tempo de execução. Os operadores mais básicos do RTTI são o `is` e o `as`, que permitem, respectivamente, testar o tipo de um objeto e efetuar uma conversão em tempo de execução. O RTTI permite extrair informações mais detalhadas como métodos e propriedades de um objeto.

O próprio Delphi faz uso intensivo de RTTI em várias partes da sua *Integrated Development Environment* (IDE), como por exemplo para salvar e carregar dados de arquivos `.dfm`[3] e também nas abas *object inspector*[4] e *structure*[5]. Internamente, o RTTI é organizado em tabelas com dados que podem ser úteis para o desenvolvedor, tais como informações sobre classes, métodos virtuais, dinâmicos e interfaces implementadas. Outro recurso interessante é a *Virtual Method Table* (VMT) que pode ser utilizada em conjunto com RTTI para obter informações mais detalhadas sobre os métodos, como ponteiros para todos os métodos virtuais declarados em uma classe e em suas ancestrais, além de ponteiros para outras tabelas. (MARTINS, 2003, p. 6-11).

No Delphi até a versão 2009, as informações de RTTI estão disponíveis apenas para métodos e *properties* de uma classe que possuem visibilidade *published* ou padrão. Porém Groves (2009) apresenta o RTTI estendido que surge com o Delphi 2010. Nesta versão do Delphi foi feita uma revisão na construção interna do RTTI, o qual agora permite acessar informações de atributos, métodos e *properties* de qualquer visibilidade. Também foi criada uma *Application Programming Interface* (API) mais facilitada para obtenção das informações. Outro recurso interessante que está disponível a partir da versão 2010 são os chamados Atributos Customizados. Trata-se de informações adicionais, customizáveis em forma de anotação, que podem ser atribuídas a classes, atributos, métodos e *properties* em tempo de *design* para serem posteriormente extraídas via RTTI, semelhante ao recurso de *annotations* da linguagem de programação Java. Todos estes recursos citados estão agora disponíveis na linguagem Delphi para plataforma Windows, versão 2010.

---

[3] "Os arquivos `.dfm` no Delphi são arquivos que mantém as informações dos componentes presentes em cada formulário." (SASSE, 2005).

[4] O *object inspector* exibe as propriedades e eventos do componente selecionado na tela em modo *designer* e permite o usuário alterar os valores das propriedades ou atribuir um evento ao objeto (GAJIC, 2006).

[5] A aba *structure* exibe em formato de árvore a hierarquia do código fonte caso o usuário esteja no modo editor de código, ou a hierarquia dos componentes do formulário caso esteja no modo *designer* (GAJIC, 2006).

Conforme Mourão (2009, p. 104-121), o uso da RTTI está quase sempre ligado a *frameworks* que venham a ser desenvolvidos para centralização e otimização dos sistemas. Esses *frameworks* têm por objetivo trazer maior flexibilidade a um sistema e produtividade ao desenvolvedor. Um exemplo seria um *framework* de validação, onde se centraliza na classe a ser validada todo o código responsável por isso, ao invés de replicar por todo sistema uma regra de negócio.

Com nova API do RTTI do Delphi 2010, é possível acessar quais propriedades foram declaradas no objeto que está sendo lido e quais foram herdadas das classes superiores, além da possibilidade de alterar os valores dos atributos e propriedades dos objetos em tempo de execução. Esta informação é de extrema importância para implementações de mapeamento objeto-relacional (MOURÃO, 2009, p. 104-121).

2.4 ATRIBUTOS CUSTOMIZADOS

Conforme Embarcadero (2010), atributos customizados, ou *custom attributes*, são um recurso da linguagem de programação Delphi que permite anotar tipos e membros de tipos com objetos especiais que carregam informações adicionais. Essas informações podem ser consultadas em tempo de execução. Os atributos customizados estendem o modelo normal de orientação a objetos com elementos orientados a aspectos.

Em geral, atributos customizados são úteis para construção de *frameworks* de propósito geral que analisam tipos estruturados como objetos em tempo de execução e introduzem um novo comportamento baseado nas informações adicionais fornecidas pelos atributos anotados. Tais atributos não modificam o comportamento dos tipos ou membros por si sós. O código que os consome precisa consultar especificamente pela existência dos atributos e tomar as ações apropriadas para aplicar o comportamento desejado (EMBARCADERO, 2010).

2.4.1 Declaração de Atributos Customizados

Embarcadero (2010) afirma que um atributo customizado é declarado como uma simples classe. Para declarar o seu próprio atributo customizado, o desenvolvedor precisa

declarar uma classe Delphi derivada de uma classe especial chamada `TCustomAttribute`, localizada na *unit* `System`. As restrições são que uma classe de atributo customizado não pode ser declarada como classe abstrata e não deve conter nenhum método abstrato. O Quadro 1 mostra um exemplo.

```
type
  TMeuAtributo = class(TCustomAttribute)
  end;
```

Fonte: adaptado de Embarcadero (2010).

Quadro 1 – Declaração de Atributos Customizados

Feito isto, a classe `TMeuAtributo` pode ser utilizada para anotar qualquer tipo, como classes, *records* ou interfaces. Também é possível anotar nos membros de tais tipos, como se pode ver no Quadro 2.

```
type
  [TMeuAtributo]
  TIntegerEspecial = type Integer;

  TAlgumaClasse = class
    [TMeuAtributo]
    procedure FazAlgumaCoisa;
  end;
```

Fonte: adaptado de Embarcadero (2010).

Quadro 2 – Anotações com Atributos Customizados

Para carregar o atributo customizado com informações adicionais que possam ser utilizadas em tempo de execução, é necessário declarar construtores para esta classe, conforme mostrado no Quadro 3.

```
type
  TAtributoComConstrutor = class(TCustomAttribute)
  public
    constructor Create(const AlgumTexto: string);
  end;
```

Fonte: adaptado de Embarcadero (2010).

Quadro 3 – Construtores em Atributos Customizados

Declarado o construtor, o atributo pode ser utilizado conforme mostrado no Quadro 4.

```
type
  TAlgumaClasse = class
    [TAtributoComConstrutor('Texto com informações.')]
    procedure FazAlgumaCoisa;
  end;
```

Fonte: adaptado de Embarcadero (2010).

Quadro 4 – Utilização dos construtores nos Atributos Customizados

Conforme Embarcadero (2010), a execução de métodos funciona nos atributos customizados normalmente como outras classes. Isto significa que podem ser definidos vários construtores sobrecarregados na classe do atributo. Porém, como estes construtores serão resolvidos em tempo de compilação, os parâmetros dos construtores necessariamente devem ser valores constantes. Não são aceitos parâmetros do tipo `var` ou `out`.

2.4.2 Extração de Atributos Customizados

Embarcadero (2010) ainda explica que para extrair em tempo de execução as informações que foram anotadas utilizando determinados atributos customizados, o desenvolvedor precisa explicitamente escrever código para consultá-las. Para isto é utilizada a classe `TRTTIContext`, localizada na *unit* `RTTI`. O Quadro 5 apresenta um exemplo de código fonte para realizar tal consulta.

```
var
  LContext: TRttiContext;
  LType: TRttiType;
  LAttr: TCustomAttribute;
begin
  { Instancia um novo contexto do RTTI }
  LContext := TRttiContext.Create

  { Extrai as informações de RTTI da classe TAlgumaClasse }
  LType := LContext.GetType(TAlgumaClasse);

  { Pesquisa pelo atributo e realiza ação especial }
  for LAttr in LType.GetAttributes() do
    if LAttr is TMeuAtributo then
      Writeln(TMeuAtributo(LAttr).InformacaoAdicional);
end;
```

Fonte: adaptado de Embarcadero (2010).

Quadro 5 – Extração de Atributos Customizados

Neste exemplo está sendo considerado que exista na classe `TMeuAtributo` uma propriedade chamada `InformacaoAdicional`, que é preenchida no construtor desta classe

com o valor que foi passado como parâmetro no construtor.

2.5 TIPOS ANULÁVEIS

Conforme MSDN (2006), a estrutura genérica `Nullable<T>` representa um objeto cujo tipo genérico é um tipo primitivo ao qual pode ser atribuído o valor `null`. Neste caso, `T` representa o tipo primitivo encapsulado.

Um tipo é dito como sendo anulável se a ele pode ser atribuído um valor ou uma referência nula indicando a ausência de valor. Conseqüentemente, um tipo anulável pode expressar um valor, ou expressar que nenhum valor existe. Por exemplo, na linguagem C#, um tipo referenciável como `String` é automaticamente anulável, enquanto um tipo primitivo inteiro de 32 bits não é. Um tipo primitivo não pode ser anulável porque ele possui capacidade suficiente para expressar somente os valores apropriados para tal tipo e não possui capacidade adicional para expressar o valor `null` (MSDN, 2006).

Tipos anuláveis são utilizados para representar coisas que podem existir ou não, dependendo da circunstância. Por exemplo, em uma coluna não-nula de uma tabela de banco de dados, o valor pode existir em uma linha e não existir em outra. Suponha-se que a coluna seja representada como um atributo de uma classe, definido como um tipo primitivo. Neste caso, o atributo poderia conter todos os valores válidos para tal coluna, mas não poderia acomodar uma informação adicional para indicar que não existe valor. Portanto, o atributo poderia ser declarado utilizando o tipo `Nullable<T>` ao invés de um tipo primitivo (MSDN, 2006).

A estrutura genérica `Nullable<T>` está disponível no .Net Framework a partir da versão 2.0 e possui duas propriedades principais: `HasValue` e `Value`. Em uma variável `Nullable<T>`, se a propriedade `HasValue` for verdadeira, o valor do objeto pode ser acessado através da propriedade `Value`. Caso contrário, o valor contido é indefinido e qualquer tentativa de acesso à propriedade `Value` dispararia uma exceção (MSDN, 2006).

2.6 PADRÕES DE PROJETO

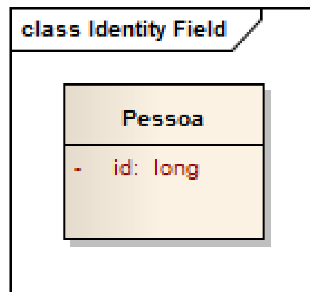
A seguir são apresentados os padrões de projeto que foram utilizados para o desenvolvimento do *framework* ou que são indicados para serem utilizados juntamente com o *framework* para o desenvolvimento de sistemas.

2.6.1 Padrão *Data Access Object* (DAO)

Segundo Trottier (2004, p. 294), o padrão de projeto DAO provê uma conexão entre a camada de regra de negócios e a camada de persistência da aplicação de forma transparente. Com isso é abstraído e encapsulado todo o acesso à fonte de dados, escondendo completamente os detalhes de sua implementação dos componentes de negócio que o utilizam. O DAO pode ser implementado para persistir os dados de várias formas diferentes, como acesso a um SGBD, arquivos *eXtensible Markup Language* (XML) ou outros tipos de persistência, ou até transmitir os dados através de *Remote Method Invocation* (RMI) ou *sockets*, por exemplo. Enquanto isso, a camada de negócio precisa se preocupar somente com a interface do DAO. Desta forma, assim como a fonte de dados pode mudar, a implementação interna do DAO também pode mudar, garantindo que nenhuma alteração é necessária no código fonte da camada de negócio que a utiliza.

2.6.2 Padrão *Identity Field*

Segundo Fowler (2006, p. 215), o padrão *Identity Field*, em português Campo Identidade, serve para guardar o campo “Id” de um banco de dados em um objeto para manter a identidade entre o objeto e uma linha do banco de dados. O campo “Id”, neste caso, representa a chave primária da tabela. A Figura 5 apresenta a aplicação desse padrão em uma classe.



Fonte: Fowler (2006, p. 215).
Figura 5 – Padrão *Identity Field*

Os bancos de dados relacionais diferenciam um registro de outro usando uma chave em particular, a chave primária. Entretanto, objetos na memória não possuem essa tal chave. Não há problemas para ler dados de um banco de dados, mas para gravá-los de volta, é necessário vincular o banco de dados ao sistema de objetos em memória. Em essência, o campo de identidade é muito simples. Basta armazenar a chave primária da tabela em um atributo dos objetos. Porém, a seguir são apresentadas algumas problemáticas que este padrão envolve e alternativas para solução. (FOWLER, 2006, p. 215).

2.6.2.1 Chaves com ou sem significado

Fowler (2006, p. 215) explica que a primeira preocupação é se devem ser usadas chaves com ou sem significado. Uma chave com significado é como o número do Cadastro de Pessoa Física (CPF) de um cliente. Uma chave sem significado é basicamente um número randômico que o banco de dados inventa e que não se destina a uso de seres humanos. O perigo de uma chave com significado é que, embora em teoria elas sejam boas chaves, na prática não o são. Para simplesmente funcionar, as chaves precisam ser únicas. Para funcionar bem, elas precisam ser imutáveis. Embora os números atribuídos sejam supostamente únicos e imutáveis, erros humanos muitas vezes fazem com que eles não sejam nem uma coisa nem outra. Se o usuário de um sistema digitar equivocadamente o número do CPF de um cliente no lugar de outro, o registro resultante não é nem único e nem imutável, presumindo que o usuário queira corrigir o erro. Com isso, chaves com significado são indicadas apenas para sistemas pequenos e casos muitos estáveis.

2.6.2.2 Chaves simples *versus* chaves compostas

Conforme Fowler (2006, p. 215), uma chave simples usa apenas um campo do banco de dados. Uma chave composta usa mais de um. A vantagem de uma chave composta é que ela freqüentemente é mais fácil de usar quando uma tabela faz sentido no contexto de outra. Um bom exemplo são os pedidos e as linhas de itens, em que uma boa chave para uma linha do item é uma chave composta pelo número do pedido e um número seqüencial que identifica a linha do pedido. Embora freqüentemente as chaves compostas façam sentido, as chaves simples são melhores em termos de uniformidade. Se forem utilizadas chaves simples em todo o sistema, pode-se usar o mesmo código para toda manipulação de chaves, enquanto as chaves compostas requerem tratamento especial para cada caso em classes concretas.

Para representar chaves compostas, a melhor opção é criar uma classe de chave. Uma classe de chave genérica pode armazenar uma seqüência de objetos que atuam como os segmentos da chave.

2.6.2.3 O tipo da chave

O tipo do campo que guarda a chave é outra questão importante. A operação mais comum que é feita com uma chave é o teste de igualdade, portanto o ideal é utilizar um tipo com operação rápida de igualdade. Outra operação importante é a obtenção da próxima chave. Assim, um tipo de inteiro longo é freqüentemente a melhor aposta. As *strings* também podem se adequar, mas a verificação da igualdade pode ser mais lenta e incrementar *strings* é um pouco mais difícil de implementar (FOWLER, 2006, p. 216).

2.6.2.4 Obtendo uma nova chave

Segundo Fowler (2006, p. 217), para criar um novo objeto persistente será necessária uma nova chave. Existem três opções principais: Deixar o banco de dados gerá-la automaticamente, usar um *Global Unique IDentifier* (GUID) ou gerar a própria chave na aplicação.

Deixar o banco de dados gerar a chave é o caminho mais fácil, desde que o banco

tenha esta funcionalidade. Se for utilizado um campo auto-gerado, cada vez que é inserido um registro no banco de dados, este gera uma chave primária única automaticamente. Porém, nem todos os bancos de dados fazem isto da mesma maneira. Muitos dos que fazem lidam com isso de um modo que causa problemas para o mapeamento objeto-relacional, por exemplo, se não existir uma maneira de resgatar o valor que foi gerado para a chave (FOWLER, 2006, p. 217).

Uma abordagem alternativa à geração automática são as chamadas *sequences*, ou seqüências, onde envia-se um comando `select` que referencia a seqüência e banco de dados então retorna o próximo valor gerado. A pesquisa da seqüência é automaticamente executada em uma transação separada, de forma que não bloqueará transações concorrentes. O seu único ponto negativo é que também não está disponível em todos os bancos de dados (FOWLER, 2006, p. 217).

Um GUID é um número gerado que idealmente é único em todas as máquinas no espaço e no tempo. Algumas plataformas fornecem funções para geração do GUID, como a API do Windows. O algoritmo é interessante e envolve endereços de placa *ethernet*, hora do dia, nano segundos e números de identificação dos *chips*. Sendo um número completamente único, considera-se uma chave segura. Uma desvantagem do GUID é que a chave resultante possui 128 bits representados em 32 caracteres, e isso pode ser um problema pois chaves longas são difíceis tanto de digitar quanto de ler. Por seu tamanho extenso, elas também podem levar a problemas de desempenho, especialmente com índices (FOWLER, 2006, p. 217).

A última opção é gerar a própria chave. Um mecanismo simples para sistemas pequenos é fazer uma varredura de tabela usando a função SQL `max` para encontrar a maior chave na tabela e então incrementá-la de uma unidade para usá-la. Infelizmente essa leitura bloqueia a tabela interna enquanto estiver sendo executada, o que significa que seu desempenho será diminuído quando houver inserções concorrentes. Também é necessário assegurar-se de que há um completo isolamento entre as transações, senão diferentes transações podem obter o mesmo valor de “Id”. Outra abordagem é usar uma tabela de chaves separada. Esta tabela tipicamente terá uma linha para cada tabela no banco de dados e possui duas colunas: nome e próximo valor disponível. Para usar esta tabela de chave, lê-se determinada linha, guarda-se o número, incrementa-se o número e grava-se o número de volta na linha (FOWLER, 2006, p. 218).

2.6.2.5 Trabalhando com herança

Quando se utiliza herança é necessário um cuidado especial. Se estiver sendo utilizada a estratégia de herança de tabela concreta ou herança de tabela de classe, deve-se mapear chaves que sejam únicas na hierarquia em vez de únicas em cada tabela. Isto deve ser feito para que não existam dois objetos com mesmo valor de identidade que são convertíveis para a classe pai, o que estaria violando a regra da unicidade. Para ter este comportamento, declara-se o campo que contém o "Id" na classe pai, que as classes filhas herdarão este campo. A estratégia de geração do novo valor também teria que ser mapeada na classe pai para garantir a unicidade em toda a hierarquia de classes (FOWLER, 2006, p. 216).

2.6.3 Padrão *Identity Map*

Conforme Fowler (2006, p. 196-199), o padrão *Identity Map*, ou Mapa de Identidade, assegura que cada objeto seja carregado apenas uma vez, mantendo cada objeto carregado em um mapa. Procura-se objetos usando o mapa quando se referindo a eles. Se fossem carregados os dados do mesmo registro do banco de dados em dois objetos diferentes, alguns valores poderiam acabar sendo perdidos quando se atualiza ambos os objetos. Relacionado a isso está um problema óbvio de desempenho. Se forem carregados os mesmos dados mais de uma vez, haverá um custo alto em chamadas remotas. Assim, deixar de carregar os mesmos dados duas vezes não apenas ajuda na consistência, mas pode também aumentar a velocidade da aplicação.

Um Mapa de Identidade mantém um registro de todos os objetos que foram lidos do banco de dados em uma única transação de negócio. Sempre que precisar de um objeto, a aplicação verifica o Mapa de Identidade primeiro para ver se já o tem.

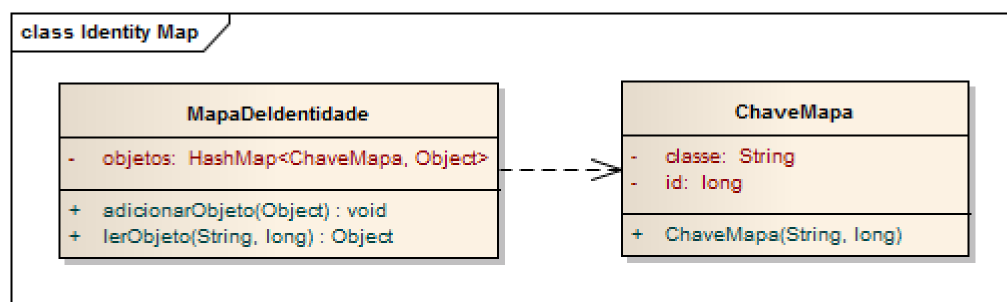
2.6.3.1 Funcionamento do Mapa de Identidade

Fowler (2006, p. 196-199) explica que a idéia básica por trás do Mapa de Identidade é ter um mapa ou uma série de mapas contendo objetos que foram trazidos do banco de dados. Em um caso simples, com um esquema isomórfico, tem-se um mapa por tabela do banco de

dados. Quando a aplicação for carregar um objeto do banco de dados, primeiro verifica o mapa. Se houver um objeto nele que corresponda ao que se estiver carregando, utiliza-se este objeto do mapa. Caso contrário, seleciona-se do banco de dados, colocando os objetos no mapa para referência futura quando houver necessidade.

Geralmente se usa um mapa de identidade para gerenciar qualquer objeto trazido de um banco de dados e modificá-lo. A razão principal é que não se deseja ter uma situação na qual dois objetos na memória correspondam a um único registro no banco de dados. Nesta situação, poderiam modificar-se os dois registros inconsistentemente e assim confundir o mapeamento do banco de dados. Outra importância do Mapa de Identidade é que ele atua como um *cache* para as leituras do banco de dados, o que significa que você pode evitar ir ao banco de dados cada vez que precisar de algum dado. O Mapa de Identidade ajuda a evitar conflitos de atualização dentro de uma mesma sessão, mas não faz nada para lidar com conflitos que atravessam várias sessões (FOWLER, 2006, p. 196-199).

A Figura 6 mostra um diagrama de classes exemplificando um mapa de identidade.



Fonte: adaptado de Fowler (2006, p. 196).

Figura 6 – Padrão *Identity Map*

Para cada mapa de identidade, existe um campo mapa, representado no diagrama por `objetos`, e métodos de acesso. A Classe `ChaveMapa` é uma classe interna da classe `MapaDeIdentidade`, para ser utilizada como chave para o `HashMap`. A implementação dos métodos `adicionarObjeto` e `lerObjeto` em Java ficaria conforme o Quadro 6.

```
public void adicionarObjeto(Object objeto) {

    String classe = objeto.getClass().getName();
    long id = lerIdObjeto(objeto);
    ChaveMapa chave = new ChaveMapa(classe, id);
    objetos.put(chave, objeto);
}

public Object lerObjeto(String classe, long id) {

    ChaveMapa chave = new ChaveMapa(classe, id);
    return objetos.get(chave);
}
```

Fonte: Fowler (2006, p. 199).

Quadro 6 – Implementação de métodos do *Identity Map* em Java

O mesmo código fonte em Delphi ficaria conforme o Quadro 7.

```
procedure TMapaIdentidade.AdicionarObjeto(Objeto: TObject);
var
  Chave: TChaveMapa;
begin
  Chave = TChaveMapa.Create(Objeto.ClassName, LerIdObjeto(Objeto));
  Objetos.Put(Chave, Objeto);
end;

function TMapaIdentidade.LerObjeto(Classe: string; Id: LongInt): TObject;
var
  Chave: TChaveMapa;
begin
  Chave = TChaveMapa.Create(Classe, Id);
  Result := Objetos.Get(Chave);
end;
```

Fonte: adaptado de Fowler (2006, p. 199).

Quadro 7 – Implementação de métodos do *Identity Map* em Delphi

2.6.4 Padrão *Strategy*

Na concepção de Gamma et al (2000, p. 315-323), o padrão *Strategy* define uma família de algoritmos, encapsula cada um deles e torna-os substituíveis. Além disso, permite que o algoritmo varie independente dos clientes que o utilizam.

Considere-se um exemplo de ordenação. Existem vários algoritmos de ordenação, como Quicksort, Bubblesort e Mergesort. Amarrar todos estes algoritmos com as classes que os utilizam não é desejável por algumas razões:

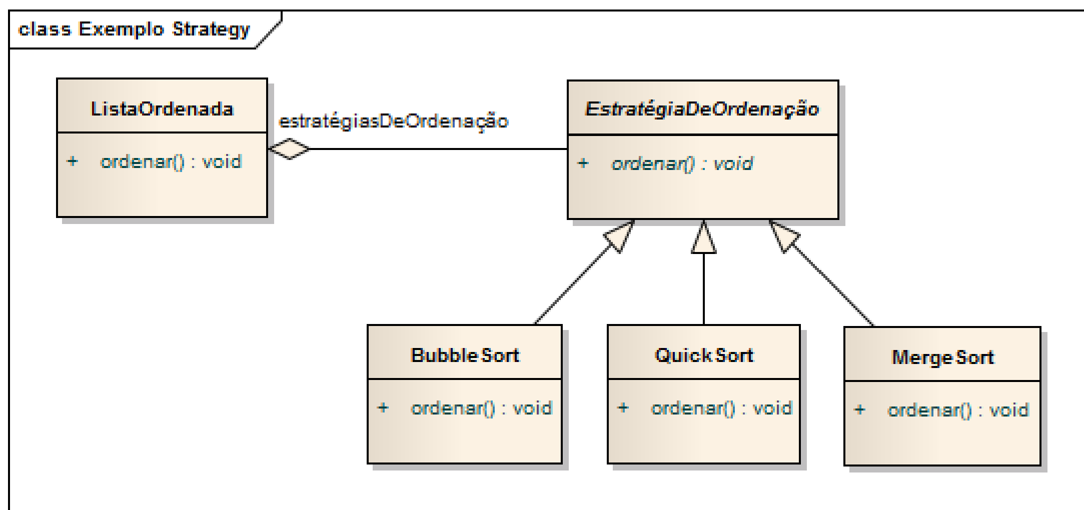
a) lógicas de negócio que necessitam de ordenação se tornariam mais complexas se incluíssem o código responsável pela ordenação. Isso tornaria o código de lógica

de negócio maior e mais difícil de manter, especialmente se for necessário suportar diversos algoritmos de ordenação;

b) diferentes algoritmos serão apropriados em diferentes momentos. Não é desejável suportar diversos algoritmos de ordenação se eles não forem utilizados;

c) é difícil adicionar novos algoritmos e variar entre os já existentes quando o código de ordenação já está embutido na lógica de negócio.

Estes problemas podem ser evitados definindo classes que encapsulam diferentes algoritmos de ordenação. Um algoritmo que é encapsulado desta forma é chamado de estratégia. A Figura 7 mostra um diagrama de classes exemplificando este padrão.



Fonte: adaptado de Gamma et al (2000, p. 316).

Figura 7 – Padrão *Strategy*

Gamma et al (2000, p. 315-323) afirma que o padrão *Strategy* pode ser utilizado quando:

a) muitas classes afins diferem apenas em seu comportamento. O padrão provê uma forma de configurar uma classe com um ou vários comportamentos;

b) são necessárias diferentes variações de um algoritmo. Por exemplo, podem ser definidos algoritmos que refletem diferentes escolhas que podem mudar com o tempo ou espaço. Estratégias podem ser utilizadas quando estas variações são implementadas como uma hierarquia de classes de algoritmos;

c) um algoritmo utiliza dados que não deveriam ser conhecidos por quem o utiliza. Usa-se o padrão *Strategy* para evitar a exposição de estruturas complexas específicas do algoritmo;

d) uma classe define muitos comportamentos, métodos complexos com múltiplos comandos condicionais.

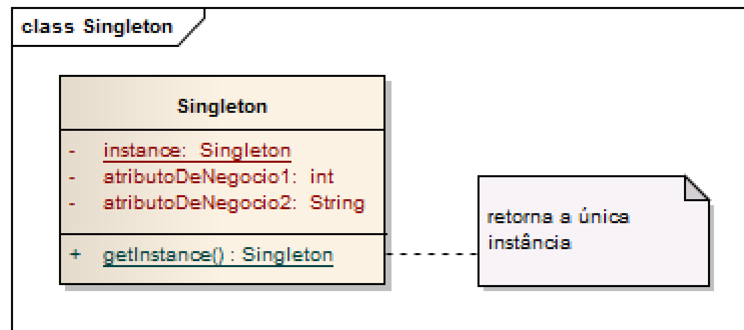
2.6.5 Padrão *Singleton*

Conforme Gamma et al (2000, p. 127-134), para certas classes é importante haver somente uma instância. Exemplos para tais casos seriam uma classe de *spooler* de impressão, uma classe de sistema de arquivos ou uma classe de gerenciador de janelas. O padrão *Singleton* serve para garantir que uma classe possua apenas uma instância, e provê um ponto global de acesso a ela.

Uma variável global faz um único objeto ser facilmente acessível, mas não garante que objetos da mesma classe possam ser instanciados em outros locais do sistema. Uma solução mais inteligente é tornar a própria classe responsável por manter o ciclo de vida da sua própria instância. Uma classe pode interceptar requisições de criação de novos objetos para garantir que outras instâncias não sejam criadas, e também pode disponibilizar um método principal para acessar esta instância. Aplica-se o padrão *Singleton* em uma classe fazendo a classe atender às seguintes regras:

a) a classe deve possuir um atributo privado e estático cujo tipo é a própria classe e cujo conteúdo inicial é `null`. Este atributo irá armazenar a única instância da classe. Um nome sugerido para este atributo é `instance`;
b) a classe deve possuir um método estático e público por onde tal instância é acessada, cujo tipo do retorno é a própria classe. Tal método deverá retornar o conteúdo do atributo `instance`. Porém, antes de retornar, faz a seguinte operação: caso o atributo esteja nulo, instancia-o;
c) a classe deve garantir que não existe nenhum construtor público. Para fazer isto, em alguns casos, é necessário tornar todos os construtores privados;
d) em linguagens que não possuem *garbage colector*, a classe deve garantir a destruição da instância única no momento apropriado, como por exemplo, na finalização do programa.

A Figura 8 mostra um diagrama de classes exemplificando este padrão.



Fonte: Gamma et al (2000, p. 130).

Figura 8 – Padrão *Singleton*

Gamma et al (2000, p. 127-134) explica que a aplicação do padrão *Singleton* traz os seguintes benefícios:

a) acesso controlado à única instância: como a classe passa a encapsular tal instância, ela pode ter um controle rígido sobre como e quando os clientes a acessam;
b) escopo reduzido: o padrão *Singleton* é uma evolução sobre as antigas variáveis globais. Ele evita que o escopo seja poluído com variáveis globais que guardam apenas um objeto;
c) pode ser refinado e suportar herança facilmente. Caso seja necessário controlar uma instância para cada classe filha, isto pode ser feito com uma lista privada estática na classe pai, para guardar o registro das classes filhas, no lugar do atributo padrão `instance`;
d) caso seja necessário para a aplicação, pode permitir um número variável de instâncias. E isto pode ser feito alterando somente o código fonte da classe `Singleton`, modificando a implementação do método de acesso à instância e o atributo onde é guardado. Não é necessário alterar o código cliente. Uma utilidade para isto seria a criação de um *pool* de objetos;
e) maior flexibilidade sobre métodos estáticos: Uma alternativa para o padrão *Singleton* seria a definição de classes com todos os atributos e métodos estáticos, porém esta abordagem torna mais trabalhosa a refatoração caso a classe deva suportar mais de uma instância. Além disso, em algumas linguagens os métodos estáticos não podem ser virtuais, então não podem ser sobrescritos métodos em classes filhas. Outra vantagem é que uma classe `Singleton` pode ser implementada de tal forma a ser instanciada somente no momento da sua primeira utilização, e isto pode ser importante para, por exemplo, evitar alocação de memória desnecessariamente.

2.6.6 Padrão *Adapter*

Gamma et al (2000, p. 139-150) explica que o padrão *Adapter*, também chamado de *Wrapper*, converte a interface de uma classe em uma outra interface esperada pelo cliente. Este padrão permite que certas classes que não podem trabalhar juntas por motivos de incompatibilidade entre interfaces, possam trabalhar em conjunto.

Algumas vezes uma classe de elementos gráficos que foi implementada para reuso não pode ser reusada porque sua interface não combina com a interface específica do domínio da aplicação.

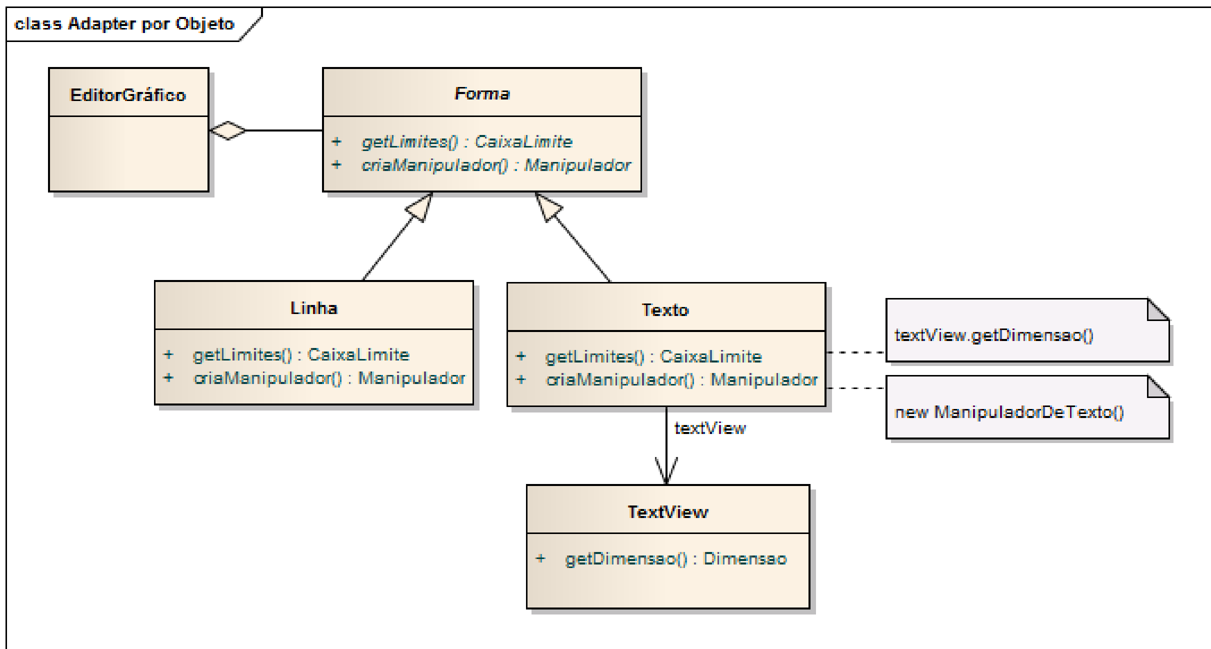
Considere-se como exemplo um editor de desenho que permite ao usuário desenhar e arranjar elementos gráficos como linhas, polígonos e texto em figuras e diagramas. O ponto chave da abstração do editor gráfico é o objeto gráfico, que possui uma forma editável e pode desenhar a si mesmo. A interface para objetos gráficos é definida por uma classe abstrata chamada `Forma`. O editor define uma classe filha de `Forma` para cada tipo de objeto gráfico: classes `Linha`, `Polígono`, `Texto` e assim por diante (GAMMA et al, 2000, p. 139-150).

As classes de formas geométricas elementares como `Linha` e `Polígono` são mais fáceis de implementar, mas a classe filha `Texto` pode ser mais difícil de implementar por ter de tratar de exibição e edição de textos, o que envolve atualizações de tela mais complexas e gerenciamento de *buffer*. Entretanto, uma biblioteca pode disponibilizar uma classe sofisticada `TextView` para exibição e edição de texto. A melhor idéia seria reutilizar a classe `TextView` para implementação da classe `Texto`, porém a implementação dessa biblioteca não contava com a classe `Forma`. Então não é possível utilizar objetos de `TextView` no lugar de um objeto `Forma` (GAMMA et al, 2000, p. 139-150).

Para fazer com que a classe `TextView` possa ser utilizada como `Forma`, poderia ser alterada a classe `TextView` para ficar conforme a interface da classe `Forma`, mas isto não é possível a menos que se tenha o código fonte da biblioteca em questão. E mesmo que se tenha não faria sentido mudar a classe `TextView`. Uma biblioteca não precisa adotar interfaces específicas do domínio da aplicação para funcionar. Ao invés disto, a classe `Texto` poderia ser definida como uma adaptação da classe `TextView` para a interface de `Forma`. Isto pode ser feito de duas formas:

a) herdando a interface de `Forma` e implementação de `TextView`;
b) acomodando uma instância de `TextView` dentro de um objeto de `Texto` e implementando a classe `Texto` sobre a interface de `TextView`.

Gamma et al (2000, p. 139-150) afirma que estas duas abordagens correspondem respectivamente a duas versões do padrão Adapter: adaptador por classe e adaptador por objeto. O diagrama da Figura 9 ilustra o caso do adaptador por objeto.



Fonte: Gamma et al. (2000, p. 140).

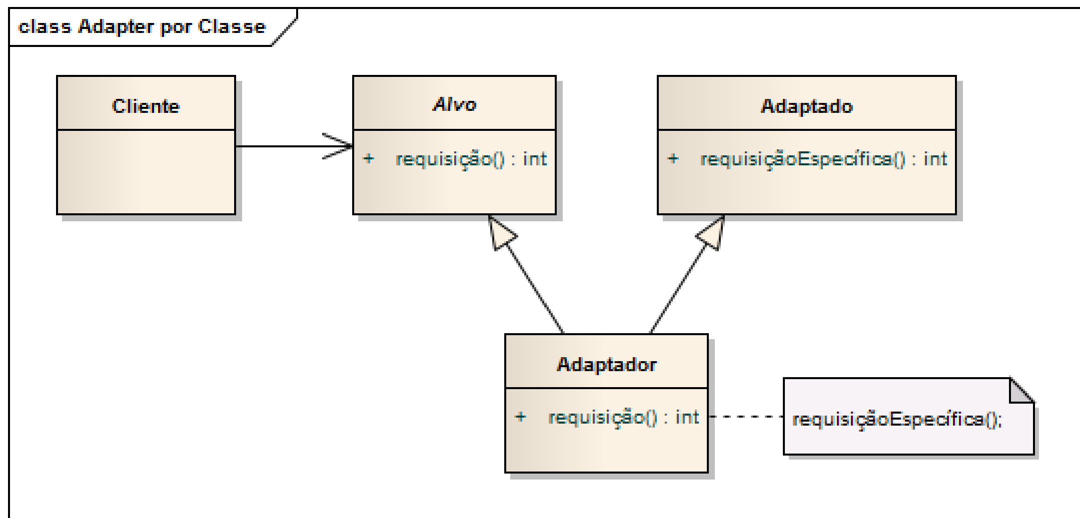
Figura 9 – Padrão *Adapter* por objeto

Este diagrama mostra como requisições de limites, declaradas na classe `Forma`, são convertidas para chamadas do método `getDimensao()` definido na `TextView`. Desta forma, o editor gráfico pode reutilizar a classe `TextView` normalmente. Freqüentemente o adaptador é responsável por funcionalidades que a classe adaptada não fornece. O diagrama mostra como o adaptador pode cumprir esta responsabilidade. O usuário pode arrastar qualquer objeto de `Forma` para uma nova posição da tela, mas a classe `TextView` não foi desenhada para suportar isto. A classe `Texto` pode adicionar esta funcionalidade faltante implementando o método `criaManipulador()`, que retorna uma instância de uma classe apropriada de `Manipulador` (GAMMA et al., 2000, p. 139-150).

`Manipulador` é uma classe abstrata para objetos que sabem como animar uma `Forma` em resposta à interação com o usuário, como arrastar uma forma para uma nova posição. Pode haver diferentes classes de `Manipulador` para diferentes formas. `ManipuladorDeTexto`, por exemplo, seria a classe correspondente para a classe `Texto`. Ao retornar uma instância de `ManipuladorDeTexto`, a classe `Texto` implementa a funcionalidade que a classe `TextView` não possui mas a classe `Forma` requer.

Um adaptador por classe pode ser implementado como uma generalização de duas classes em uma linguagem que suporta herança múltipla, ou uma generalização de uma classe

que implementa uma determinada interface em linguagens que não suportam herança múltipla. A Figura 10 ilustra um exemplo (GAMMA et al., 2000, p. 139-150).

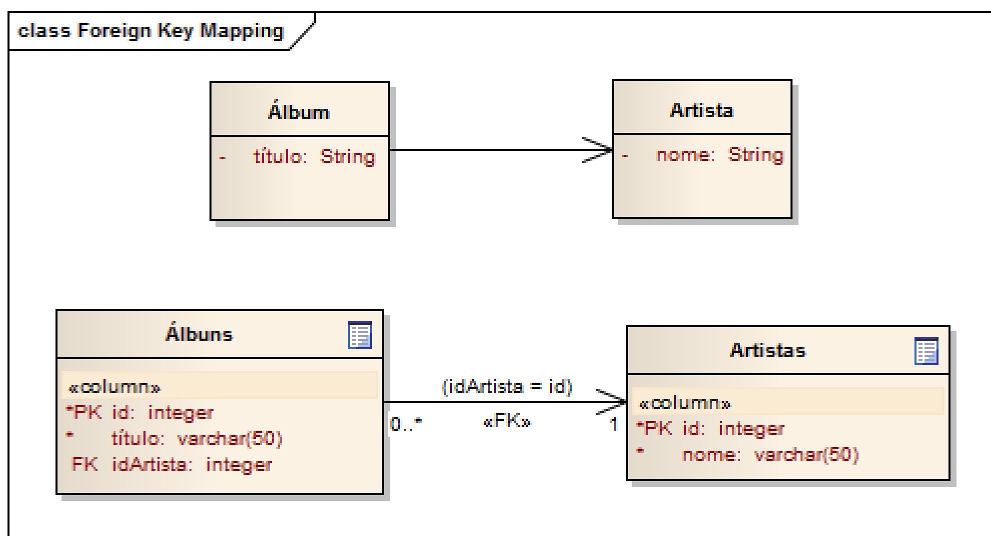


Fonte: Gamma et al (2000, p. 143).

Figura 10 – Padrão *Adapter* por classe

2.6.7 Padrão *Foreign Key Mapping*

Segundo Fowler (2006, p. 233-243), o padrão de projeto *Foreign Key Mapping*, ou Mapeamento de Chave Estrangeira, serve para mapear uma associação entre objetos para uma referência de chave estrangeira entre tabelas no banco de dados. A Figura 11 exemplifica este padrão, onde se mostra as classes acima e as tabelas abaixo.

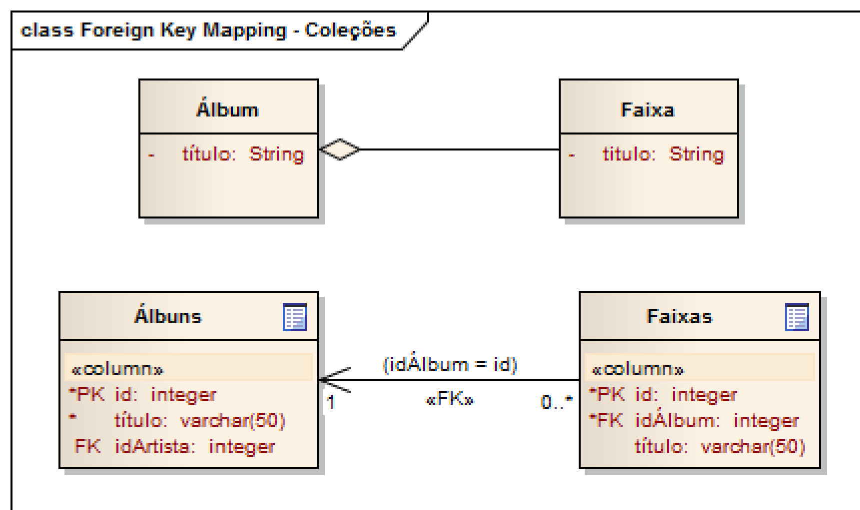


Fonte: Fowler (2006, p. 233).

Figura 11 – Padrão *Foreign Key Mapping*

Os objetos podem se referir uns aos outros diretamente por meio de referências.

Mesmo o sistema orientado a objetos mais simples conterá um pequeno grupo de objetos conectados entre si. Para gravar esses objetos em um banco de dados, é vital gravar essas referências. Contudo, já que os dados das referências em si são explícitos à instância particular do programa sendo executado, não se pode simplesmente gravar os valores de dados em estado bruto. Outra complicação é o fato de que os objetos podem facilmente possuir coleções de referências para outros objetos, conforme exemplificado na Figura 12. Tal estrutura viola a primeira forma normal dos bancos de dados relacionais (FOWLER, 2006, p. 233-243).



Fonte: Fowler (2006, p. 234).

Figura 12 – Padrão *Foreign Key Mapping* com coleções

Um Mapeamento de Chave Estrangeira mapeia uma referência a um objeto como uma chave estrangeira no banco de dados. A chave óbvia para resolver esse problema é o campo "Id". Cada objeto contém a chave do banco de dados da tabela apropriada. Se dois objetos são conectados com uma associação, esta associação pode ser substituída por uma chave estrangeira no banco de dados. Explicando de forma mais simples, quando for salvar um álbum no banco de dados, grava-se o "Id" do artista ao qual o álbum está associado no registro do álbum (FOWLER, 2006, p. 233-243).

2.6.7.1 Mapeamento de coleções

Fowler (2006, p. 233-243) explica que um caso mais complicado surge quando se tem uma coleção de objetos. Não é possível gravar uma coleção no banco de dados, então é necessário inverter a direção da referência. Assim, quando existe uma coleção de faixas no álbum, coloca-se a chave estrangeira do álbum no registro da faixa. Além disso, é necessário

tratar atualizações. Atualizações significam que faixas podem ser acrescentadas ou removidas da coleção do álbum. Para tratar quais alterações aplicar no banco de dados, basicamente podem ser utilizadas três estratégias: excluir e inserir, adicionar um ponteiro reverso ou diferenciar a coleção.

Com a exclusão e a inserção, devem-se excluir todas as faixas do banco de dados vinculadas ao álbum e então inserir todas as que estão atualmente no álbum. À primeira vista essa estratégia parece um tanto espantosa, especialmente se não tiver sido alterada nenhuma faixa. Contudo, a lógica é fácil de implementar e, como tal, funciona muito bem comparada com as alternativas. A desvantagem é que isso só pode ser feito se as faixas forem mapeamentos dependentes, o que significa que elas devem pertencer ao álbum e não podem ser referenciadas de fora dele (FOWLER, 2006, p. 233-243).

Adicionar um ponteiro reverso coloca um vínculo da faixa de volta para o álbum, efetivamente tornando a associação bidirecional. Isso altera o modelo de objetos, mas, por outro lado, pode-se tratar a atualização usando a técnica simples para campos univalorados.

Se nenhuma dessas opções for atrativa, pode-se fazer uma diferenciação. Existem duas possibilidades: diferenciar em relação ao estado corrente do banco de dados ou diferenciar em relação ao que foi lido na primeira vez. Diferenciar em relação ao banco de dados envolve reler a coleção do banco de dados e então compará-la com a coleção no álbum. Qualquer coisa no banco de dados que não estiver no álbum obviamente foi removida. Qualquer coisa no álbum que não estiver no disco é obviamente um novo item a ser acrescentado (FOWLER, 2006, p. 233-243).

Diferenciar em relação ao que foi lido na primeira vez significa que se deve guardar os objetos lidos. Isso é melhor já que evita outra leitura do banco de dados. No caso geral, qualquer coisa que tenha sido adicionada à coleção precisa ser primeiro verificada para ver se é um objeto novo. Isso pode ser feito verificando se ele tem uma chave. Se não tiver, ele precisa ser adicionado ao banco de dados.

Para a remoção, deve-se descobrir se a faixa foi movida para outro álbum, se ela não tem um álbum ou se foi completamente excluída. Se ela tiver sido movida para outro álbum, ela deve ser atualizada quando for atualizado esse outro álbum. Se ela não tiver um álbum, é necessário colocar um `null` na chave estrangeira. Se a faixa foi excluída, então ela deve ser apagada. Tratar exclusões é muito mais fácil se o vínculo reverso for obrigatório, como é aqui, onde toda faixa deve estar em um álbum. Dessa maneira, não é necessário preocupar-se em detectar itens removidos da coleção, já que eles serão atualizados quando for processado o álbum ao qual eles foram adicionados (FOWLER, 2006, p. 233-243).

Se esse vínculo for imutável, significando que não pode ser alterado o álbum de uma faixa, então a adição sempre significa inserção, e a remoção sempre significa exclusão. Isso torna as coisas ainda mais simples.

2.6.7.2 Tratamento de ciclos

Uma das coisas com que se deve tomar cuidado é a existência de ciclos nas associações. Digamos que seja necessário carregar um pedido que tem uma associação com um cliente. O cliente tem um conjunto de pagamentos, e cada pagamento tem pedidos os quais ele está pagando, o que poderia incluir o pedido original que está sendo carregado. Para evitar se perder em ciclos existem duas escolhas que, no final das contas, resumem-se à forma como são criados os objetos. Normalmente, é uma boa idéia para um método de criação incluir dados que lhe darão um objeto completamente formado. Se isso for feito, é necessário colocar a Carga Tardia, mencionada no item 2.6.8, em pontos apropriados para quebrar os ciclos (FOWLER, 2006, p. 233-243).

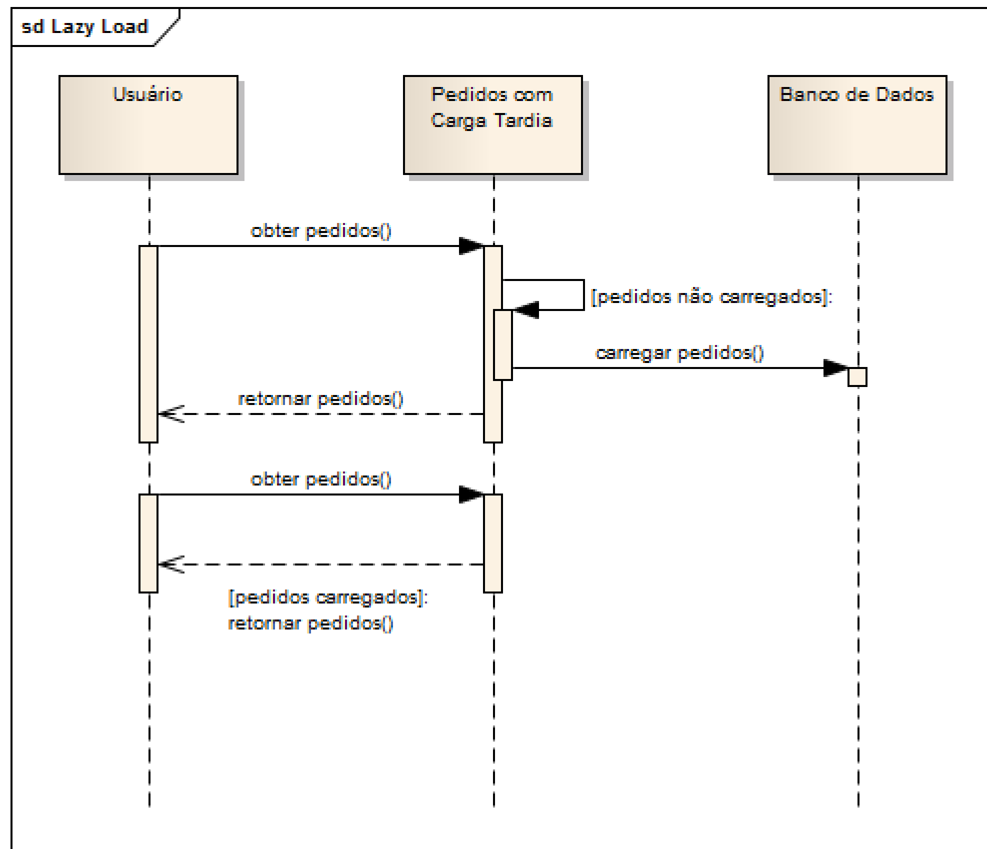
A outra escolha é criar objetos vazios e imediatamente colocá-los em um Mapa de Identidade. Dessa maneira, quando o ciclo voltar ao início, o objeto já estará carregado e o ciclo terminará. Os objetos que são criados não estão completamente formados, mas deveriam estar ao final do procedimento de carga. Isso evita ter que tomar decisões em casos especiais sobre o uso de Carga Tardia apenas para fazer uma carga correta.

2.6.7.3 Onde não utilizar

Fowler (2006, p. 233-243) afirma que um Mapeamento de Chave Estrangeira pode ser usado para quase todas as associações entre classes. Porém não é possível usá-lo em associações muitos-para-muitos. As chaves estrangeiras são valores únicos, e a primeira forma normal diz que não se pode armazenar diversas chaves estrangeiras em um único campo.

2.6.8 Padrão *Lazy Load*

Fowler (2006, p. 200-213) explica que o padrão *Lazy Load*, ou Carga Tardia, define um objeto que não contém todos os dados dos quais precisa, mas sabe como obtê-los. O diagrama de seqüência da Figura 13 mostra uma visão geral do funcionamento desse padrão.



Fonte: adaptado de Fowler (2006, p. 200).

Figura 13 – Padrão *Lazy Load*

Para carregar dados de um banco de dados para a memória, é conveniente projetar um sistema de modo que quando é carregado um objeto de interesse também sejam carregados os objetos relacionados a ele. Isso torna a carga mais fácil para o desenvolvedor que estiver usando o objeto, que de outra forma teria que carregar explicitamente todos os objetos de que precisa. Todavia, se esta premissa for levada ao seu final, chegará ao ponto em que carregar um objeto pode ter o mesmo efeito de carregar toda a base de dados de uma só vez, algo que prejudicaria o desempenho do sistema quando apenas alguns dos objetos são realmente necessários (FOWLER, 2006, p. 200-213).

Uma carga tardia interrompe este processo de carga por um tempo, deixando um marcador na estrutura do objeto de modo que se os dados forem necessários podem ser carregados apenas quando forem usados. Caso tais dados por algum motivo não sejam

utilizados, eles nem mesmo serão carregados.

Fowler (2006, p. 200-213) diz que há cinco estratégias principais pelas quais pode ser implementada a carga tardia: inicialização tardia, *proxy* virtual, armazenador de valor, fantasma ou utilização de programação orientada a aspectos. A seguir são detalhadas estas estratégias e comentado sobre como escolher a estratégia mais apropriada para cada caso.

2.6.8.1 Inicialização tardia

Inicialização tardia é a abordagem mais simples. A idéia básica é que cada acesso ao campo verifique primeiro para ver se ele é nulo. Se for, calcula-se o valor do campo antes de retorná-lo. Para que isto funcione bem, deve-se assegurar que o campo seja encapsulado, o que quer dizer que todo acesso ao campo, mesmo de dentro da classe, é feito por meio de um método de leitura. Usar inicialização tardia é simples, mas tende a forçar uma dependência entre o objeto e o banco de dados (FOWLER, 2006, p. 200-213).

2.6.8.2 *Proxy* virtual

Um *proxy* virtual é um objeto que parece com o objeto que deveria estar no campo, mas na verdade não contém nada. Apenas quando um dos seus métodos é chamado, ele carrega o objeto correto a partir do banco de dados. A vantagem do *proxy* virtual é que ele parece exatamente com o objeto que deve estar lá. A desvantagem é que ele não é este objeto, o que pode resultar num problema de identidade. Além disso, pode-se ter mais de um *proxy* virtual para o mesmo objeto real. Todos esses *proxies* terão diferentes identidades de objeto, mas ainda assim representam o mesmo objeto conceitual (FOWLER, 2006, p. 200-213).

Em alguns ambientes, outro problema é a necessidade de se criar muitos *proxies* virtuais, um para cada classe que estiver usando *proxy*. Isso pode ser evitado em linguagens tipadas dinamicamente, mas em linguagens tipadas estaticamente este trabalho é mais difícil. Mesmo quando a plataforma fornece recursos convenientes, como os *proxies* de Java, outros obstáculos podem aparecer.

2.6.8.3 Armazenador de valor

Com classes do domínio, pode-se contornar esses problemas usando um armazenador de valor. Conceitualmente, trata-se de um objeto que encapsula algum outro objeto. Para obter o objeto subjacente, solicita-se seu valor ao armazenador de valor, mas apenas no primeiro acesso ele obtém os dados do banco de dados. As desvantagens do armazenador de valor são que a classe precisa saber que ele existe e que se perde a característica explícita da forte verificação de tipos. Pode-se evitar problemas de identidade assegurando que o armazenador de valor nunca seja passado para além da classe a qual ele pertence (FOWLER, 2006, p. 200-213).

2.6.8.4 Fantasma

Um fantasma é o objeto real em um estado parcial. Quando é carregado o objeto do banco de dados, ele contém apenas seu “Id”. Sempre que for acessado um campo, ele carrega seu estado completo. É claro que não há necessidade de carregar todos os dados de uma vez só. Pode-se reuni-los em grupos que sejam comumente usados juntos. Ao usar um fantasma, pode-se inseri-lo imediatamente no mapa de identidade. Dessa forma é mantida a identidade e são evitados todos os problemas causados por referências cíclicas durante a leitura de dados (FOWLER, 2006, p. 200-213).

2.6.8.5 Programação orientada a aspectos

A Carga Tardia é uma boa candidata à programação orientada a aspectos[6]. Pode-se colocar o comportamento da Carga Tardia em um aspecto separado, o que permite alterar a estratégia de carga tardia separadamente assim como liberar os desenvolvedores do domínio de ter que lidar com questões de carga tardia (FOWLER, 2006, p. 200-213).

---

[6] Segundo Soares e Borba (2002), programação orientada a aspectos é um paradigma de programação que permite que a implementação de um sistema seja separada em requisitos funcionais e não funcionais, separando código de funções específicas como, por exemplo, controle de concorrência, tratamento de exceções e *log*.

2.6.8.6 Escolhendo a melhor estratégia

Muitas vezes existem situações nas quais diferentes casos de uso funcionam melhor com diferentes variedades de Carga Tardia, ou então, casos onde é melhor não utilizá-la. A maneira de lidar com isso é ter diferentes objetos de interação com o banco para os diferentes casos de uso. Assim, pode-se ter dois objetos mapeadores de pedidos: um que carrega os itens imediatamente e outro que os carrega tardiamente. O código da aplicação escolhe o mapeador apropriado dependendo do caso de uso. Uma variação disso é ter o mesmo objeto carregador básico, mas delegar para um objeto estratégia a decisão sobre o padrão de carga. Isto é um pouco mais sofisticado, mas pode ser uma maneira melhor de fatorar comportamento (FOWLER, 2006, p. 200-213).

A decisão de quando usar a Carga Tardia é relacionada à quantidade de informação que se deseja trazer do banco de dados quando se carrega um objeto e quantas chamadas ao banco de dados isso irá requerer. Normalmente não faz sentido usar a Carga Tardia em um campo que é armazenado na mesma linha do resto do objeto, porque na maioria das vezes não custa mais trazer dados adicionais em uma chamada, mesmo se o campo for grande como um *Binary Large OBject* (BLOB). Só vale a pena considerar Carga Tardia se o campo requer uma chamada de banco de dados extra para ser acessado (FOWLER, 2006, p. 200-213).

Muitas vezes, é uma boa idéia trazer todos os dados que serão necessários em uma única chamada, de forma a ter logo todo o grafo de objetos carregado, especialmente se isso corresponder a uma única interação com uma interface gráfica com o usuário. O melhor momento para usar a Carga Tardia é quando ela envolver uma chamada extra, e os dados que estiverem sendo requisitados não forem usados quando o objeto principal for usado.

2.6.9 Padrão *Model View Controller* (MVC)

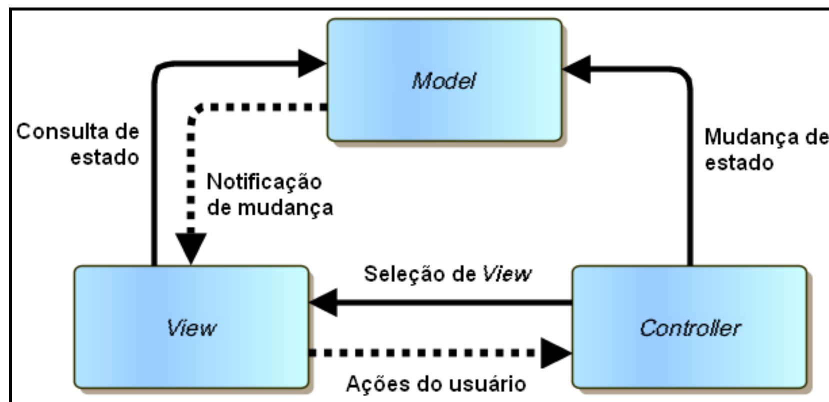
Conforme Quicoli (2007, p. 26-34), o padrão de projeto MVC separa a aplicação em três camadas distintas, *model*, *view* e *controller*, que podem ser definidas da seguinte forma:

a) *model* é a representação da informação que a aplicação utiliza. Em um sistema orientado a objetos o *model* pode ser composto pelas classes de entidades do sistema, como por exemplo classes `Pessoa`, `Cliente` e `NotaFiscal`;
b) *view* é a apresentação gráfica do *model*. Por exemplo, um formulário expondo as

informações de um cliente. Uma *view* está sempre ligada a um *model* e sabe como exibi-lo ao usuário. Para um determinado *model* podem existir várias *views*. Além disso, uma *view* também pode ser composta por várias *views*;

c) *controller* é responsável por mapear as ações do usuário em ações do sistema, por exemplo se um botão ou item do menu é clicado é ele que deve definir como a aplicação responderá.

A vantagem da separação da aplicação nestas três camadas é torná-las independentes e desacopláveis, além de possibilitar a reutilização do mesmo *model* em várias *views*. A Figura 14 mostra como é feita a comunicação entre as três camadas do MVC, na qual as linhas contínuas representam métodos e as linhas pontilhadas representam eventos.



Fonte: Quicoli (2007, p. 26-34).

Figura 14 – A comunicação dentro do MVC

2.7 TRABALHOS CORRELATOS

Verificou-se que já foram elaborados projetos envolvendo ORM em diversas linguagens de programação. Dentre os pesquisados, alguns *frameworks* serão utilizados como base para este trabalho, os quais são listados a seguir. Também serão obtidas ideias da ferramenta para aplicação do padrão DAO em sistemas desenvolvidos em Delphi (SARDAGNA, 2007).

2.7.1 Hibernate

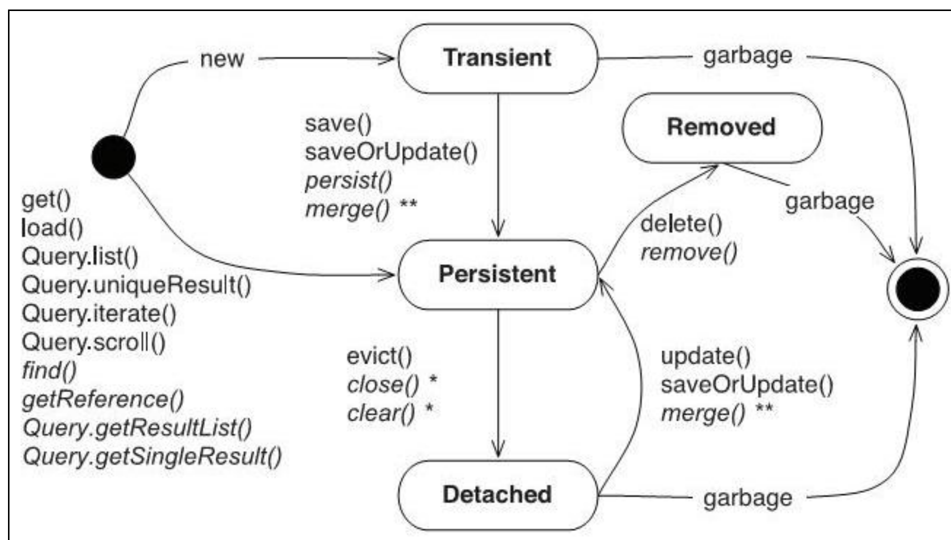
Segundo Silva e Silva (2009, p. 113-114), Hibernate é um *framework* de mapeamento

objeto-relacional desenvolvido em Java, atualmente considerado o mais conhecido do mercado. Sua principal função é abstrair a camada de persistência, economizando esforço e preocupações referentes a tal tarefa. Possui uma arquitetura fácil de configurar, simplificando bastante a tarefa do desenvolvedor. A partir da versão 3 o Hibernate implementa a especificação Java Persistence API (JPA) através do conceito de anotações, existente na linguagem Java, possibilitando definir o mapeamento diretamente na classe.

Conforme Bauer e King (2007, p. 192), o Hibernate suporta os seguintes modos de mapeamento de herança:

a) herança de tabela por classe concreta;
b) herança de tabela única;
c) herança de tabela por subclasse.

Bauer e King (2007, p. 386) apresentam através da Figura 15 os possíveis estados dos objetos de entidade no Hibernate, indicando o seu ciclo de vida. Na especificação JPA, tais objetos são mantidos pela classe `EntityManager`, cujos nomes dos métodos são mostrados em itálico no diagrama. Os métodos que não aparecem em itálico são os correspondentes das versões anteriores do Hibernate, que não implementavam a especificação JPA.



Fonte: Bauer e King (2007, p. 386).

Figura 15 – Estados dos objetos e suas transações no Hibernate

2.7.2 *Instant Objects*

Instant Objects (Instant Objects, 2006) é um *framework* para desenvolvimento de sistemas em Delphi que conta com um conjunto de componentes de código fonte aberto. Foi

desenvolvido para construir aplicações orientadas a objetos baseadas em objetos de negócio que podem ser persistidos em uma base de dados. O *framework* conta com componentes de persistência de objetos e também apresentação dos dados através de *data-aware controls*, que são componentes visuais para exibição dos dados. Ele suporta vários SGBDs diferentes, tanto orientados a arquivos quanto orientados a SQL, como Firebird, Microsoft SQL Server, IBM DB2, Oracle, InterBase, Informix, Advantage Database Server, ElevateSoft DBISAM, FlashFiler, Nexus DB, MySQL, PostgreSQL e outros.

2.7.3 Ferramenta para aplicação do DAO em sistemas desenvolvidos em Delphi

Sardagna (2007, p. 13) descreve seu *software* como uma ferramenta para aplicação do padrão DAO em um sistema já existente. Seu funcionamento baseia-se em retirar o código SQL do código fonte de negócio da aplicação e colocá-lo de forma organizada em uma camada de persistência que é gerada pela ferramenta, seguindo o padrão DAO. Tal ferramenta atua diretamente no código fonte da aplicação, alterando-o para uma estrutura que tenha a camada de persistência encapsulada no padrão de projeto DAO.

3 DESENVOLVIMENTO DO *FRAMEWORK*

Neste capítulo é explicado como o *framework* foi desenvolvido. Em primeiro lugar, são detalhados os principais requisitos funcionais e não funcionais que o *framework* se propõe a atender. Em seguida é apresentada a especificação, que dá uma explicação mais formal sobre o desenvolvimento do *framework*. Logo após é apresentado o modelo de anotação que foi elaborado para o mapeamento objeto-relacional. É então explicada a implementação do *framework*, onde são citados quais padrões de projetos, técnicas e ferramentas foram utilizadas. Por fim, é descrito como as funcionalidades foram testadas, quais os resultados e como o *framework* foi documentado.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O *framework* proposto deverá:

a) disponibilizar um modelo de anotação baseada em Atributos Customizados para o mapeamento objeto-relacional que possa ser utilizada na implementação de classes de entidades (Requisito Funcional – RF);
b) gerar comandos `select`, `insert`, `update` e `delete` na linguagem SQL a partir das classes e anotações definidas pelo usuário (RF);
c) gerar as tabelas do banco de dados a partir das classes e anotações definidas pelo usuário (RF);
d) disponibilizar uma camada de persistência que permita criar, obter, salvar e remover objetos de entidades de forma transparente (RF);
e) disponibilizar um gerenciador de objetos de entidades que controle o ciclo de vida dos objetos e permita fazer *cache* de objetos em memória local (RF);
f) ser desenvolvido utilizando os padrões de projeto DAO, Singleton, Adapter, Strategy, Identity Field, Identity Map, Foreign Key Mapping, Lazy Load e permitir que as aplicações que o utilizem possam ser desenvolvidas utilizando o padrão MVC (Requisito Não-Funcional - RNF);
g) ser implementado utilizando o ambiente de desenvolvimento Delphi 2010 para Windows (RNF);

h) suportar a arquitetura cliente-servidor, na qual várias instâncias da aplicação acessam o mesmo banco de dados (RNF);
i) ser compatível com o sistema operacional Microsoft Windows nas versões XP, Vista e 7 (RNF);
j) suportar os SGBDs MySQL e Firebird (RNF).

3.2 ESPECIFICAÇÃO

Nesta seção é apresentada a especificação do *framework* utilizando conceitos de orientação a objetos e a *Unified Modeling Language* (UML). Foi utilizada a ferramenta Enterprise Architect para a modelagem do projeto e a elaboração de diagramas de casos de uso, de classes, de componentes e de seqüência. Em primeiro lugar é mostrado o comportamento do *framework* através de um diagrama de casos de uso. Depois é apresentada a estrutura do *framework*, expondo uma visão geral dos componentes nos quais foi dividido com diagrama de componentes, e para cada componente uma visão das principais classes que o compõe através de diagrama de classes. Por fim é apresentada a interação entre algumas classes do *framework* com o diagrama de seqüência.

3.2.1 Comportamento do *framework*

O *framework* possui sete casos de uso que são demonstrados na Figura 16. Cada um dos casos de uso será explicado separadamente, apresentando os cenários, pré-condições para que possa ser executado e pós-condições que indicam o estado resultante da execução do caso de uso. Em todos os casos de uso, o único ator é a aplicação.

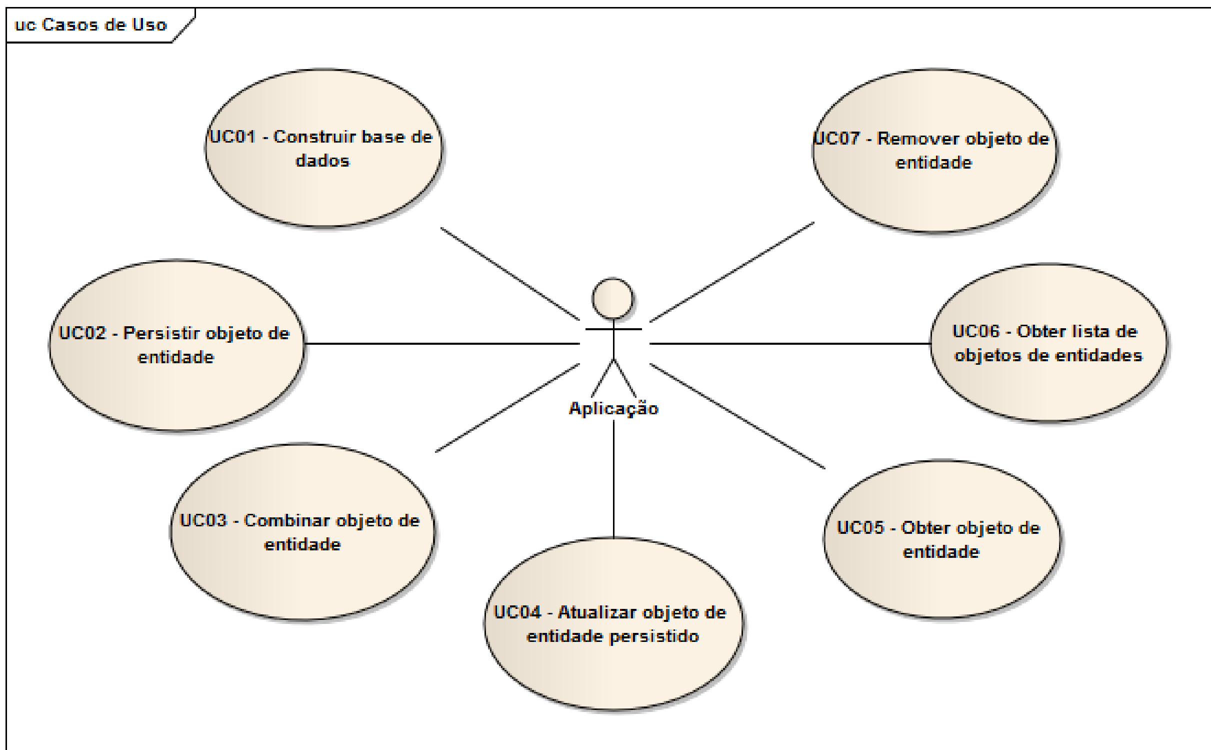


Figura 16 – Diagrama de casos de uso

"Um caso de uso representa um relato de uso de certa funcionalidade do sistema em questão, sem revelar a estrutura e o comportamento internos desse sistema" (MENEZES, 2007, p. 54). A seguir são explicados todos os casos de uso do diagrama. Foram abstraídos detalhes como análise das estratégias de mapeamento, herança, associações e geração de SQL, pois são detalhes que a aplicação não precisa se preocupar ao utilizar os casos de uso. O *framework* disponibiliza estas funcionalidades de forma transparente.

3.2.1.1 UC01 – Construir base de dados

O Quadro 8 mostra os detalhes do caso de uso `UC01 - Construir base de dados`, que é a criação das estruturas internas dentro da base de dados, como tabelas, construções e *sequences*. Considera-se que o elemento base de dados propriamente dito, por exemplo, o arquivo `.fdb` no caso do Firebird, já esteja criado e configurado no SGBD.

UC01 – Construir base de dados	
Pré-condições	Classes de entidades mapeadas; gerenciador de base de dados (TDatabaseManager) instanciado; conexão aberta no banco de dados; base de dados instanciada no SGBD.
Pós-condições	Base de dados criada e pronta para persistência de objetos.
Cenário	**Passos**
Principal	1. Aplicação invoca o método BuildDatabase do TDatabaseManager; 2. *Framework* cria todas as tabelas no banco de dados, incluindo campos, chaves primárias e chaves únicas; 3. *Framework* cria todas as chaves estrangeiras no banco de dados; 4. *Framework* cria todas as *sequences* no banco de dados, caso existam *sequences* mapeadas.

Quadro 8 – Caso de uso `UC01 - Construir base de dados`

3.2.1.2 UC02 – Persistir objeto de entidade

O Quadro 9 mostra os detalhes do caso de uso `UC02 - Persistir objeto de entidade`. Entenda-se como sendo a gravação de um objeto novo no banco de dados e sua inserção no mapa de objetos persistidos. A partir deste momento, o objeto passa a estar mapeado.

UC02 – Persistir objeto de entidade	
Pré-condições	Classes de entidades mapeadas; base de dados criada; gerenciador de objetos (TObjectManager) instanciado; conexão aberta no banco de dados.
Pós-condições	Objeto persistido no banco de dados; objeto mapeado no gerenciador de objetos.
Cenário	**Passos**
Principal	1. Aplicação instancia um novo objeto de entidade; 2. Aplicação atribui valores aos atributos ou *properties* do objeto criado; 3. Aplicação invoca o método Persist do TObjectManager, passando o objeto criado como parâmetro; 4. *Framework* valida o objeto que está sendo persistido; 5. *Framework* persiste o objeto, inserindo o mesmo no banco de dados; 6. *Framework* adiciona o objeto no mapa de objetos persistidos.
Exceção 1	No passo 4, caso o objeto já esteja persistido: 1. *Framework* gera exceção informando que o objeto já está persistido.
Exceção 2	No passo 4, caso o objeto já possua o atributo Id preenchido: 1. *Framework* gera exceção informando que não é permitido persistir um objeto com Id já preenchido.

Quadro 9 – Caso de uso `UC02 - Persistir objeto de entidade`

3.2.1.3 UC03 – Combinar objeto de entidade

O Quadro 10 mostra os detalhes do caso de uso `UC03 - Combinar objeto de entidade`. Entenda-se como sendo a combinação de um objeto novo que referencia um registro já gravado no banco de dados, com o valor correspondente já salvo. Considera-se que pode haver outro objeto já mapeado referente ao mesmo registro.

UC03 – Combinar objeto de entidade	
Pré-condições	Classes de entidades mapeadas; base de dados criada; gerenciador de objetos (TObjectManager) instanciado; conexão aberta no banco de dados; informações do objeto já persistidas no banco de dados.
Pós-condições	Objeto atualizado no banco de dados; objeto mapeado no gerenciador de objetos.
Cenário	**Passos**
Principal	1. Aplicação instancia um novo objeto de entidade; 2. Aplicação atribui valores aos atributos ou *properties* do objeto criado, preenchendo também o atributo Id indicando qual objeto deseja-se alterar; 3. Aplicação invoca o método Merge do TObjectManager, passando o objeto criado como parâmetro; 4. *Framework* valida o objeto que está sendo atualizado; 5. *Framework* aplica update no banco de dados; 6. *Framework* verifica se já existe um objeto com mesmo Id mapeado; 7. *Framework* adiciona o objeto no mapa de objetos persistidos; 8. *Framework* retorna a instância do objeto atualizado.
Fluxo alternativo 1	No passo 6, caso já exista outro objeto com mesmo Id mapeado: 1. *Framework* copia os valores de todos os atributos do objeto que já estava mapeado para o objeto que foi passado como parâmetro para atualizar; 2. *Framework* retorna a instância do objeto que já estava mapeado; 3. Aplicação descarta o objeto que foi criado inicialmente e passa a manipular o objeto que foi retornado pelo *framework*; 4. Finaliza o cenário principal.
Exceção 1	No passo 4, caso o objeto já esteja mapeado: 1. *Framework* gera exceção informando que o objeto já está mapeado.
Exceção 2	No passo 4, caso o objeto não possua o atributo Id preenchido: 1. *Framework* gera exceção informando que não é permitido atualizar um objeto sem o Id preenchido.

Quadro 10 – Caso de uso `UC03 - Combinar objeto de entidade`

3.2.1.4 UC04 - Atualizar objeto de entidade persistido

O Quadro 11 mostra os detalhes do caso de uso `UC04 - Atualizar objeto de entidade persistido`. Entenda-se como sendo a gravação dos valores atuais de um objeto

já persistido no banco de dados. Pode ser feita com objetos que foram persistidos, combinados ou obtidos. Não pode ser feita com objetos novos nem objetos removidos.

UC04 – Atualizar objeto de entidade persistido	
Pré-condições	Classes de entidades mapeadas; base de dados criada; gerenciador de objetos (`TObjectManager`) instanciado; conexão aberta no banco de dados; objeto já persistido e mapeado no gerenciador de objetos.
Pós-condições	Objeto atualizado no banco de dados.
Cenário	**Passos**
Principal	1. Aplicação atribui novos valores aos atributos ou *properties* de um objeto já persistido e mapeado; 2. Aplicação invoca o método `Flush` do `TObjectManager`; 3. *Framework* detecta os atributos que tiveram valor alterado e executa comando de `update` no banco de dados para aplicar as alterações;

Quadro 11 – Caso de uso `UC04 - Atualizar objeto de entidade persistido`

3.2.1.5 UC05 – Obter objeto de entidade

O Quadro 12 mostra os detalhes do caso de uso `UC05 - Obter objeto de entidade`. Entenda-se como sendo a seleção de um objeto de entidade do banco de dados, com os respectivos valores armazenados. Após obtido, o objeto está pronto para ser manipulado pela aplicação, acessando ou alterando os valores de seus atributos.

UC05 – Obter objeto de entidade	
Pré-condições	Classes de entidades mapeadas; base de dados criada; gerenciador de objetos (`TObjectManager`) instanciado; conexão aberta no banco de dados; base de dados com informações cadastradas.
Pós-condições	Objeto de entidade instanciado e carregado com os valores vindos do banco de dados; objeto mapeado no gerenciador de objetos.
Cenário	**Passos**
Principal	1. Aplicação invoca o método `Find` do `TObjectManager`, passando como parâmetro a classe e o valor do Id desejado; 2. *Framework* seleciona o registro do banco de dados correspondente ao Id informado, da tabela que representa a classe informada; 3. *Framework* instancia um objeto da classe informada e carrega todos os atributos ou *properties* mapeadas com os valores lidos do banco de dados; 4. *Framework* adiciona o objeto no mapa de objetos persistidos; 5. *Framework* retorna o objeto instanciado; 6. Aplicação manipula o objeto retornado.
Exceção 1	No passo 2, caso a seleção do banco de dados não retorne nenhum registro: 2.1. *Framework* retorna `nil`; 2.2. Retorna ao passo 6.

Quadro 12 – Caso de uso `UC05 - Obter objeto de entidade`

3.2.1.6 UC06 – Obter lista de objetos de entidades

O Quadro 13 mostra os detalhes do caso de uso `UC06 - Obter lista de objetos de entidades`. É semelhante ao caso de uso `UC05 - Obter objeto de entidade`, com a diferença que podem ser selecionados vários objetos da mesma entidade de uma só vez.

UC06 – Obter lista de objetos de entidades	
Pré-condições	Classes de entidades mapeadas; base de dados criada; gerenciador de objetos (`TObjectManager`) instanciado; conexão aberta no banco de dados; base de dados com informações cadastradas.
Pós-condições	Objetos de entidade instanciados e carregados com os valores vindos do banco de dados; objetos mapeados no gerenciador de objetos.
Cenário	**Passos**
Fluxo alternativo 1	1. Aplicação invoca o método `FindAll` do `TObjectManager`, passando como parâmetro a classe; 2. *Framework* seleciona os registros do banco de dados, de uma ou mais tabelas que representam a classe informada; 3. *Framework* instancia vários objetos da classe informada, um para cada registro resultante da seleção, e para cada objeto carrega todos os atributos ou *properties* mapeadas com os valores lidos do banco de dados; 4. *Framework* adiciona todos os objetos instanciados no mapa de objetos persistidos; 5. *Framework* retorna uma lista com todos os objetos instanciados; 6. Aplicação manipula os objetos retornados na lista.
Exceção 1	No passo 2, caso a seleção do banco de dados não retorne nenhum registro: 1. *Framework* retorna uma lista vazia; 2. Retorna ao passo 6.

Quadro 13 – Caso de uso `UC06 - Obter lista de objetos de entidades`

3.2.1.7 UC07 – Remover objeto de entidade

O Quadro 14 mostra os detalhes do caso de uso `UC07 - Remover objeto de entidade`. Entenda-se como sendo a deleção de um objeto no banco de dados e remoção do mesmo do gerenciador de objetos.

UC07 – Remover objetos de entidades	
Pré-condições	Classes de entidades mapeadas; base de dados criada; gerenciador de objetos (`TObjectManager`) instanciado; conexão aberta no banco de dados; objeto já persistido e mapeado no gerenciador de objetos.
Pós-condições	Objeto removido do banco de dados; objeto removido do mapeamento no gerenciador de objetos.
Cenário	**Passos**
Principal	1. Aplicação invoca o método `Remove` do `TObjectManager`, passando o objeto persistido como parâmetro; 2. *Framework* valida o objeto que está sendo removido; 3. *Framework* deleta o registro do banco de dados.
Exceção 1	No passo 2, caso o objeto ainda não esteja mapeado: 1. *Framework* gera exceção informando que o objeto precisa estar mapeado para remover.
Exceção 2	No passo 2, caso o objeto não possua o atributo `Id` preenchido: 1. *Framework* gera exceção informando que não é permitido remover um objeto sem o `Id` preenchido.
Exceção 3	No passo 3, caso o banco de dados dispare alguma exceção, como por exemplo, violação de chave estrangeira: 1. *Framework* repassa a exceção disparada pelo banco de dados para a aplicação.

Quadro 14 – Caso de uso `UC07 - Remover objetos de entidades`

3.2.2 Estrutura do *framework*

O *framework* foi dividido em duas partes principais: o motor de persistência e o gerenciador de objetos. O motor de persistência contém os componentes que juntos provêm a infraestrutura básica para a camada de persistência, realizando geração de código SQL em tempo de execução, tratando conexão com o SGBD e atuando na lógica de execução dos comandos na base de dados. Já o gerenciador de objetos disponibiliza uma interface de acesso ao *framework* de forma transparente para a aplicação, com métodos de interação que manipulam objetos de entidades de forma puramente orientada a objetos. A aplicação interage com o gerenciador de objetos, cujo papel também é controlar o ciclo de vida dos objetos de entidades e fazer *cache* de leitura em memória local.

Conforme se pode ver na Figura 17, o motor de persistência é divido em três componentes: os aplicadores de comandos, os geradores de SQL e o mediador de conexão. Já o gerenciador de objetos contém apenas um componente, chamado obviamente de gerenciador de objetos. Ambos utilizam outro componente chamado explorador de metadados, que foi destacado à parte e é responsável por coletar e manipular as informações

de metadados através de RTTI.

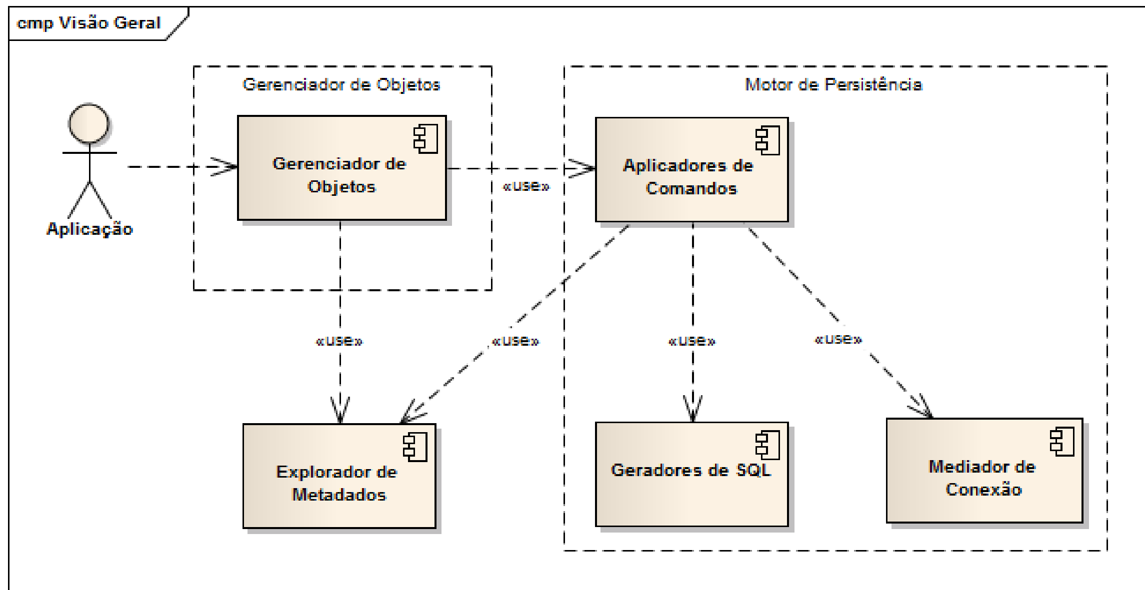


Figura 17 – Diagrama de componentes

3.2.2.1 Gerenciador de objetos

O componente Gerenciador de Objetos contém as classes responsáveis por prover uma interface de persistência transparente para a aplicação, com métodos que delegam responsabilidades para o motor de persistência. Além disso, devem controlar o ciclo de vida dos objetos de entidades e fazer *cache* de leitura em memória local. O diagrama de classes da Figura 18 mostra as classes deste componente.

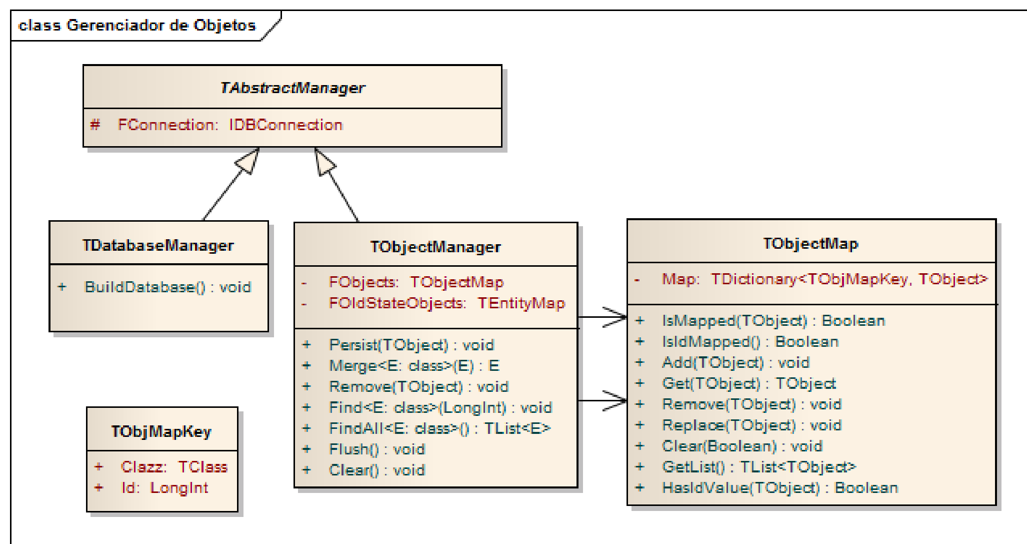


Figura 18 – Diagrama de classes do gerenciador de objetos

As classes `TObjectManager` e `TDatabaseManager` são a interface para a aplicação, e

ambas possuem a conexão com o SGBD que é utilizada para aplicação dos comandos. A `TDatabaseManager` possui método para criação da estrutura da base de dados baseada nas entidades registradas no *framework* e seus respectivos mapeamentos, criando todas as tabelas, chaves primárias, chaves únicas, chaves estrangeiras, sequências e demais estruturas que possam ser mapeadas. Já a classe `TObjectManager` disponibiliza métodos para persistir, atualizar, remover e obter objetos de entidades do banco de dados, que são respectivamente os métodos `Persist`, `Merge`, `Remove` e `Find`. O método `FindAll` serve para obter uma lista com todas as instâncias de uma determinada classe.

Cada objeto que for persistido, atualizado ou obtido através da classe `TObjectManager` deve ser armazenado em seu mapa de identidade `FObjects`. Todos os objetos que são inseridos neste mapa devem obrigatoriamente ter o atributo identificador preenchido. Após armazenado, o objeto passa a ser gerenciado pela classe `TObjectManager`, a qual deve detectar as alterações feitas nos atributos do objeto. O segundo mapa denominado `FOldStateObjects` serve para armazenar cópias dos objetos gerenciados, com o estado original do objeto no momento em que foi lido do SGBD, para que se possa comparar com o objeto gerenciado e determinar quais atributos foram alterados. O método `Flush` serve para aplicar no SGBD as alterações pendentes feitas nos objetos que estão gerenciados. O método `Clear` serve para limpar os dois mapas do `TObjectManager`, fazendo com que todas as instâncias de entidades mantidas pelo gerenciador deixem de ser gerenciadas.

A classe `TObjectMap` representa um mapa de objetos de entidades e deve implementar o padrão de projeto `IdentityMap`. Para isto ela possui um `TDictionary`, classe nativa da linguagem Delphi com funcionalidade semelhante ao `HashMap` da linguagem Java. A chave do dicionário é uma combinação da classe da entidade com o seu identificador, encapsulados na classe `TObjMapKey`, e o valor do dicionário é o próprio objeto de entidade. Além disto, a classe `TObjectMap` possui uma série de funções utilitárias para que o seu conteúdo possa ser manipulado adequadamente pelo `TObjectManager`.

3.2.2.2 Aplicadores de comandos

Os Aplicadores de Comandos são as classes responsáveis pela lógica de execução de comandos no SGBD. Os comandos são inicialmente montados em uma estrutura orientada a objetos que define tabelas, construções, campos, filtragens, junções e projeções que o

comando deve realizar. Esta estrutura é passada como parâmetro para o gerador de SQL que a traduz para código SQL em *strings* utilizando o dialeto do banco específico que varia de um gerador para outro. O código SQL é então executado no banco, e pode ter ou não parâmetros. Nos comandos de consulta como `select`, também é responsabilidade do aplicador de comando interpretar o retorno da consulta e carregar o objeto, lista ou árvore de objetos a partir das informações lidas do SGBD. Para cada comando que é executado no banco, é possível interceptar a execução do comando através de *listeners* que podem ser adicionados. Tais *listeners* podem ser utilizados para geração de *logs*, por exemplo.

O diagrama de classes da Figura 19 exibe as classes deste componente.

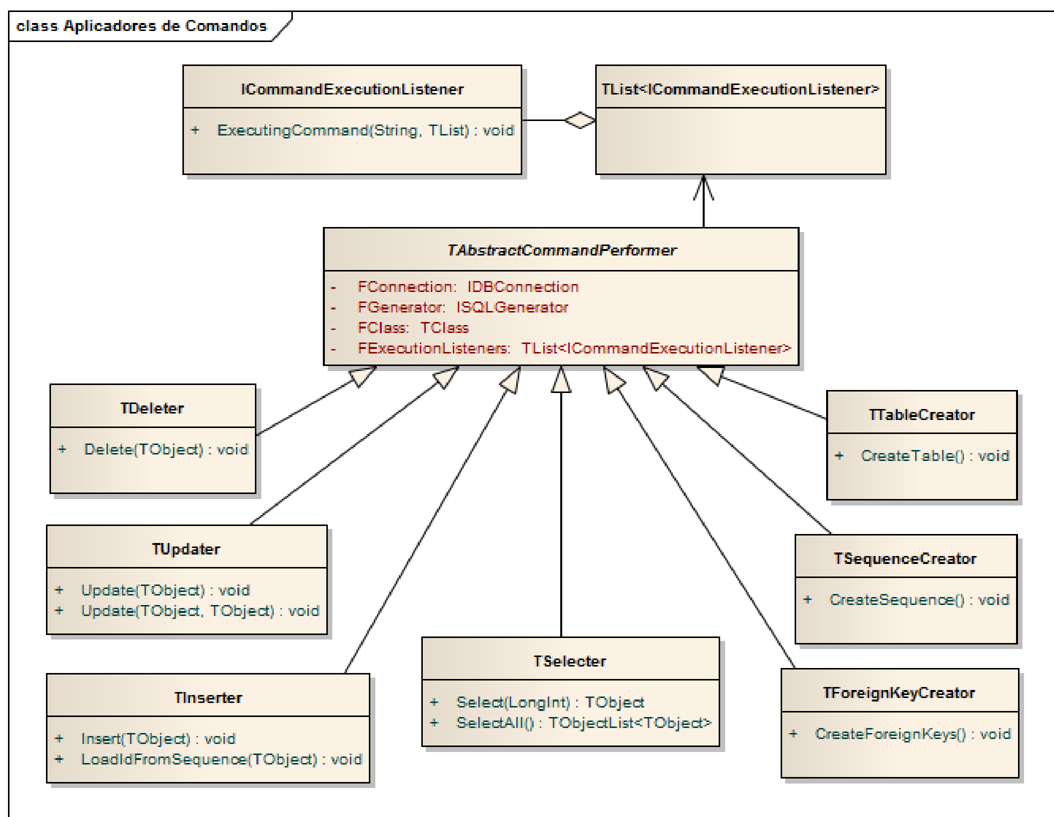


Figura 19 – Diagrama de classes dos aplicadores de comandos

Para cada tipo de comando existe uma classe concreta responsável, a fim de separar as diferentes lógicas. As classes `TDeleter`, `TUpdater`, `TInserter` e `TSelecter` são utilizadas pelo `TObjectManager` e aplicam comandos em *Data Manipulation Language* (DML[7]) enquanto as classes `TTableCreator`, `TSequenceCreator` e `TForeignKeyCreator` são utilizadas pelo `TDatabaseManager` e aplicam comandos em *Data Definition Language*

[7] Segundo Leme (2008), DML é a linguagem de manipulação de dados, que permite especificar operações de recuperação e alterações dos dados do banco de dados.

(DDL[8]).

Todas as classes aplicadoras de comandos descendem de uma classe abstrata chamada `TAbstractCommandPerformer`, herdando os seguintes atributos:

a) a conexão com o SGBD, por onde será aplicado o comando;
b) uma instância do gerador de SQL apropriada para o SGBD específico ao qual o sistema está conectado, que será utilizado para geração do código SQL;
c) a referência para a classe de entidade para a qual deve ser gerado o comando;
d) a lista de *listeners* que serão disparados para cada comando executado.

O *listener* de execução de comandos não é uma classe e sim uma interface que define apenas o método `ExecutingCommand`. Desta forma os *listeners* podem ser implementados com mais flexibilidade.

3.2.2.3 Geradores de SQL

Este componente define os geradores de código SQL, que são responsáveis por interpretar as estruturas de comandos recebidas dos Aplicadores de Comandos e traduzi-las para código SQL. O diagrama de classes da Figura 20 mostra as classes deste componente.

---

[8] Conforme Leme (2008), DDL é a linguagem de definição de dados que descreve a estrutura do banco de dados. Possui comandos de criação, alteração e exclusão de tabelas e visões. Gera um catálogo a partir da descrição dos dados.

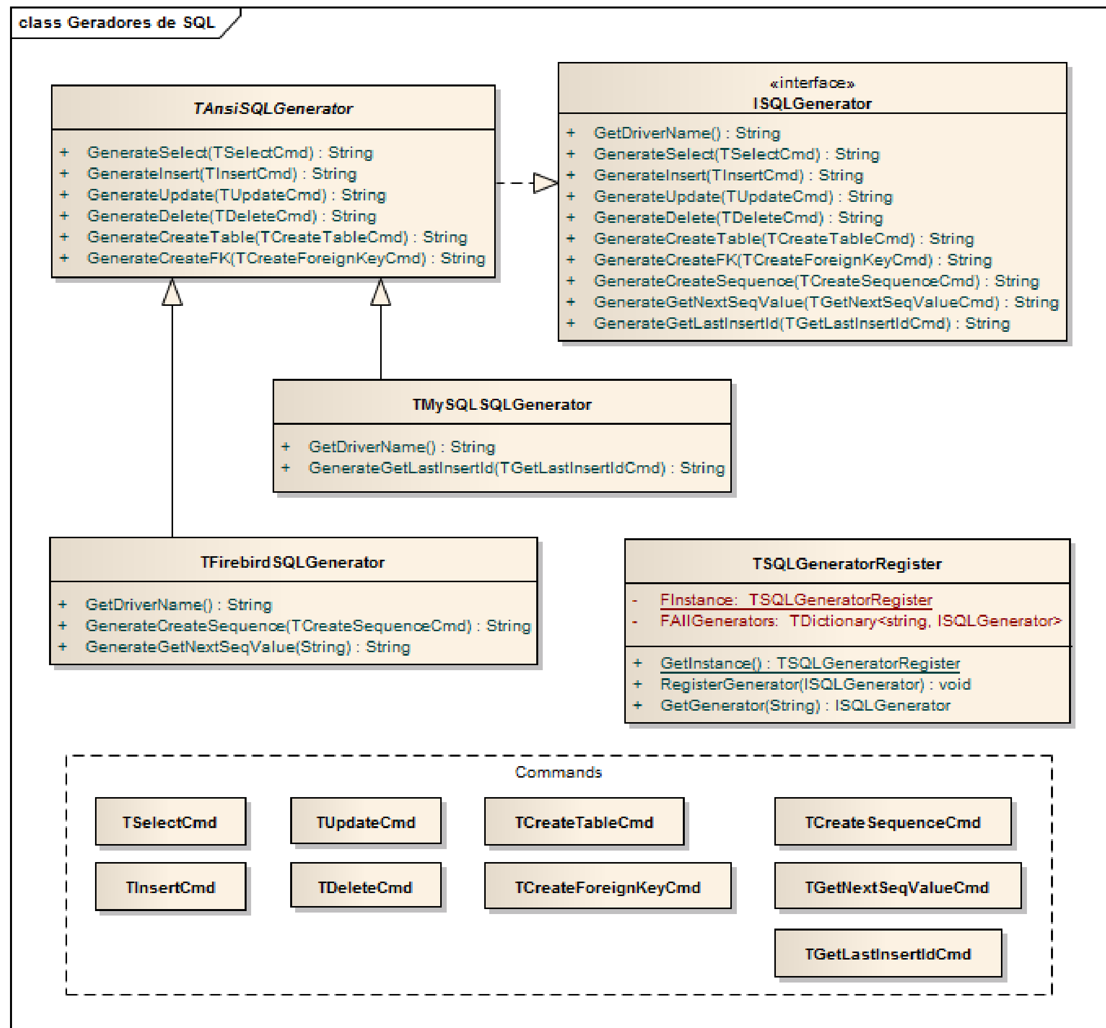


Figura 20 – Diagrama de classes dos geradores de SQL

A interface `ISQLGenerator` define todos os métodos disponíveis para geração de SQL, que cada gerador é obrigado a implementar. Esta abstração foi criada baseada no padrão de projeto Strategy, a fim de tornar o *framework* facilmente extensível adicionando suporte para diferentes SGBDs. Os Aplicadores de Comandos conhecem somente a `ISQLGenerator`, ignorando suas implementações.

Para cada SGBD suportado pelo *framework*, existe uma classe concreta que implementa `ISQLGenerator`. Como muitos bancos de dados possuem linguagens SQL similares, existem partes do código de geração de SQL que são repetitivas entre as classes dos geradores. Para resolver este problema foi criada uma classe intermediária chamada `TAnsiSQLGenerator` que reúne esta lógica em comum. Desta forma, evitou-se duplicidade no código fonte do *framework* e delegou-se às classes correspondentes a cada SGBD apenas as especificidades de cada banco.

O registro de todas as diferentes implementações de geradores de SQL é feita pela classe `TSQLGeneratorRegister`, que contém um dicionário com as instâncias dos geradores,

cada uma identificada por uma *string* que representa o nome do *driver* do gerador. Conforme declarado na interface `ISQLGenerator`, cada gerador deve implementar um método `GetDriverName` que simplesmente deve retornar uma *string* que identifica para qual *driver* este gerador foi implementado, como por exemplo `'Firebird'`.

Quando os Aplicadores de Comandos são instanciados, eles requisitam através da função `GetGenerator` da classe `TSQLGeneratorRegister` a instância do gerador de SQL correspondente ao banco de dados atual, passando como parâmetro o nome do *driver* obtido do objeto de conexão com o SGBD. Caso não haja nenhum gerador correspondente ao *driver* informado, é gerada uma exceção.

Para limitar em apenas uma instância a classe `TSQLGeneratorRegister` e criar um único ponto de acesso a ela, foi utilizado o padrão de projeto Singleton. Todas as classes concretas de geradores de SQL se auto-registram através do método `RegisterGenerator` da classe `TSQLGeneratorRegister`.

As classes que são exibidas na parte inferior do diagrama, na seção `Commands`, representam os comandos que são instanciados e montados pelos Aplicadores de Comandos e passados como parâmetro para os métodos de geração dos geradores de SQL. Seus atributos não são detalhados para que a especificação não fique extensa demais. Essas classes não deverão ter regras de negócio e sim apenas armazenam dados para geração dos comandos em uma estrutura organizada.

3.2.2.4 Mediador de conexão

Neste componente estão definidas as interfaces e classes que interagem com a API do *driver* do banco de dados, atuando como mediador entre o *framework* e os componentes de acesso direto ao SGBD. Foram definidas três interfaces principais:

a) `IDBConnection`: representa o objeto de conexão com o banco de dados, com métodos para conectar, desconectar, verificar conexão aberta, criar *statements* e obter o nome do *driver*;

b) `IDBStatement`: é o objeto por onde são efetivamente executados os comandos no SGBD, com métodos para atribuir o comando SQL, atribuir os parâmetros para o comando, executar comandos imperativos e executar comandos de seleção. O método de execução de comandos de seleção retorna um objeto da interface

IDBResultSet;

c) IDBResultSet: representa o resultado tabular de um comando de seleção, com os registros lidos do banco de dados. Possui métodos para ler os valores dos campos, navegar para o próximo registro e verificar se está vazio.

Este componente é utilizado apenas pelos Aplicadores de Comandos, os quais utilizam somente estas três interfaces, sem conhecer suas implementações. Desta forma, o *framework* é mais facilmente extensível, pois é possível substituir não somente o *driver* de conexão com o SGBD como também o conjunto de componentes internos que o acessam. Tais componentes poderiam ser, por exemplo, DBExpress ou Borland *Database Engine* (BDE).

O diagrama de classes da Figura 21 apresenta as classes deste componente.

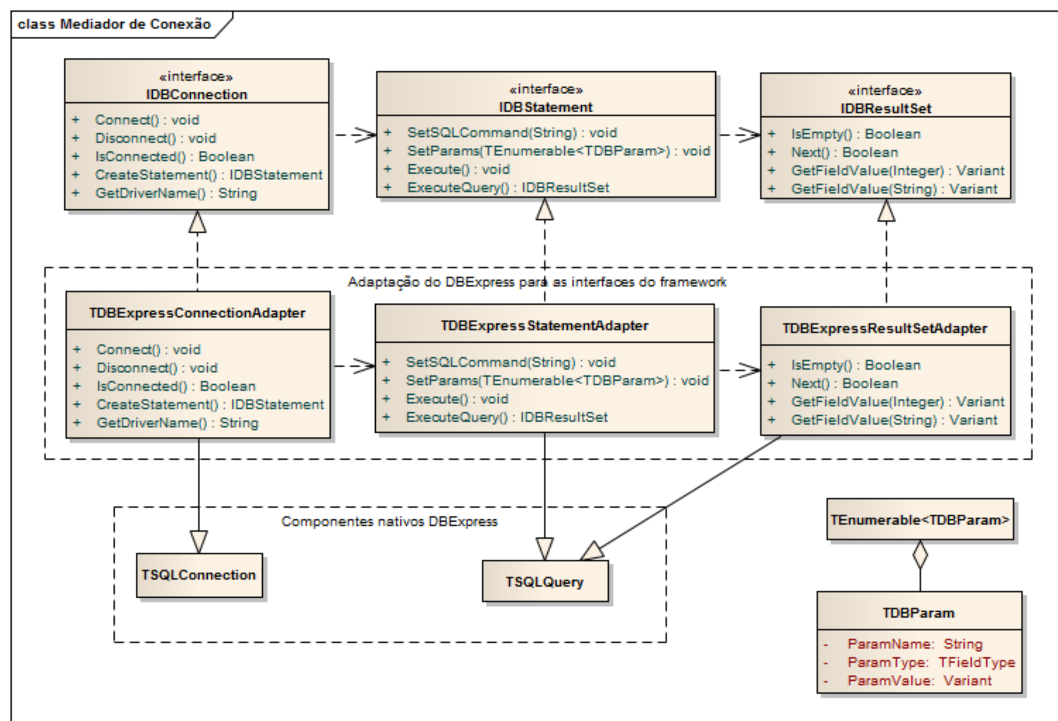


Figura 21 – Diagrama de classes do mediador de conexão

As três interfaces definidas neste componente podem ter sua lógica de interação com o SGBD implementadas por completo em classes novas, ou então reutilizarem esta lógica de outras classes que já realizam este serviço. Para isto podem ser reaproveitadas bibliotecas de terceiros ou componentes nativos do Delphi aplicando o padrão de projeto Adapter.

Para o funcionamento do *framework*, foram utilizados os componentes da biblioteca DBExpress, que são componente nativos no Delphi 2010. Foi utilizado o padrão Adapter utilizando herança, dando origem às classes de adaptação TDBExpressConnectionAdapter, TDBExpressStatementAdapter e TDBExpressResultSetAdapter. Tais classes descendem das classes do DBExpress, que são TSQLConnection e TSQLQuery, e implementam as interfaces definidas pelo *framework*. A classe TSQLQuery foi estendida tanto para atender

IDBStatement quanto para IDBResultSet pelo fato de ela por si só implementar as funcionalidades definidas por estas duas interfaces.

A classe TDBParam serve apenas para encapsular um parâmetro que é passado para um *statement* definindo seu nome, tipo e valor. A classe TEnumerable<T> é nativa do Delphi e representa uma coleção genérica enumerável com elementos do tipo T.

3.2.2.5 Explorador de metadados

Este componente define as classes que são responsáveis por explorar os metadados de mapeamento objeto-relacional declarados nas classes de entidades da aplicação, fornecendo informações detalhadas de mapeamento para os demais componentes do *framework*, além de interagir com os objetos de entidades.

As três classes definidas neste componente são utilitárias e implementam o padrão de projeto Singleton para criar um ponto de acesso a elas, e para que possa haver apenas um objeto de cada instanciado no sistema. Estas classes fazem *cache* das informações de mapeamento para garantir um bom desempenho. O atributo FInstance e o método GetInstance fazem parte da implementação do padrão Singleton. O diagrama de classes da Figura 22 mostra as classes deste componente.

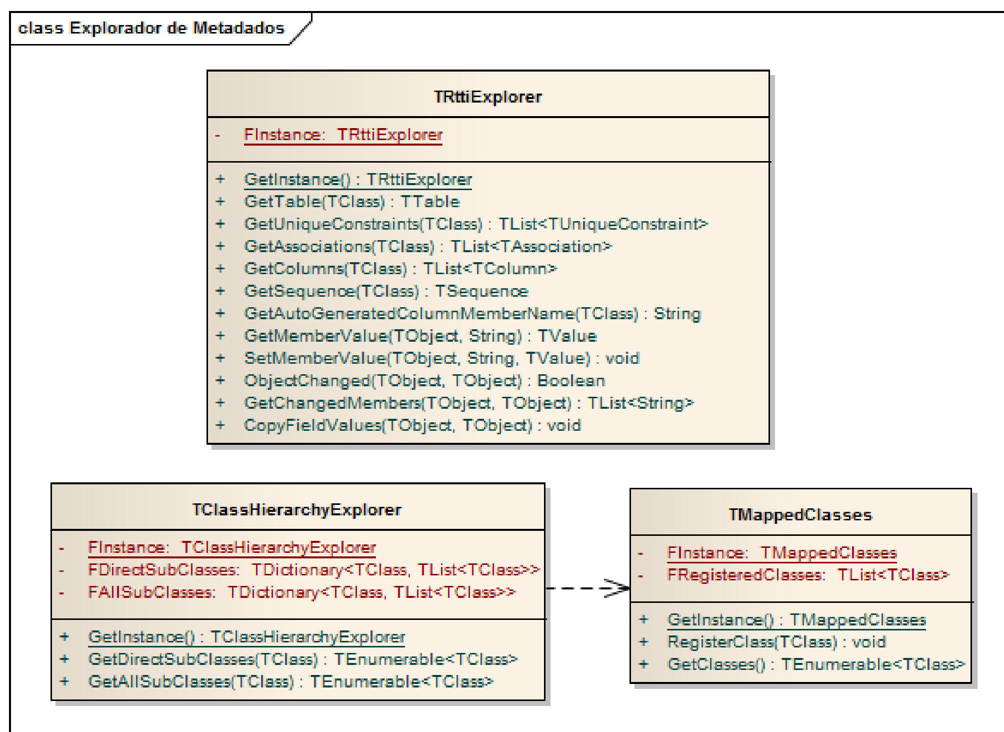


Figura 22 – Diagrama de classes do explorador de metadados

A classe `TRttiExplorer` serve para centralizar a interação com a API do RTTI do Delphi, obtendo as informações de mapeamento objeto-relacional das classes de entidades e interagindo com os objetos de entidades através de RTTI. Possui métodos para:

a) obter a tabela à qual a classe de entidade está mapeada;
b) obter as chaves únicas da classe de entidade;
c) obter as associações da classe de entidade com outras classes de entidades;
d) obter as colunas às quais a classe de entidade está mapeada, cada coluna mapeada a um atributo ou *property* da classe;
e) obter a *sequence* para geração de valor para o campo id da classe de entidade;
f) obter o atributo ou *property* auto-gerada de uma classe de entidade;
g) obter o valor do atributo ou *property* de um objeto de entidade;
h) atribuir um determinado valor a um atributo ou *property* de um objeto de entidade;
i) comparar dois objetos da mesma classe para determinar quais atributos ou *properties* estão com valores diferentes;
j) copiar valores dos atributos ou *properties* de um objeto para outro, sendo que os dois objetos são da mesma classe.

A classe `TMappedClasses` é responsável por realizar o registro de todas as classes de entidades mapeadas no *framework*, para que se possa ter uma lista de todas as classes mapeadas. O *framework* possui por padrão a habilidade de registrar automaticamente todas as classes de entidades mapeadas que estão sendo utilizadas pela aplicação. Em situações especiais, esta configuração pode ser desabilitada e as classes podem ser registradas manualmente através da `TMappedClasses`. A `TMappedClasses` possui métodos para:

a) registrar uma classe de entidade no *framework*;
b) obter a lista de classes registradas no *framework*.

A classe `TClassHierarchyExplorer` serve para fornecer informações de herança entre classes de entidades. No Delphi, a partir de uma classe, é possível descobrir qual é sua classe pai através do atributo nativo `ClassParent`, porém não existe uma alternativa fácil para obter suas classes filhas. O objetivo maior da classe `TClassHierarchyExplorer` é fornecer esta navegabilidade de cima para baixo dentro da hierarquia de classes, mas apenas para as classes de entidades mapeadas. Portanto, ela utiliza a `TMappedClasses` para obter a lista com todas as classes mapeadas e então cruza as informações de parentesco entre estas classes para montar uma representação das árvores de herança. A classe `TClassHierarchyExplorer` possui métodos para:

a) obter, a partir de uma classe, a lista de todas as classes descendentes, sejam filhas,

netas, ou qualquer nível inferior;

b) obter, a partir de uma classe, a lista de todas as suas subclasses diretas, ou seja, apenas as que estão no primeiro nível abaixo da classe que foi passada como parâmetro.

3.2.3 Interação entre as classes

O diagrama de seqüência da Figura 23 mostra a interação entre as principais classes do *framework* para efetuar a persistência de um objeto de entidade. São mostradas em nível macro as principais trocas de mensagens que são feitas desde a chamada do método `persist` no `TObjectManager`, que é a interface do *framework* com a aplicação, até a execução do comando de inserção no banco de dados. Alguns detalhes de implementação foram omitidos, bem como regras de negócio. Foi emitida, por exemplo, a exceção a ser gerada quando o objeto a ser persistido já se encontra persistido.

A execução do comando de inserção no banco de dados é feita no método `Execute` da interface `IDBStatement`, e sua implementação pode variar. A implementação padrão desta interface é a classe `TDBExpressStatementAdapter`, onde o método `Execute` dispara o método `ExecSQL` da classe `TSQLQuery`.

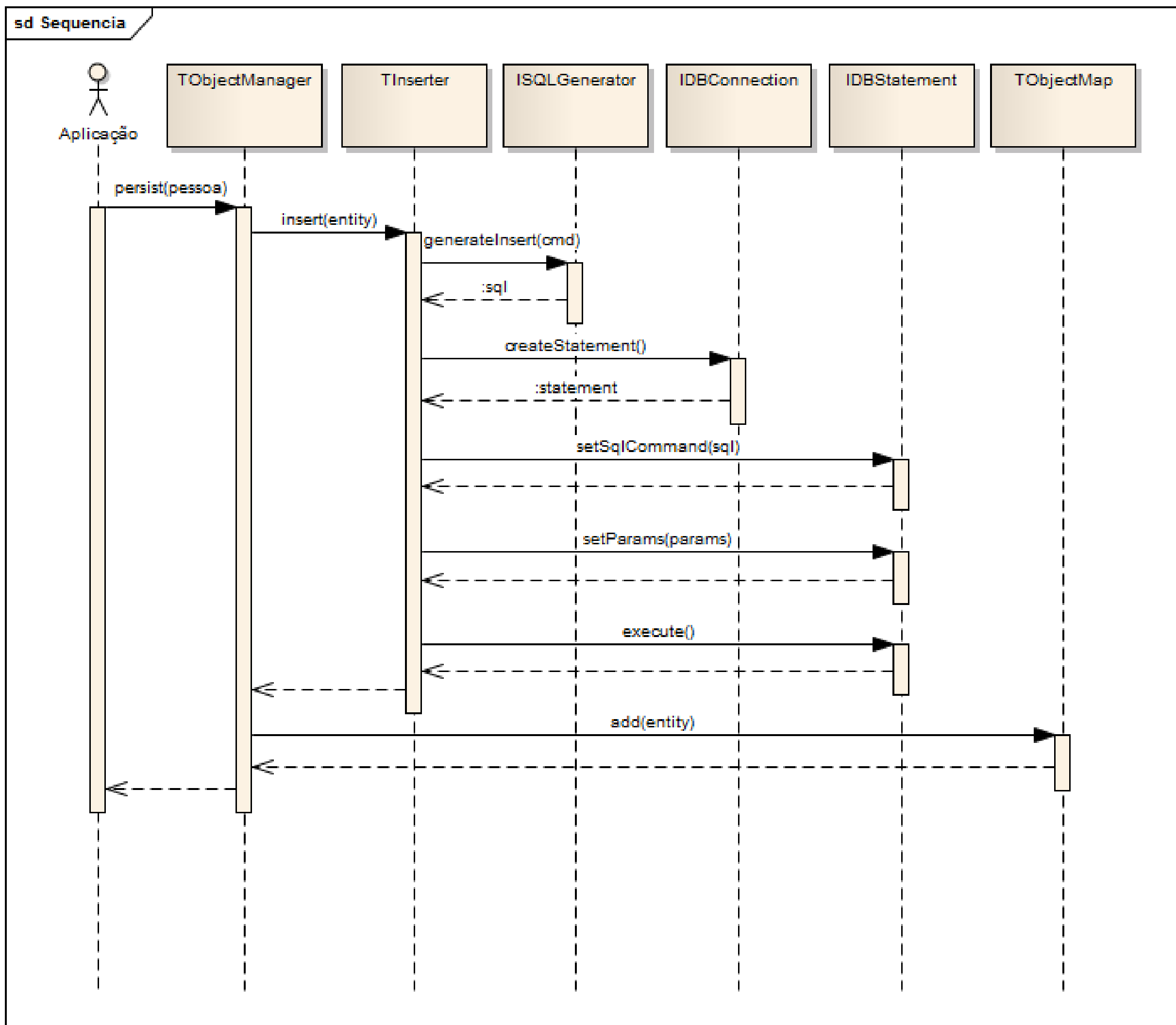


Figura 23 – Diagrama de seqüência do *framework* para o método `Persist`

3.3 MODELO DE ANOTAÇÃO PARA MAPEAMENTO

Para que possa ser realizado o mapeamento objeto-relacional entre as entidades da aplicação e o banco de dados, foi criado um modelo de anotação baseado em Atributos Customizados, recurso do Delphi que foi descrito na fundamentação teórica, tópico 2.4. Basicamente, a elaboração deste modelo foi a criação de um conjunto de classes descendentes da classe nativa `TCustomAttribute`. Muitos dos atributos criados foram inspirados nas anotações da especificação JPA, implementada pelo *framework* Hibernate.

São mostradas na Figura 24 todas as classes que foram criadas. A seguir, detalha-se o significado de cada classe para o mapeamento. Por fim, é mostrado um exemplo de como

pode ser utilizado este modelo de anotação para mapear uma classe de entidade na aplicação.

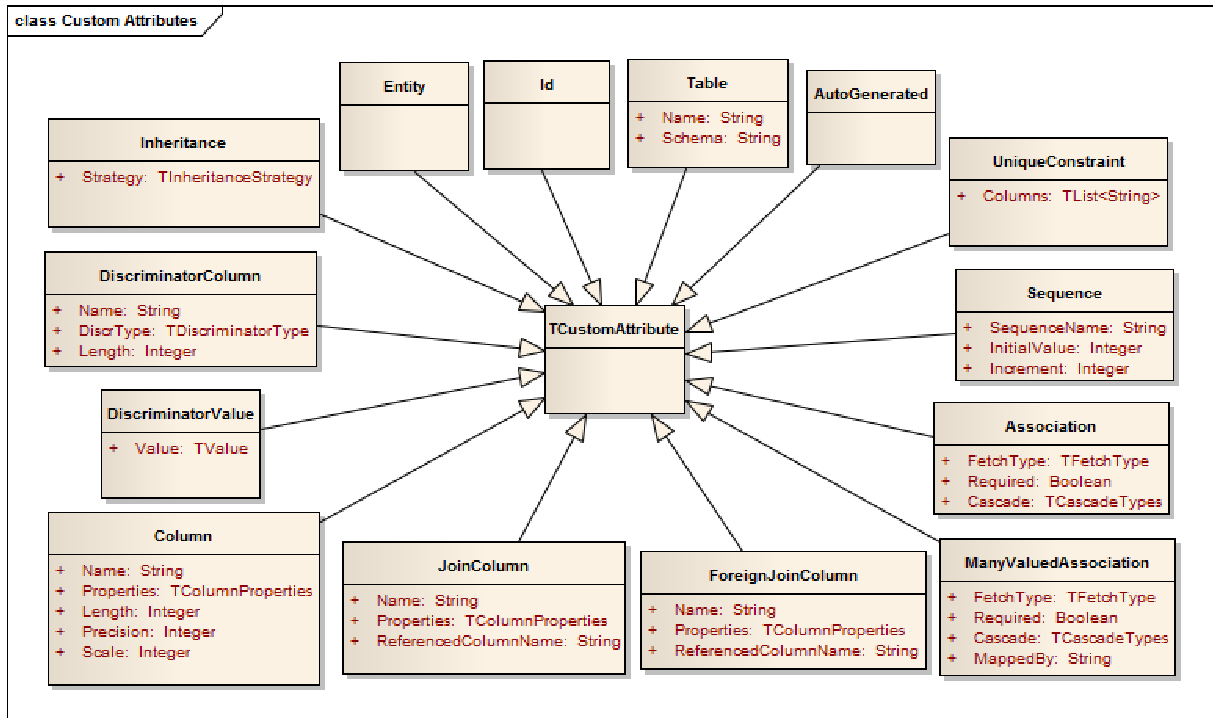


Figura 24 – Classes do modelo de anotação

Segue abaixo a lista de classes que foram criadas, com seu respectivo significado:

a) `Entity`: Define uma classe como classe de entidade. Uma classe de entidade é uma classe que pode ser persistida. Este atributo deve ser anotado acima da declaração da classe e seu construtor não possui nenhum parâmetro;

b) `Id`: Define o campo que representa o identificador desta classe. Tal campo deve ter um valor único, sem significado para o mundo real, e deve ser de algum tipo numérico inteiro como `LongInt` ou `Integer`. Com isto mapeia-se este campo para a chave primária da tabela no banco de dados. A anotação do atributo `Id` pode ser feita acima de um atributo ou *property* da classe, também sem parâmetros. Toda classe deve possuir um e somente um `Id`. Se o `Id` for anotado na classe pai não se deve anotar o `Id` na classe filha;

c) `Table`: Define em qual tabela do banco de dados a classe atual será persistida. Este atributo deve ser anotado acima da declaração da classe e possui dois parâmetros: o nome da tabela e o esquema desta tabela no banco de dados;

d) `Autogenerated`: Define um campo da classe como gerado automaticamente. Com isto, o *framework* não gerará um valor para este atributo e irá delegar ao banco de dados a geração do seu valor através do recurso de auto-incremento de uma coluna. Pode ser anotado acima de um atributo ou *property*. Em SGBDs que não

suportam colunas auto-incrementadas, esta anotação é ignorada;

e) `UniqueConstraint`: Define uma chave única composta para a tabela no banco de dados. Não é necessário declarar uma chave única para o campo que foi definido como `Id`, porque esta chave já é automaticamente mapeada como chave primária. Além disto, chaves únicas compostas por apenas um campo podem ser mapeadas mais facilmente na anotação do atributo `Column`, descrita abaixo. O atributo `UniqueConstraint` pode ser anotado uma ou mais vezes acima de cada classe, e seus parâmetros são uma lista de *strings* com os nomes dos campos da chave única;

f) `Sequence`: Define uma *sequence* para geração de valores para um campo da classe. Recebe três parâmetros: o nome da *sequence*, o valor inicial e o valor de incremento. Pode ser anotado acima de um atributo ou *property*. Esta anotação é utilizada apenas ao utilizar SGBDs que suportam *sequences*, caso contrário ela é ignorada pelo *framework* e não influencia no mapeamento;

g) `Association`: Define uma associação da classe atual para outra classe de entidade, que pode ser inclusive a própria classe. Para todo atributo de uma classe de entidade cujo tipo é também uma classe de entidade, deve-se anotar uma `Association` mapeando esta associação para que o *framework* saiba como tratá-la. O parâmetro `FetchType` é o tipo de carregamento, que pode ser `ftLazy` ou `ftEager`. `ftLazy` significa que o atributo será carregado somente na primeira vez que for requisitado, implementando o padrão de projeto Lazy Load. `ftEager` significa que será carregando já no momento em que o objeto atual for carregado. O parâmetro `Required` indica se o atributo é requerido para a classe e irá influenciar no modo como o sistema faz junções nos comandos `select`. Se o campo é requerido, será feito `inner join`, caso contrário será feito `left join`. E o último parâmetro `Cascade` define um conjunto de ações a serem feitas em cascata para esta associação. Estas ações podem ser `Persist`, `Merge`, `Remove`, `Refresh`. Uma ação de cascata do tipo `Persist` indica, por exemplo, que quando o objeto atual for gravado no banco de dados, antes disto será gravado o objeto que está atribuído nesta associação. O atributo `Association` pode ser anotado acima de um atributo ou *property*;

h) `ManyValuedAssociation`: Define uma associação para outra classe em um atributo multivalorado, como por exemplo, uma lista de itens em uma classe de

nota fiscal. Possui todas as propriedades de uma Association e mais uma propriedade opcional chamada MappedBy, na qual deve ser informado o nome do atributo ou *property* da classe associada que faz a associação inversa correspondente, nos casos onde existe associação bidirecional. Em associações unidirecionais, o parâmetro MappedBy não deve ser informado. Para que o *framework* consiga determinar a classe que está sendo mapeada em uma ManyValuedAssociation, o atributo ou *property* anotado deve utilizar uma lista genérica como TList<T>;

i) JoinColumn: É utilizada junto com as anotações de Association e define uma das colunas que compõe a chave estrangeira para a tabela da classe associada. Para cada anotação de Association, deve ser anotada uma ou mais JoinColumns, dependendo da quantidade de colunas que compõe a chave estrangeira. As anotações de Association e JoinColumn são parte da implementação do padrão de projeto Foreign Key Mapping e devem ser anotadas juntas, uma abaixo da outra;

j) ForeignJoinColumn: É utilizada junto com as anotações de ManyValuedAssociation e define uma das colunas que compõe a chave estrangeira da tabela associada para a tabela atual, caso a associação seja unidirecional. Em associações bidirecionais este atributo não é necessário, pois o mapeamento é feito com o MappedBy. As anotações de ManyValuedAssociation e ForeignJoinColumn também são parte da implementação do padrão de projeto Foreign Key Mapping;

k) Column: Define o mapeamento entre uma coluna da tabela no banco de dados e um atributo ou *property* da classe de entidade. Seus parâmetros são: nome da coluna, propriedades da coluna, comprimento, precisão e escala. O comprimento precisa ser informado somente em colunas do tipo string. A precisão e a escala, somente em colunas numéricas de tipos reais como Double ou Currency. As propriedades da coluna são um conjunto de propriedades que podem ser Unique, Required, DontInsert e DontUpdate. Unique define a coluna como sendo única, mapeando uma chave única no banco de dados. Required define que a coluna é not null no banco de dados. DontInsert e DontUpdate definem, respectivamente, que esta coluna não será incluída em comandos de insert e de update a serem efetuados no SGBD para esta entidade. O atributo Column pode ser anotado acima de um

atributo ou *property*;

l) `Inheritance`: Define mapeamento de herança, indicando uma classe como sendo raiz de uma hierarquia de classes de entidades. Deve ser mapeada acima da classe que é raiz da hierarquia, informando o parâmetro Strategy que define a estratégia de herança, que pode ser `TablePerConcreteClass`, `SingleTable` ou `JoinedTables`. `TablePerConcreteClass` indica que cada classe concreta da hierarquia deverá ser mapeada para uma diferente tabela no banco de dados, a qual contém todas as colunas da classe incluindo as colunas herdadas. `SingleTable` indica que todas as classes da hierarquia serão mapeadas para apenas uma tabela no banco, que contém todas as colunas possíveis de todas as classes. Para esta estratégia, é necessário utilizar os atributos `DiscriminatorColumn` e `DiscriminatorValue` para realizar o mapeamento corretamente. A terceira estratégia `JoinedTables` índica que será criada uma tabela para cada classe da hierarquia, mesmo para as classes abstratas, sendo que cada tabela contém somente os campos declarados em cada classe correspondente, além da chave primária e uma chave estrangeira para a tabela que representa a classe pai;

m) `DiscriminatorColumn`: Aplica-se a herança do tipo `SingleTable` e define uma coluna que será utilizada para discriminar os registros da tabela, indicando a qual classe cada registro pertence. Os parâmetros são nome da coluna, tipo do discriminador e comprimento. O tipo do discriminador pode ser `String` ou `Integer` e o comprimento é opcional e aplica-se somente se o tipo do discriminador for `String`. O atributo `DiscriminatorColumn` deve obrigatoriamente ser anotado acima da classe raiz da hierarquia;

n) `DiscriminatorValue`: Aplica-se a herança do tipo `SingleTable` e define o valor discriminativo da classe atual, para que o *framework* saiba determinar de qual classe é cada objeto lido da tabela única. Para cada classe concreta da hierarquia, deve ser anotado acima da declaração da classe informando o parâmetro que é o valor discriminativo, que pode ser `String` ou `Integer`. Cada classe concreta deverá ter um valor discriminativo diferente.

O Quadro 15 mostra um exemplo de como realizar as anotações em uma classe de entidade.

```
type
  [Entity]
  [Table('NOTAS_FISCAIS')]
  [Sequence('SEQ_NOTAS_FISCAIS')]
  [UniqueConstraint('NUMERO, ID_CLIENTE')]
  TNotaFiscal = class
  private
    FId: LongInt;
    FNumero: Integer;
    FEmissao: TDateTime;
    FCliente: TCliente;
    FItens: TList<TItemNota>;
  public
    [Id]
    [AutoGenerated]
    [Column('ID', [cpUnique, cpRequired])]
    property Id: LongInt read FId write FId;

    [Column('NUMERO', [cpRequired])]
    property Numero: Integer read FNumero write FNumero;

    [Column('DATA_EMISSAO', [cpRequired])]
    property Emissao: TDateTime read FEmissao write FEmissao;

    [Association(ftEager, orRequired, ctCascadeAll)]
    [JoinColumn('ID_CLIENTE')]
    property Cliente: TCliente read FCliente write FCliente;

    [ManyValuedAssociation(ftEager, orRequired, ctCascadeAll)]
    [ForeignJoinColumn('ID_NOTA_FISCAL', [cpRequired])]
    property Itens: TList<TItemNota> read FItens write FItens;
  end;
```

Quadro 15 – Exemplo de anotações em classe de entidade

Neste exemplo, foram utilizadas as seguintes classes de atributos customizados: `Enitity`, `Table`, `Sequence`, `UniqueConstraint`, `Id`, `AutoGenerated`, `Column`, `Association`, `JoinColumn`, `ManyValuedAssociation` e `ForeignJoinColumn`. Os atributos de colunas e associações foram anotados nas *properties* `Id`, `Numero`, `Emissao`, `Cliente` e `Itens`. O *framework* também permite anotá-los diretamente nos atributos da classe `FId`, `FNumero`, `FEmissao`, `FCliente` e `FItens`. Esta flexibilidade foi criada para o caso de o programador não querer declarar *properties* em suas classes para encapsular os atributos. Já os atributos `Entity`, `Table`, `Sequence` e `UniqueConstraint` devem obrigatoriamente ser anotados na classe.

No caso de herança entre classes, a classe filha pode herdar as anotações de `Table` e `Sequence` da classe pai. As classes de atributos `Inheritance`, `DiscriminatorColumn` e `DiscriminatorValue` são utilizadas apenas em casos de herança, por isto não aparecem no exemplo.

Para declarar estas anotações em sua classe, o programador deve adicionar a unidade `MetadataEngine.Attributes` na cláusula `uses` da unidade onde está sendo declarada a

classe. Sem isto o editor do Delphi não reconhecerá as classes de atributos customizados que estão sendo utilizados, e funcionalidades como *code completion* não funcionarão.

3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas na implementação do *framework*, as convenções seguidas, bem como detalhes da implementação dos componentes do *framework*. Por fim, é mostrada sua operacionalidade.

3.4.1 Técnicas e ferramentas utilizadas

A implementação do *framework* seguiu os conceitos da orientação a objetos, e foi separada em diferentes unidades e classes de acordo com as diferentes complexidades de implementação. Conforme previsto na especificação, a implementação fez uso dos padrões de projeto:

a) DAO, visto que o gerenciador de objetos é um objeto DAO;
b) Singleton, usado nas classes `TMappedClasses`, `TClassHierarchyExplorer`, `TSQLGeneratorRegister`, `TGlobalConfigs` e `TRttiExplorer`;
c) Adapter, usado para implementação do adaptador de conexão com o DBExpress;
d) Strategy, usado nos geradores de SQL para implementar suporte a vários SGBDs;
e) Identity Field, implementado através do atributo customizado `Id` e rotinas que o tratam, identificando unicamente os objetos;
f) Identity Map, implementado na classe `TObjectMap` que é utilizada pelo gerenciador de objetos;
g) Foreign Key Mapping, implementado com os atributos customizados `Association`, `ManyValuedAssociation` e suas rotinas de tratamento;
h) Lazy Load, implementado como estratégia de carregamento nos atributos `Association` e `ManyValuedAssociation`, utilizando a estrutura `Proxy<T>` e rotinas específicas de tratamento.

O *framework* foi implementado utilizando o ambiente de desenvolvimento

Embarcadero Delphi 2010 para Windows. Foram utilizados os componentes da *Visual Component Library* (VCL[9]) e *RunTime Library* (RTL[10]) do Delphi, além dos componentes DBExpress de acesso ao banco de dados, também nativos. Não foi utilizada nenhuma biblioteca que não fosse nativa, portanto todo o seu código fonte depende apenas do Delphi para compilar e executar. Alguns recursos que foram utilizados, como Atributos Customizados e a nova API do RTTI, estão disponíveis apenas a partir do Delphi 2010, portanto seu código fonte não pode ser facilmente portado para versões anteriores.

Durante a implementação do *framework*, foi utilizado o sistema de controle de versão Subversion para organizar as versões do código fonte. Como aplicativo cliente foi utilizado o TortoiseSVN. Segundo TIGRIS (2010), TortoiseSVN é uma ferramenta cliente de Subversion integrada ao Windows Explorer. Ela possui código fonte aberto e realiza controle de revisões, versões e fontes. Como não se trata de um *plugin* de integração com alguma IDE específica, pode ser utilizada idenpendentemente da ferramenta de desenvolvimento ou linguagem de programação.

Foram utilizados os SGBDs Firebird na versão 2.1 e MySQL na versão 5.1. Tanto o componente servidor quanto o componente cliente dos dois SGBDs foram instalados em uma máquina local. Para confirmar as informações gravadas pelo *framework* no banco de dados Firebird, foi utilizada a ferramenta de administração de banco de dados Flamerobin. No caso do MySQL, foi utilizada a ferramenta MySQL Query Browser.

3.4.2 Convenções seguidas

A implementação seguiu algumas convenções para tornar o código fonte organizado, legível e reutilizável. Seguem algumas convenções utilizadas:

a) o código fonte foi implementado utilizando unidades, classes, identificadores e mensagens em idioma inglês, para abrir a possibilidade de tornar o código fonte público e poder ser editado por outras pessoas, inclusive de outros países;
b) para cada tipo de exceção que o *framework* gera foi criada uma classe de exceção

---

[9] Conforme Embarcadero (2009), VCL é um conjunto de componentes criado pela Borland para desenvolvimento na linguagem Delphi. Ela inclui componentes visuais, não-visuais e classes utilitárias para construção de aplicações para Windows, web, bancos de dados e console.

[10] Segundo Embarcadero (2009), por trás da VCL o Delphi fornece uma vasta biblioteca de rotinas e classes, chamada de RTL, que provê funcionalidades comuns necessárias em qualquer tipo de aplicação.

distinta. Todas essas classes descendem da classe `EOPFBaseException` definida no *framework*, que por sua vez descende da classe nativa `Exception`. As classes de exceções foram reunidas em diferentes *units* de acordo com seu contexto;

c) todas as alocações e deslocações dinâmicas de objetos foram implementadas de forma a evitar vazamentos de memória. Foi utilizado o gerenciador de memória[11] nativo do Delphi para detectar vazamentos de memória no *framework*, e foram resolvidos todos os vazamentos encontrados durante os testes;

d) sempre que possível, foram utilizadas estruturas de dados nativas da linguagem, a fim de tornar o código fonte do *framework* mais facilmente portável para as próximas versões do Delphi, sem precisar realizar migrações de estruturas específicas;

e) sempre que possível, foram utilizadas coleções genéricas como `TList<T>`, `TObjectList<T>` e `TDictionary<TKey, TValue>`, declaradas na unidade `Generics.Collections`, ao invés das coleções antigas da unidade `Classes`. Estas estruturas genéricas estão disponíveis no Delphi a partir da versão 2009.

3.4.3 Implementação dos componentes do *framework*

Os componentes do *framework* foram implementados conforme apresentado na especificação. Nessa seção são mostrados alguns trechos do código fonte das principais classes de cada componente.

No componente Gerenciador de Objetos, a principal classe é a `TObjectManager`. O Quadro 16 mostra a implementação do método `Persist` da classe `TObjectManager`.

---

[11] Conforme Embarcadero (2006), a partir do Delphi 2006 existe um gerenciador de memória nativo que permite reportar todos os blocos de memória que não foram liberados. Esta funcionalidade pode ser habilitada configurando a variável `ReportMemoryLeaksOnShutdown` para `True`. Com isto, um relatório de vazamentos de memória aparece em uma mensagem na tela ao fechar a aplicação.

```
procedure TObjectManager.Persist(Entity: TObject);
var
  Inserter: TInserter;
begin
  if FObjects.IsMapped(Entity) then
    raise EObjectAlreadyPersistent.Create(Entity);

  if FObjects.HasIdValue(Entity) then
    raise ECannotPersististWithId.Create(Entity);

  CascadePersist(Entity);

  Inserter := FCommandFactory.GetCommand<TInserter>(Entity.ClassType);
  try
    if TRttiExplorer.GetInstance.HasSequence(Entity.ClassType, True) then
      Inserter.LoadIdFromSequence(Entity);

    Inserter.Insert(Entity);

    FObjects.Add(Entity);
    FOldStateObjects.Add(TRttiExplorer.GetInstance.Clone(Entity));
  finally
    Inserter.Free;
  end;
end;
```

Quadro 16 – Implementação do método `Persist`

No componente Explorador de Metadados, a principal classe é a `TRttiExplorer`. O Quadro 17 mostra a implementação do método `GetChangedMembers` da classe `TRttiExplorer`. Este método serve para detectar quais atributos ou *properties* foram alterados em um objeto. O gerenciador de objetos chama este método passando como parâmetro duas versões do mesmo objeto: a versão atual e a cópia que foi criada refletindo o estado em que se encontrava no banco de dados.

```
function TRttiExplorer.GetChangedMembers(OriginalObj,
  DirtyObj: TObject): TList<string>;
var
  RttiType: TRttiType;
  Member: TRttiMember;
  OriginalValue, DirtyValue: TValue;
  C: TColumn;
begin
  Assert(OriginalObj.ClassType = DirtyObj.ClassType);

  RttiType := FContext.GetType(OriginalObj.ClassType);
  Result := TList<string>.Create;

  for C in GetColumns(OriginalObj.ClassType, True) do
  begin
    if not C.IsDiscriminator then
    begin
      case C.MemberType of
        mtField:    Member := RttiType.GetField(C.ClassMemberName);
        mtProperty: Member := RttiType.GetProperty(C.ClassMemberName);
      else
        Member := nil;
      end;

      OriginalValue := GetMemberValue(OriginalObj, Member.Name);
      DirtyValue := GetMemberValue(DirtyObj, Member.Name);

      if not ValueIsEqual(OriginalValue, DirtyValue) then
        Result.Add(Member.Name);
    end;
  end;
end;
```

Quadro 17 – Implementação do método `GetChangerMembers`

No componente Aplicadores de Comandos, não existe uma classe principal e sim várias classes aplicadoras de comandos, todas descendentes de `TAbstractCommandPerformer`. O Quadro 18 mostra o código do método `Delete` da classe `TDeleter`.

```
procedure TDeleter.Delete(Entity: TObject);
var
  Command: TDeleteCommand;
  SQLField: TSQLWhereField;
  IdColValue: TValue;
  IdValue: Variant;
  ParamName, SQL: string;
  Params: TObjectList<TDBParam>;
begin
  Command := TDeleteCommand.Create;
  try
    SQLField := TSQLWhereField.Create(CreateSQLTable,FIdCol.Name,woEqual);

    Command.Table := CreateSQLTable;
    Command.WhereFields.Add(SQLField);

    SQL := FGenerator.GenerateDelete(Command);
    IdColValue := TRttiExplorer.GetInstance.GetColumnValue(Entity,FIdCol);
    IdValue := TUtils.ValueToVariant(IdColValue);
    ParamName := FGenerator.GetDefaultWhereParamName(SQLField);

    Params := TObjectList<TDBParam>.Create;
    try
      Params.Add(TDBParam.Create(ParamName,FIdCol.FieldType,IdValue));
      Execute(SQL,Params,False);
    finally
      Params.Free;
    end;
  finally
    Command.Free;
  end;
end;
```

Quadro 18 – Implementação do método `Delete`

No componente Geradores de SQL, foram criadas classes geradoras de comandos SQL e classes auxiliares. A maior classe deste componente é a `TAnsiSQLGenerator`. O Quadro 19 mostra a implementação do método `GenerateInsert` da classe `TAnsiSQLGenerator`.

```
function TAnsiSQLGenerator.GenerateInsert(Cmd: TInsertCommand): string;
var
  FieldNames: TList<string>;
  I: Integer;
begin
  Assert(Cmd.InsertFields.Count > 0);

  FieldNames := GetFieldNames(Cmd.InsertFields, False);
  try
    Result := 'INSERT INTO ';

    if Cmd.Table.Schema <> '' then
      Result := Result + Cmd.Table.Schema + '.';

    Result := Result + Cmd.Table.Name + ' ('#13#10'  ' +
      TUtils.ConcatStrings(FieldNames) + ')'#13#10'VALUES ('#13#10'  ';

    for I := 0 to Cmd.InsertFields.Count - 1 do
    begin
      if I > 0 then
        Result := Result + ', ';
      Result := Result + ':' + GetDefaultParamName(Cmd.InsertFields[I]);
    end;

    Result := Result + ');';
  finally
    FieldNames.Free;
  end;
end;
```

Quadro 19 – Implementação do método `GenerateInsert`

Ainda no componente Geradores de SQL foram criadas classes de comandos, que servem para encapsular as informações necessárias para geração de um comando SQL. Alguns exemplos destas classes são `TSelectCommand`, `TInsertCommand`, `TUpdateCommand`, `TDeleteCommand` e `TCreateTableCommand`, entre outras. Todas as classes que encapsulam comandos de DML descendem de uma classe abstrata chamada `TDMLCommand`. O Quadro 20 mostra um trecho de código fonte de algumas destas classes para exemplificar.

```
type
  TDMLCommand = class abstract
  private
    FTable: TSQLTable;
  public
    property Table: TSQLTable read FTable write FTable;
  end;

  TSelectCommand = class(TDMLCommand)
  private
    FSelectFields: TObjectList<TSQLSelectField>;
    FJoins: TObjectList<TSQLJoin>;
    FWhereFields: TObjectList<TSQLWhereField>;
    FGroupByFields: TObjectList<TSQLField>;
    FOrderByFields: TObjectList<TSQLOrderField>;
  public
    // Getters e setters dos atributos privados ocultados...
  end;

  TInsertCommand = class(TDMLCommand)
  private
    FInsertFields: TObjectList<TSQLField>;
  public
    // Getters e setters dos atributos privados ocultados...
  end;
```

Quadro 20 – Classes de comandos nos geradores de SQL

Para que se possa entender melhor, são mostrados no Quadro 21 trechos de código das classes auxiliares que compõe as classes de comandos.

```
type
  TSQLTable = class
  private
    FName: string;
    FSchema: string;
    FAlias: string;
  public
    // Getters e setters dos atributos privados ocultados...
  end;

  TSQLField = class
  private
    FTable: TSQLTable;
    FField: string;
  public
    // Getters e setters dos atributos privados ocultados...
  end;

  TJoinType = (jtInner, jtLeft);

  TSQLJoinSegment = class
  private
    FPKField: TSQLField;
    FFKField: TSQLField;
  public
    // Getters e setters dos atributos privados ocultados...
  end;

  TSQLJoin = class
  private
    FJoinType: TJoinType;
    FSegments: TObjectList<TSQLJoinSegment>;
  public
    // Getters e setters dos atributos privados ocultados...
  end;

  // Demais classes auxiliares...
```

Quadro 21 – Classes auxiliares nos geradores de SQL

3.4.4 Funcionalidades e estratégias implementadas

A seguir são apresentadas algumas funcionalidades e estratégias que foram implementadas no *framework*.

3.4.4.1 Mapeamento de herança

Foi implementado suporte a mapeamento de herança. Para tal, foram implementadas todas as três estratégias de herança explicadas na fundamentação teórica:

a) herança de tabela por classe concreta;

b) herança de tabela única;

c) herança de tabela por subclasse.

Tais estratégias podem ser configuradas ao realizar a anotação do atributo customizado `Inheritance` nas classes de entidades, passando como parâmetro a enumeração que representa a estratégia desejada: `isTablePerConcreteClass`, `isSingleTable` ou `isJoinedTables`.

3.4.4.2 Mapeamento de associações

Para mapeamento de associações entre entidades, foram implementados os seguintes mapeamentos:

a) muitos-para-um;

b) um-para-muitos.

Estes mapeamentos representam, respectivamente, associações diretas e agregações. Associações do tipo um-para-um não foram implementadas, mas é possível utilizar uma associação muitos-para-um e limitar a cardinalidade na própria aplicação. Associações do tipo muitos-para-muitos também não foram implementadas, mas podem ser decompostas em associações um-para-muitos dos dois lados da associação, simulando uma associação muitos-para-muitos. Associações do tipo muitos-para-muitos com tabela associativa podem ser simuladas da mesma forma, mas neste caso é necessário criar explicitamente a classe associativa. O *framework* não está preparado para criá-la automaticamente.

Para utilizar associações do tipo muitos-para-um, utiliza-se o atributo customizado `Association` anotado acima do atributo ou *property* da classe que referencia a outra classe de entidade. Para utilizar associações do tipo um-para-muitos, utiliza-se o atributo customizado `ManyValuedAssociation`, anotado acima do atributo ou *property* da classe que armazena a lista de objetos. O mapeamento de associações no *framework* reflete a utilização do padrão de projeto Foreign Key Mapping. O *framework* mapeia as anotações de `Association` e `ManyValuedAssociation` para chaves estrangeiras no banco de dados.

3.4.4.3 *Object Identifier*

Para garantir a unicidade dos objetos, foram implementados tratamentos para a regra do *Object Identifier* utilizando os conceitos do padrão de projeto Identity Field. Como o *framework* não deve alterar nem adicionar atributos às classes de entidades da aplicação, resolve-se da seguinte maneira: o programador deve declarar em suas classes um atributo que armazenará o identificador do objeto, e anotar acima deste o atributo customizado `Id`. A partir deste momento, o *framework* associa este atributo à chave primária da tabela.

Para o funcionamento correto da persistência desta entidade, o atributo `Id` não deve ter significado e deve ser de algum tipo numérico inteiro, como `Integer` ou `LongInt`. O valor deste campo é gerado automaticamente pelo *framework* no momento apropriado para a persistência, mas para isto deve-se utilizar a anotação `Sequence` ou `AutogeneratedValue`. O *framework* atualmente não suporta um `Id` composto por mais de um campo.

3.4.4.4 Nullable

Para que a aplicação possa manipular o valor `null` em atributos de entidades que são de tipos primitivos, foi implementado um tipo especial chamado Nullable, semelhante à estrutura Nullable da plataforma .NET, citada na fundamentação teórica. O Nullable é um tipo genérico e é declarado como `Nullable<T>`. Por ser genérico, pode-se decompor em vários outros tipos como `Nullable<string>`, `Nullable<boolean>`, `Nullable<TDateTime>`, entre outros. Não há restrições para o tipo `T`, podendo aceitar qualquer tipo, inclusive classes como `TCliente` por exemplo. Porém, para classes não há a necessidade de usar a estrutura Nullable, já que a elas pode ser atribuído o valor `nil`.

O tipo `Nullable<T>` foi implementado como um *record* no Delphi. O Quadro 22 mostra o *record* com seus atributos e métodos.

```
type
  Nullable<T> = record
  private
    FValue: T;
    FHasValue: Boolean; // Default False

    class procedure CheckNullOperation(Left, Right: Nullable<T>); static;
    function GetIsNull: Boolean;
    function GetValue: T;
    procedure SetValue(const Value: T);
    function GetValueOrDefault: T;

  public
    constructor Create(Value: T); overload;

    property HasValue: Boolean read FHasValue;
    property IsNull: Boolean read GetIsNull;
    property Value: T read GetValue write SetValue;
    property ValueOrDefault: T read GetValueOrDefault;

    class operator Implicit(Value: TNullRecord): Nullable<T>;

    class operator Implicit(Value: T): Nullable<T>;
    class operator Implicit(Value: Nullable<T>): T;

    class operator Explicit(Value: T): Nullable<T>;
    class operator Explicit(Value: Nullable<T>): T;

    class operator Equal(Left, Right: Nullable<T>): Boolean;
    class operator NotEqual(Left, Right: Nullable<T>): Boolean;

    class operator GreaterThan(Left, Right: Nullable<T>): Boolean;
    class operator GreaterThanOrEqual(Left, Right: Nullable<T>): Boolean;

    class operator LessThan(Left, Right: Nullable<T>): Boolean;
    class operator LessThanOrEqual(Left, Right: Nullable<T>): Boolean;
  end;
```

Quadro 22 – Implementação da estrutura Nullable

Basicamente o `Nullable<T>` contém um atributo para o valor, `FValue`, do tipo `T`, e um atributo para indicar a presença ou não de valor, `FHasValue`, do tipo `Boolean`. Se `FHasValue` for `False`, o valor armazenado por `FValue` é indefinido. Neste estado, a estrutura `Nullable<T>` não permite que o valor seja acessado, pois não há valor. Qualquer tentativa de leitura ou utilização do valor indefinido gerará uma exceção.

Os atributos internos do `Nullable<T>` foram colocados como privados. Foram utilizados métodos dinâmicos e estáticos para realizar o controle de acesso. Também foram implementados operadores para conversão implícita e explícita para o tipo `T`, para o valor `null`, e operadores de comparação. No Delphi, estes operadores definem qual comportamento será executado ao comparar duas variáveis do tipo `Nullable<T>` ou então comparar uma variável do tipo `Nullable<T>` com uma do tipo `T`, por exemplo. Grande parte destas funcionalidades utilizadas, como tipos genéricos, *records* com métodos, operadores, e

definição de visibilidades em *records* não existiam em versões anteriores ao Delphi 2009.

A razão do `Nullable<T>` ter sido criado como *record* ao invés de classe é a instanciação implícita. No Delphi, toda classe precisa ser explicitamente instanciada antes de ser utilizada, e destruída logo após. Já nos *records* isso não é necessário, pois eles são implicitamente instanciados. Como não faria muito sentido o programador ter que instanciar e destruir cada atributo anulável de uma classe, esta alternativa foi muito mais confortável e transparente.

3.4.4.5 Lazy Load

O *framework* utiliza o *Lazy Load* no carregamento de entidades associadas. Não é utilizado *Lazy Load* em atributos que não são associações, ou seja, todos os demais atributos de uma entidade são carregados ao mesmo tempo quando o *framework* carrega a entidade. Além disso, para utilização do *Lazy Load*, deve-se definir a estratégia de carregamento `ftLazy` ao anotar a associação, caso contrário a entidade associada será também carregada no mesmo momento. Para a implementação do padrão de projeto *Lazy Load* foi criada uma estrutura genérica especial declarada como `Proxy<T>`. As alternativas de implementação do *Lazy Load* são citadas na fundamentação teórica, das quais foram escolhidas o *proxy* virtual e o armazenador de valor. A solução elaborada é uma junção destas duas alternativas.

A Quadro 23 mostra o código fonte da estrutura `Proxy<T>`.

```
type
  Proxy<T: class> = record
  private
    FValue: T;
    FLoaded: Boolean;
    FId: LongInt;
    FManager: TObjectManager;
    function GetValue: T;
    procedure SetValue(const Value: T);
  public
    property Value: T read GetValue write SetValue;

    class operator Equal(Left, Right: Proxy<T>): Boolean;
    class operator NotEqual(Left, Right: Proxy<T>): Boolean;
  end;
```

Quadro 23 – Implementação da estrutura Proxy

O `Proxy<T>` é um *record* genérico que encapsula um valor do tipo `T`, onde `T` obrigatoriamente deve ser uma classe. Na prática, `T` deve ser uma classe de entidade mapeada, como `TCliente` por exemplo. Cada `Proxy<T>` contém um atributo onde é armazenado o valor

depois de carregado, um indicador de valor já carregado, o Id do objeto a ser carregado e uma referência ao gerenciador de objetos.

O valor da entidade é acessado através da *property* `Value`. No momento em que esta *property* é lida, o *proxy* verifica se o valor já foi carregado. Caso ainda não tenha sido carregado, ele carrega o objeto de entidade utilizando o gerenciador de objetos, passando como parâmetro o Id do objeto. A classe do objeto a ser requisitado é a classe genérica `T`.

A implementação dos métodos `GetValue` e `SetValue` é mostrada na Quadro 24.

```
function Proxy<T>.GetValue: T;
begin
  if not FLoaded then
  begin
    if FId <= 0 then
      FValue := nil
    else
    begin
      if FManager = nil then
        raise EEntityManagerNotSet.Create(T);
      FValue := FManager.Find<T>(FId);
    end;
    FLoaded := True;
  end;
  Result := FValue;
end;

procedure Proxy<T>.SetValue(const Value: T);
begin
  FValue := Value;
  FLoaded := True;
end;
```

Quadro 24 – Implementação do *Lazy Load* no *Proxy*

3.4.5 Operacionalidade da implementação

Esta seção apresenta a operacionalidade do *framework* para construção de aplicações em Delphi. Como o *framework* não disponibiliza uma interface gráfica, a operacionalidade é demonstrada através de trechos de código fonte exemplificando a sua utilização. Como o objetivo aqui é puramente didático, é demonstrado um estudo de caso de desenvolvimento de uma aplicação hipotética: um aplicativo para gerenciar uma biblioteca de músicas e clipes armazenados em computador. A aplicação será desenvolvida utilizando o padrão de projeto MVC. Para este exemplo considera-se que a aplicação tenha sete classes, conforme o diagrama da Figura 25.

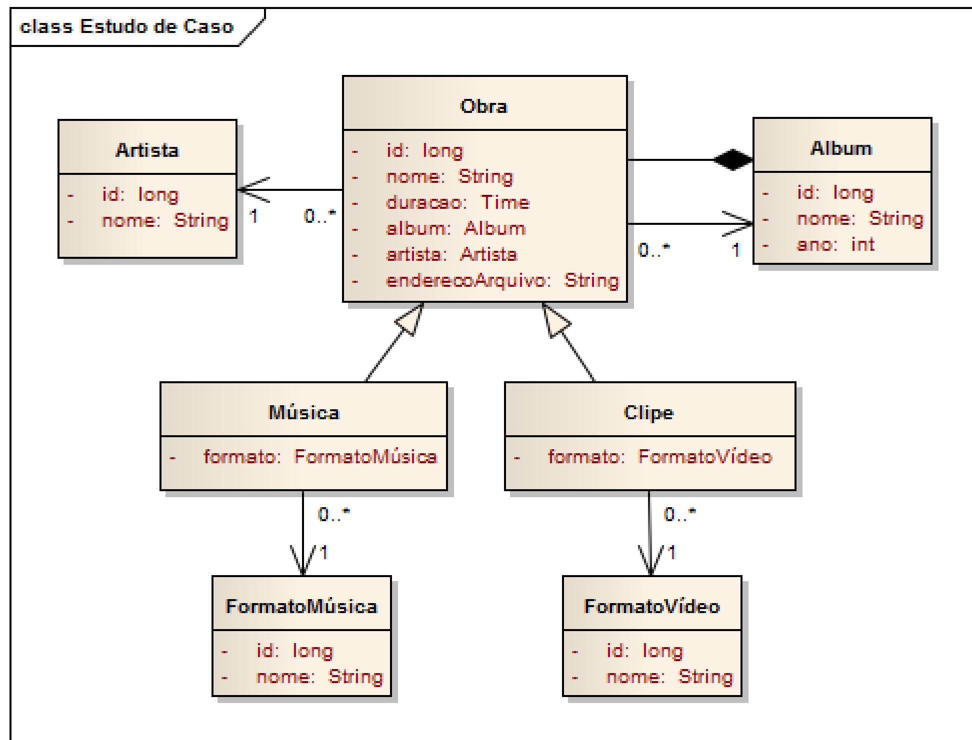


Figura 25 – Diagrama de classes do estudo de caso

Para iniciar o desenvolvimento da aplicação, o primeiro passo é abrir o ambiente Delphi e criar um novo projeto do tipo VCL Forms Application, conforme Figura 26. Deve-se salvar o código fonte do projeto em uma pasta apropriada e importar para este projeto o código fonte do *framework*, com todas as suas unidades, para que possam ser utilizadas ao longo da aplicação.

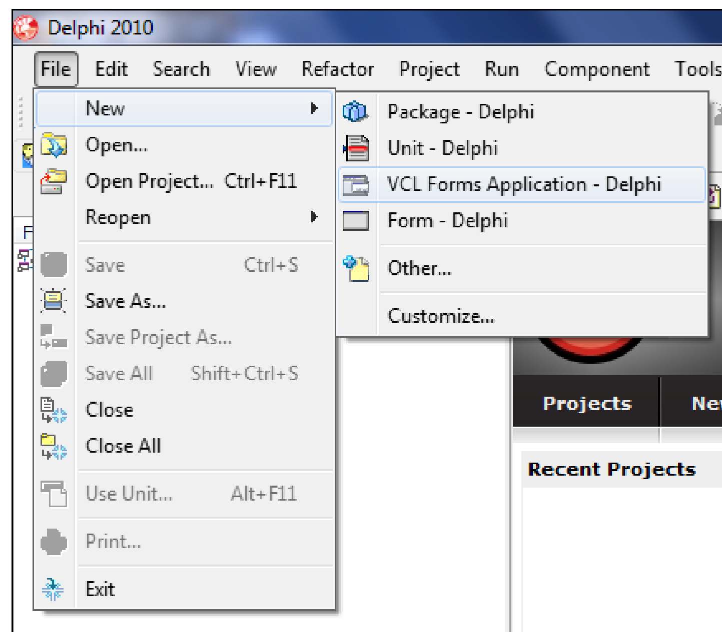


Figura 26 – Criação de novo projeto no Delphi

Depois disto, implementa-se as classes de entidades, que representam o modelo do padrão MVC. Estas classes são implementadas utilizando a linguagem Delphi normal,

declarando atributos e *properties* para manter os valores das entidades. Recomenda-se salvar as classes de entidades em uma pasta própria para separar o modelo do restante da aplicação. A classe `Artista`, por exemplo, ficaria parecida com o código mostrado na Figura 27. Até aqui ainda não foram necessários os componentes do *framework*.

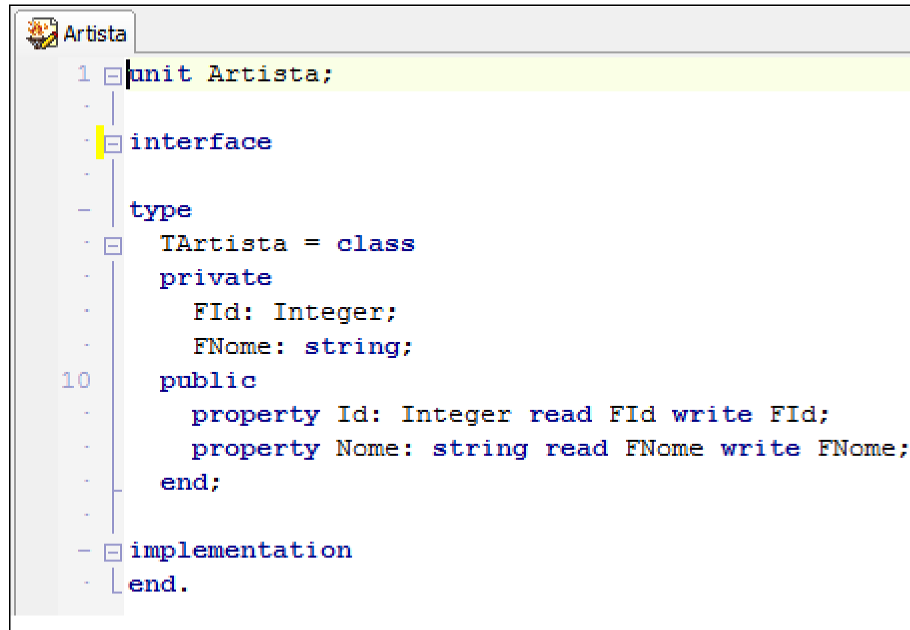


Figura 27 – Implementação da classe `TArtista`

Com as classes implementadas, começa-se a utilizar o *framework*. O próximo passo é a anotação dos atributos customizados nas classes definidas. Para isto, deve-se adicionar a unidade `MetadataEngine.Attributes` na cláusula `uses` da unidade onde está declarada a classe. Nesta etapa, deve-se definir um nome para a tabela onde será persistida a classe, bem como os nomes e definições das colunas, e também a forma de geração do “Id”, neste caso *sequence*. A implementação ficará conforme a Figura 28.

```
unit Artista;

interface

uses
  MetadataEngine.Attributes;

type
  [Entity]
  [Table('ARTISTAS')]
  [Sequence('SEQ_ARTISTAS')]
  TArtista = class
  private
    FId: Integer;
    FNome: string;
  public
    [Id]
    [Column('ID', [cpUnique, cpRequired, cpDontUpdate])]
    property Id: Integer read FId write FId;

    [Column('NOME', [cpRequired], 100)]
    property Nome: string read FNome write FNome;
  end;
```

Figura 28 – Classe `TArtista` mapeada no *framework*

A Figura 29 demonstra como ficaria a classe `Obra`. Foi utilizado o tipo especial `Nullable` nos campos que não são obrigatórios, importando a unidade `SpecialTypes.Nullable`. Também foi utilizado o tipo especial `Proxy` nos campos que possuem carga tardia, importando a unidade `SpecialTypes.Proxy`. Nos campos que armazenam o álbum e o artista, foi anotada a `Association` com tipo de carregamento `ftLazy`, para que seus valores sejam carregados somente na primeira utilização.

Como a estrutura `Proxy` requer que o valor seja acessado através da *property* `Value`, pode ser incômodo utilizar `Obra.Artista.Value` para acessar o objeto de `Artista` por todo o código fonte da aplicação. Caso não haja tratamento especial a ser efetuado nos métodos de acesso a este atributo, isto pode ser melhorado da seguinte forma:

a) coloca-se as anotações de `Association` e `JoinColumn` diretamente no atributo ao invés de na *property*;

b) declara-se a *property* como sendo do tipo `TArtista` ao invés de `Proxy<TArtista>`. Nota-se que o atributo continua com o tipo `Proxy<TArtista>`;

c) implementa-se os métodos de acesso, `GetArtista` e `SetArtista`, nos quais atribui-se ou obtém-se o valor através da *property* `Value` do `Proxy`.

A classe `Obra` é a classe raiz da hierarquia. Por isso, foi anotado o atributo

customizado `Inheritance` especificando a estratégia de herança. Neste caso, foi adotada a estratégia de tabela única, `isSingleTable`. Ao utilizar esta estratégia, deve-se adicionar a anotação de um `DiscriminatorColumn` na classe raiz, e um `DiscriminatorValue` em cada classe filha não-abstrata.

```
Obra
uses
  Album, Artista, MetadataEngine.Attributes, SpecialTypes.Nullable,
  SpecialTypes.Proxy;

type
  [Entity]
  [Table('OBRAS')]
  [Sequence('SEQ_OBRAS')]
  [Inheritance(isSingleTable)]
  [DiscriminatorColumn('TIPO', dtString)]
  TObra = class
  private
    FId: Integer;
    FNome: string;
    FEnderecoArquivo: string;
    FDuracao: Nullable<TTime>;

    [Association(ftLazy, orOptional, ctCascadeAll)]
    [JoinColumn('ID_ALBUM', [])]
    FAlbum: Proxy<TAlbum>;

    [Association(ftLazy, orOptional, ctCascadeAll)]
    [JoinColumn('ID_ARTISTA', [])]
    FArtista: Proxy<TArtista>;

    function GetAlbum: TAlbum;
    function GetArtista: TArtista;
    procedure SetAlbum(const Value: TAlbum);
    procedure SetArtista(const Value: TArtista);
  public
    [Id]
    [Column('ID', [cpUnique, cpRequired, cpDontUpdate])]
    property Id: Integer read FId write FId;

    [Column('NOME', [cpRequired], 100)]
    property Nome: string read FNome write FNome;

    [Column('ENDERECO_ARQUIVO', [cpUnique, cpRequired], 300)]
    property EnderecoArquivo: string read FEnderecoArquivo write FEnderecoArquivo;

    [Column('DURACAO', [])]
    property Duracao: Nullable<TTime> read FDuracao write FDuracao;

    property Album: TAlbum read GetAlbum write SetAlbum;
    property Artista: TArtista read GetArtista write SetArtista;
  end;
```

Figura 29 – Classe `TObra` mapeada no *framework*

A Figura 30 mostra a implementação dos métodos de acesso da classe `TObra`.

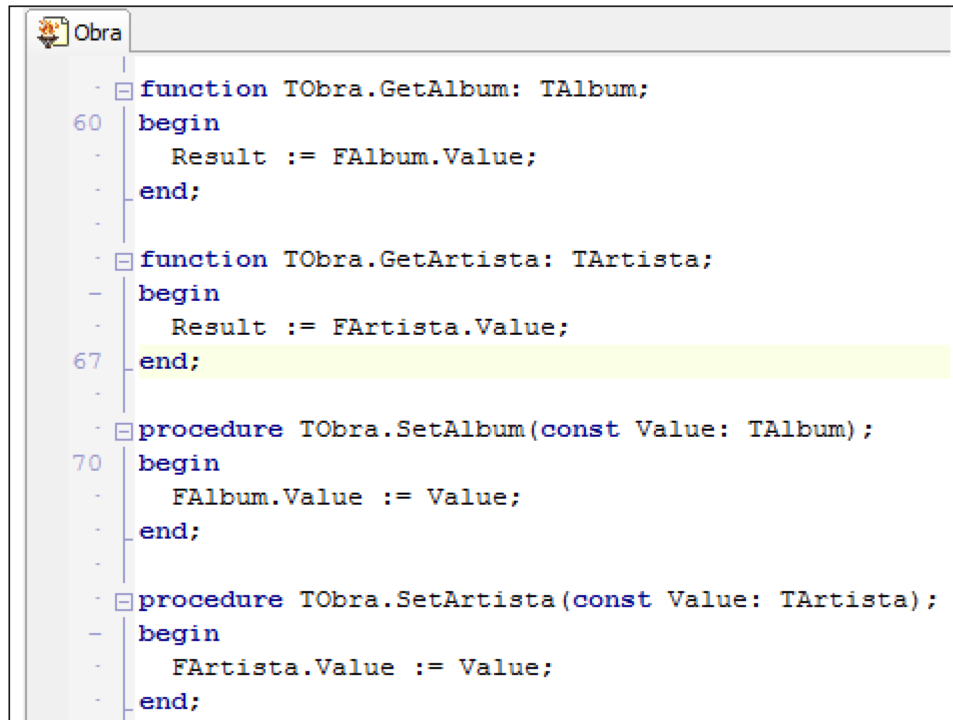


Figura 30 – Implementação dos métodos de acesso da classe TObra

A Figura 31 mostra como ficaria a classe TAlbum. Nota-se a utilização da anotação ManyValuedAssociation acima da *property* Obras que é do tipo TList<TObra>. Como se trata de uma associação bidirecional, pois também existe uma Association de TObra para TAlbum, foi informado o valor 'FAlbum' para o parâmetro MappedBy. Isto indica que o *framework* deve utilizar a Column mapeada no atributo FAlbum da classe TObra como chave estrangeira também para esta associação.

```
type
  [Entity]
  [Table('ALBUNS')]
  [Sequence('SEQ_ALBUNS')]
  TAlbum = class
  private
    FId: Integer;
    FNome: string;
    FAno: Nullable<Integer>;
    FObras: TList<TObra>;
  public
    [Id]
    [Column('ID', [cpUnique, cpRequired, cpDontUpdate])]
    property Id: Integer read FId write FId;

    [Column('NOME', [cpRequired], 100)]
    property Nome: string read FNome write FNome;

    [Column('ANO', [])]
    property Ano: Nullable<Integer> read FAno write FAno;

    [ManyValuedAssociation(ftEager, orOptional, ctCascadeAll, 'FAlbum')]
    property Obras: TList<TObra> read FObras write FObras;

    constructor Create; virtual;
    destructor Destroy; override;
  end;
```

Figura 31 – Classe TAlbum mapeada no *framework*

A Figura 32 mostra como ficariam as classes `TMusica` e `TClipe`. Para cada uma delas, foi dado um nome diferente ao anotar o `DiscriminatorValue`. Este nome informado é o nome que ficará gravado na coluna discriminadora no banco de dados, identificando a classe de cada registro da tabela. Além disso, neste caso as associações foram mapeadas como tipo de carregamento `ftEager`, para que o objeto de formato seja carregado juntamente com a música ou clipe. Não é necessário o atributo `Id` nestas classes porque o mesmo é herdado da classe `TObra`, assim como a anotação de `Sequence`.

As classes `TFormatoMúsica` e `TFormatoVídeo` foram omitidas porque ficariam muito parecidas com a classe `TArtista`, seguindo o mesmo padrão.

```
Musica
type
  [DiscriminatorValue('MUSICA')]
  TMusica = class(TObra)
  private
    FFormato: TFormatoMusica;
  public
    [Association(ftEager, orRequired, ctCascadeAll)]
    [JoinColumn('ID_FORMATO_MUSICA', [cpRequired])]
    property Formato: TFormatoMusica read FFormato write FFormato;
  end;
```

```
Clipe
type
  [DiscriminatorValue('CLIPE')]
  TClipe = class(TObra)
  private
    FFormato: TFormatoVideo;
  public
    [Association(ftEager, orRequired, ctCascadeAll)]
    [JoinColumn('ID_FORMATO_VIDEO', [cpRequired])]
    property Formato: TFormatoVideo read FFormato write FFormato;
  end;
```

Figura 32 – Classes `TMusica` e `TClipe` mapeadas no *framework*

Depois de definidos todos os mapeamentos nas classes de entidades, tem-se o modelo implementado. Pode-se então implementar as visões e os controles que envolveriam as lógicas de negócio da aplicação. As visões seriam formulários Delphi, implementados utilizando os componentes visuais da VCL. O *framework* não possui componentes para criação de telas, visto que seu objetivo é fornecer a camada de persistência.

Para este exemplo, será mostrada apenas uma interface com o usuário, a tela de cadastro de músicas. Esta interface poderia ser desenhada conforme a Figura 33. As demais telas de cadastro da aplicação, como cadastro de vídeos, de artistas, de álbuns, de formatos de música e formatos de vídeos, poderiam seguir o mesmo padrão.

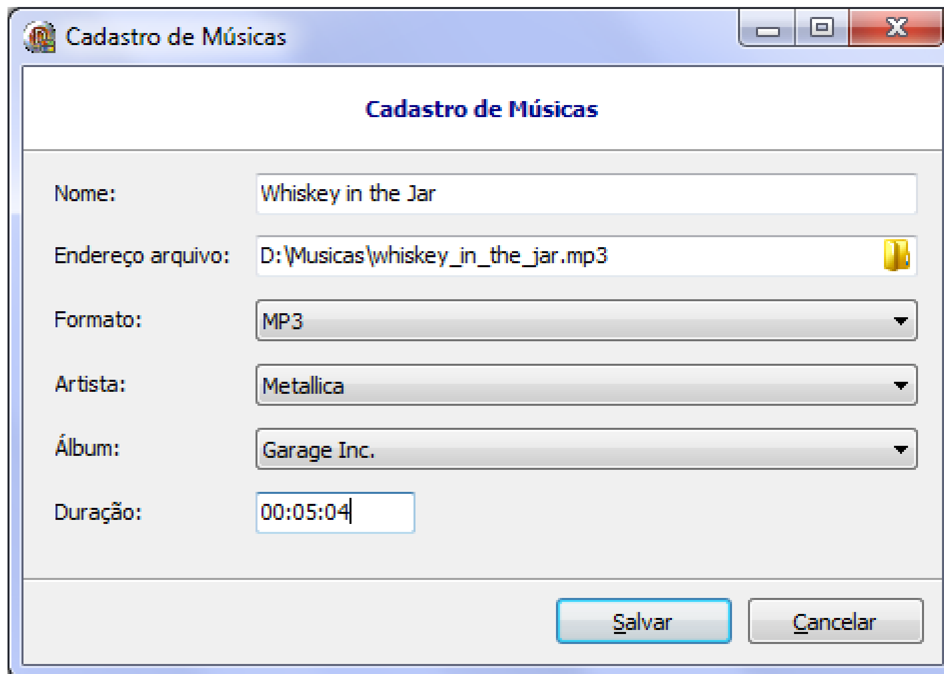


Figura 33 – Tela de cadastro de músicas

Neste formulário existem campos de seleção de formato, artista e álbum. Os valores contidos nestes campos devem ser carregados utilizando o gerenciador de objetos, fazendo-se uma chamada para `ObjectManager.FindAll`. Para o carregamento, pode-se basear no exemplo mostrado na Figura 34.

```
CadMusica
procedure TFrmCadMusica.FormCreate(Sender: TObject);
var
  Formatos: TList<TFormatoMusica>;
  Artistas: TList<TArtista>;
  Albuns: TList<TAlbum>;
  F: TFormatoMusica;
begin
  Formatos := ObjectManager.FindAll<TFormatoMusica>;
  try
    ComboBoxFormato.Clear;
    for F in Formatos do
      ComboBoxFormato.AddItem(F.Nome, F);
  finally
    Formatos.Free;
  end;

  Artistas := ObjectManager.FindAll<TArtista>;
  // Carrega o combo de artistas da mesma forma...

  Albuns := ObjectManager.FindAll<TAlbum>;
  // Carrega o combo de álbuns da mesma forma...
end;
```

Figura 34 – Implementação do carregamento dos componentes da tela

A tela de cadastro de músicas deve utilizar um controlador para realizar a gravação das

músicas. Como sugestão de implementação, o evento `OnClick` do formulário dispararia uma ação do controlador, passando como parâmetro as informações preenchidas na tela, encapsuladas em um objeto `Musica` auto-contido. A implementação ficaria conforme a Figura 35.

```
CadMusica
procedure TFrmCadMusica.BtSalvarClick(Sender: TObject);
var
  Musica: TMusica;
  Controller: TMusicasController;
  Hour, Min, Sec: Word;
begin
  Musica := TMusica.Create;
  Musica.Nome := EdNome.Text;
  Musica.EnderecoArquivo := EdEndArquivo.Text;

  if CbAlbum.ItemIndex >= 0 then
    Musica.Album := TAlbum(CbAlbum.Items.Objects[CbAlbum.ItemIndex]);

  if CbArtista.ItemIndex >= 0 then
    Musica.Artista := TArtista(CbArtista.Items.Objects[CbArtista.ItemIndex]);

  Hour := StrToInt(Copy(EdDuracao.Text, 1, 2));
  Min := StrToInt(Copy(EdDuracao.Text, 3, 2));
  Sec := StrToInt(Copy(EdDuracao.Text, 5, 2));
  Musica.Duracao := EncodeTime(Hour, Min, Sec, 0);

  Controller := TMusicasController.Create;
  try
    Controller.SalvarMusica(Musica);
  finally
    Controller.Free;
  end;
end;
```

Figura 35 – Implementação do botão de salvar música

A Figura 36 mostra como ficaria a implementação do método `SalvarMusica` do controlador. O mesmo faria uso do gerenciador de objetos do *framework* para salvar a música no banco de dados através do método `Persist`. Neste exemplo, o gerenciador de objetos é criado e destruído dentro do escopo de um método, mas em transações mais complexas o seu ciclo de vida pode e deve ser maior.

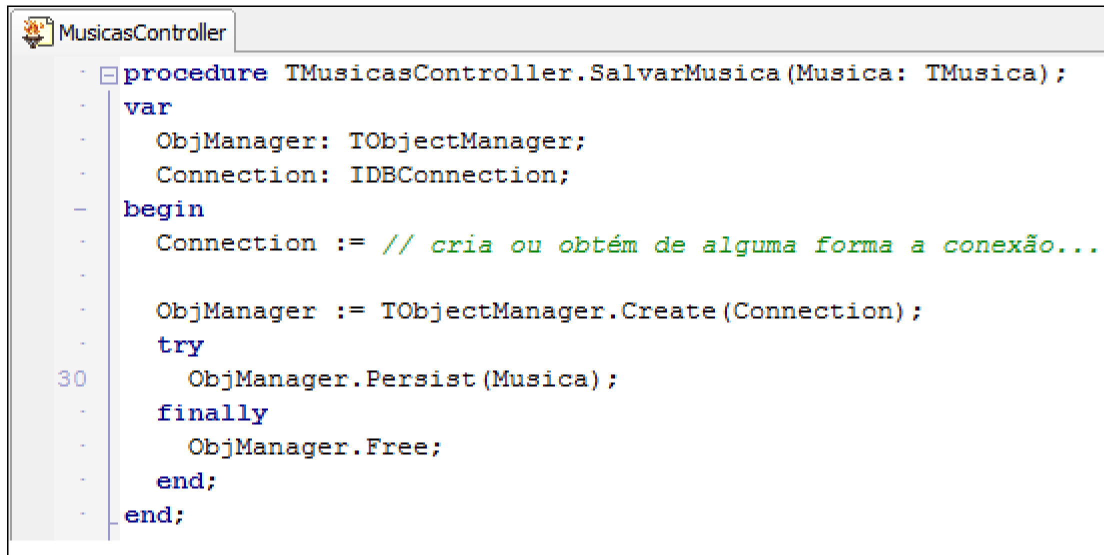


Figura 36 – Implementação do método para salvar música no controlador

O *framework* não possui um gerenciamento completo de conexões, mas permite que a aplicação as gerencie como desejar, bastando que passe a conexão como parâmetro na instanciação do gerenciador de objetos. Em aplicações que possuem apenas uma conexão com um banco de dados, sugere-se a criação de uma classe única de acesso à conexão, seguindo o padrão de projeto Singleton.

A classe única de acesso à conexão poderia ser responsável pela instanciação e configuração do objeto de conexão. Para a configuração do objeto de conexão, que implementa a interface `IDBConnection`, pode-se basear no exemplo da Figura 37. Nela, carrega-se a conexão a partir dos parâmetros de conexão e conecta-se ao banco de dados. A conexão mostrada na figura é feita com o SGBD Firebird.

Como se pode ver no exemplo, o *framework* também não gerencia parâmetros de conexão. Os mesmos devem ser gerenciados pela própria aplicação. Logicamente, em um sistema de produção, os parâmetros de conexão como usuário, senha e caminho da base de dados deveriam ser lidos de algum meio externo como arquivo criptografado ou registro do Windows, por exemplo.

```
ConexaoUnica
procedure TConexaoUnica.CarregaConexao;
var
  Conn: TDBExpressConnectionAdapter;
begin
  Conn := TDBExpressConnectionAdapter.Create(Application);

  Conn.DriverName := 'Firebird';
  Conn.ConnectionName := 'IBConnection';
  Conn.LibraryName := 'dbxfb.dll';
  Conn.GetDriverFunc := 'getSQLDriverINTERBASE';
  Conn.VendorLib := 'fbclient.dll';

  Conn.Params.Values['DriverName'] := Conn.DriverName;
  Conn.Params.Values['Database'] := 'localhost:C:\Caminho\Da\Base\base.fdb';
  Conn.Params.Values['User_Name'] := 'sysdba';
  Conn.Params.Values['Password'] := 'masterkey';
  Conn.Params.Values['SQLDialect'] := '3';
  Conn.Params.Values['Interbase TransIsolation'] := 'ReadCommited';
  Conn.Params.Values['Trim Char'] := 'True';

  Conn.LoginPrompt := False;
  Conn.SQLHourGlass := True;

  FConnection := Conn;
  FConnection.Connect;
end;
```

Figura 37 – Configuração do objeto de conexão

A aplicação também deverá possuir telas de consulta das informações cadastradas. Nestas telas, deverá ser possível alterar ou excluir os objetos cadastrados. Não são mostradas estas telas, mas a Figura 38 mostra exemplos de implementação para alterar e excluir objetos. Note-se que como o método `Find` é genérico, não há a necessidade de conversão de tipos de objetos. O gerenciador de objetos já devolve o objeto no tipo correto.

CadMusica

```
var
  Musica: TMusica;
begin
  // Para alterar uma música, sabendo-se que seu código é 1000
  // Opção 1: Flush
  // Neste caso os valores dos campos serão mantidos, muda-se apenas o Nome.
  Musica := ObjectManager.Find<TMusica>(1000);
  Musica.Nome := 'Novo nome da música';
  ObjectManager.Flush;

  // Para alterar uma música, sabendo-se que seu código é 1000
  // Opção 2: Merge
  // Neste caso todos os campos além do Id e Nome ficarão vazios.
  Musica := TMusica.Create;
  Musica.Id := 1000;
  Musica.Nome := 'Novo nome da música';
  ObjectManager.Merge(Musica);

  // Para excluir uma música, sabendo-se que seu código é 1000
  Musica := ObjectManager.Find<TMusica>(1000);
  ObjectManager.Remove(Musica);
end;
```

Figura 38 – Exemplos de alteração e exclusão de objetos

Com a aplicação pronta para ser compilada e executada, ainda falta a construção da base de dados, ou seja, a criação das tabelas. Esta tarefa pode ser feita utilizando o gerenciador de base de dados, disponibilizado pelo *framework* através da classe `TDatabaseManager`. A implementação pode ser feita conforme a Figura 39.

O método `BuildDatabase` realiza a construção de todas as tabelas, chaves primárias, chaves únicas, chaves estrangeiras e *sequences* no banco de dados. Esta operação geralmente é lenta e deve ser realizada apenas uma vez. Em uma aplicação local, pequena e monousuário, poderia ser executada na primeira vez que é aberta a aplicação. Já em um sistema cliente-servidor, seria interessante compilar esta parte para um aplicativo de configuração em separado. Cabe ao desenvolvedor decidir o local onde esta implementação deve ser colocada de acordo com o caso.

```
procedure TFrmPrincipal.BtCriarBancoClick(Sender: TObject);
var
  Connection: IDBConnection;
  Manager: TDatabaseManager;
begin
  Connection := // Cria ou obtém a conexão de alguma forma...

  Connection.Connect;
  try
    Manager := TDatabaseManager.Create(Connection);
    try
      Manager.CreateDatabase;
    finally
      Manager.Free;
    end;
  finally
    Connection.Disconnect;
  end;
end;
```

Figura 39 – Construção da base de dados

3.5 RESULTADOS E DISCUSSÃO

Nesta seção são apresentados os resultados obtidos da geração de SQL e analisado o desempenho da persistência de objetos em alguns testes realizados. Também é feita uma comparação do *framework* com trabalhos correlatos.

3.5.1 Análise do SQL gerado

A seguir são mostrados alguns testes feitos com o *framework* e o código SQL que o mesmo gerou para cada teste. Ao final, são feitos comentários sobre estes resultados. Estes testes foram realizados utilizando o SGBD Firebird, gerando SQL para este banco. A geração de SQL para o MySQL teria apenas uma diferença: ao invés de *sequences*, ter-se-ia colunas do tipo *auto-increment*.

Para todos os testes mostrados a seguir, consideram-se as classes de entidades mostradas no Quadro 25, as quais sumarizam em um único código todos os recursos

discutidos até o momento. Nota-se a auto-associação da classe `TCliente` através do atributo `FClienteCobranca`.

```
[Entity]
[Table('CLIENTES')]
[Sequence('SEQ_CLIENTES', 1, 1)]
TCliente = class
private
  [Id]
  [Column('CODIGO', [cpUnique, cpRequired])]
  FCodigo: LongInt;
  [Column('NOME', [cpRequired])]
  FNome: string;
  [Column('IDADE')]
  FIdade: Nullable<Integer>;
  [Column('NASCIMENTO')]
  FNascimento: Nullable<TDate>;
  [Column('ATIVO')]
  FIsAtivo: Nullable<Boolean>;
  [Association(ftEager, orOptional, ctCascadeAll)]
  [JoinColumn('CLIENTE_COBRANCA')]
  FClienteCobranca: Proxy<TCliente>;
public
  // Getters e setters omitidos
end;

[Entity]
[Table('NOTAS_FISCAIS')]
[Sequence('SEQ_NOTAS_FISCAIS')]
TNotaFiscal = class
private
  [Id]
  [Column('CODIGO', [cpUnique, cpRequired])]
  FCodigo: LongInt;
  [Column('NUMERO', [cpUnique, cpRequired])]
  FNumero: Integer;
  [Association(ftEager, orOptional, ctCascadeAll)]
  [JoinColumn('CODIGO_CLIENTE')]
  FCliente: Proxy<TCliente>;
  function GetCliente: TCliente;
  procedure SetCliente(const Value: TCliente);
public
  // Getters e setters omitidos
end;
```

Quadro 25 – Implementação das entidades utilizadas nos testes

3.5.1.1 Teste 1 – Construção da base de dados

O primeiro teste foi feito para analisar a construção da base de dados a partir destas duas classes de entidades. O Quadro 26 mostra fielmente, incluindo as quebras de linha, o código SQL gerado a partir da chamada do método `BuildDatabase` do `TDatabaseManager`.

```
CREATE TABLE CLIENTES (
  CODIGO INTEGER NOT NULL,
  NOME VARCHAR(255) NOT NULL,
  IDADE INTEGER,
  NASCIMENTO DATE,
  ATIVO CHAR(1),
  CLIENTE_COBRANCA INTEGER,
  CONSTRAINT PK_CLIENTES PRIMARY KEY (CODIGO));

CREATE TABLE NOTAS_FISCAIS (
  CODIGO INTEGER NOT NULL,
  NUMERO INTEGER NOT NULL,
  CODIGO_CLIENTE INTEGER,
  CONSTRAINT PK_NOTAS_FISCAIS PRIMARY KEY (CODIGO),
  CONSTRAINT UK_NOTAS_FISCAIS UNIQUE (NUMERO));

ALTER TABLE CLIENTES
  ADD CONSTRAINT FK_CLIENTES_CLIENTES_CLIENTE_CO
  FOREIGN KEY (CLIENTE_COBRANCA)
  REFERENCES CLIENTES(CODIGO);

ALTER TABLE NOTAS_FISCAIS
  ADD CONSTRAINT FK_NOTAS_FISCAIS_CLIENTES_CODIG
  FOREIGN KEY (CODIGO_CLIENTE)
  REFERENCES CLIENTES(CODIGO);

CREATE GENERATOR SEQ_CLIENTES;

CREATE GENERATOR SEQ_NOTAS_FISCAIS;
```

Quadro 26 – SQL de construção da base de dados

Foram executados no total seis comandos SQL. Em primeiro lugar, o *framework* cria todas as tabelas correspondentes às entidades, sem as chaves estrangeiras. Depois são alteradas as tabelas, adicionando as chaves estrangeiras. Por fim, criadas as *sequences*. No SGBD Firebird, as *sequences* são chamadas de *generators*. As chaves estrangeiras não são criadas diretamente no comando `create table` porque em um caso de referência mútua entre duas tabelas, o *framework* não saberia determinar qual tabela teria que ser criada por primeiro.

Nota-se que o código SQL gerado é eficiente e possui uma boa legibilidade. Isto provê facilidade em casos onde é necessário monitorar ou analisar o SQL que está sendo gerado pelo *framework*.

3.5.1.2 Teste 2 – Persistência de objetos

O segundo teste foi feito para analisar o código SQL gerado para persistência de objetos. O Quadro 27 mostra o código fonte que foi implementado na aplicação para a realização deste teste.

```
NotaFiscal := TNotaFiscal.Create;
NotaFiscal.Numero := 100;

NotaFiscal.Cliente := TCliente.Create;
NotaFiscal.Cliente.Nome := 'Nome do cliente';
NotaFiscal.Cliente.Nascimento := EncodeDate(2010, 10, 23);
NotaFiscal.Cliente.IsAtivo := False;

Manager.Persist(NotaFiscal);
```

Quadro 27 – Implementação do teste de persistência de objetos

Os comandos SQL gerados pelo *framework* para este teste, juntamente com seus parâmetros, podem ser vistos no Quadro 28.

```
SELECT GEN_ID(SEQ_CLIENTES, 1)
FROM RDB$DATABASE;
```

```
INSERT INTO CLIENTES (
  CODIGO, NOME, IDADE, NASCIMENTO, ATIVO, CLIENTE_COBRANCA)
VALUES (
  :A_CODIGO, :A_NOME, :A_IDADE, :A_NASCIMENTO, :A_ATIVO,
:A_CLIENTE_COBRANCA);

A_CODIGO = "1" (ftInteger)
A_NOME = "Nome do cliente" (ftString)
A_IDADE = NULL (ftInteger)
A_NASCIMENTO = "23/10/2010" (ftDate)
A_ATIVO = "False" (ftBoolean)
A_CLIENTE_COBRANCA = NULL (ftInteger)
```

```
SELECT GEN_ID(SEQ_NOTAS_FISCAIS, 1)
FROM RDB$DATABASE;
```

```
INSERT INTO NOTAS_FISCAIS (
  CODIGO, NUMERO, CODIGO_CLIENTE)
VALUES (
  :A_CODIGO, :A_NUMERO, :A_CODIGO_CLIENTE);

A_CODIGO = "1" (ftInteger)
A_NUMERO = "100" (ftInteger)
A_CODIGO_CLIENTE = "1" (ftInteger)
```

Quadro 28 – Código SQL gerado para persistência de objetos

Verifica-se que o objeto que foi passado como parâmetro para persistência foi o objeto `NotaFiscal`, porém o primeiro comando SQL gerado foi para persistência do objeto `Cliente`. O *framework* tomou esta ação porque detectou que o objeto a ser persistido, `NotaFiscal`, estava associado com outro objeto que ainda não estava persistido. Então este segundo objeto precisa ser persistido primeiro.

Comparando-se o Quadro 27 com o Quadro 28, nota-se o quanto menos o *framework* permite ao programador escrever para produzir um código de manutenção de notas e clientes.

3.5.1.3 Teste 3 – Obtenção de objetos

O terceiro teste foi feito para analisar o código SQL gerado para obtenção de objetos. O Quadro 29 mostra o código fonte que foi implementado na aplicação para a realização deste teste.

```
NotaFiscal := Manager.Find<TNotaFiscal>(1000);
```

Quadro 29 – Implementação do teste de obtenção de objetos

Os comandos SQL gerados pelo *framework* para este teste, juntamente com seus parâmetros, podem ser vistos no Quadro 30.

```
SELECT A.CODIGO AS A_CODIGO, A.NUMERO AS A_NUMERO, B.CODIGO AS B_CODIGO,
B.NOME AS B_NOME, B.IDADE AS B_IDADE, B.NASCIMENTO AS B_NASCIMENTO,
B.ATIVO AS B_ATIVO, B.CLIENTE_COBRANCA AS B_CLIENTE_COBRANCA
FROM NOTAS_FISCAIS AS A
  LEFT JOIN CLIENTES AS B ON (B.CODIGO = A.CODIGO_CLIENTE)
WHERE A.CODIGO = :WHERE_A_CODIGO;

WHERE_A_CODIGO = "1000" (ftInteger)
```

Quadro 30 – Código SQL gerado para obtenção de objetos

Verifica-se que os objetos associados ao objeto `NotaFiscal` que possuem associação com carregamento `ftEager` são carregados juntamente com a nota fiscal em apenas uma requisição ao banco de dados. Isto é feito através de junções do tipo `inner join` ou `left join`. Além disso, verifica-se também que o *framework* adiciona *aliases* em todas as tabelas da cláusula `from` e campos da cláusula `select`. Isto é feito para garantir a unicidade dos campos no resultado do comando SQL, já que podem haver campos com mesmo nome em tabelas diferentes, ou então a mesma tabela pode ser utilizada mais de uma vez no mesmo comando SQL. Por exemplo, se houver uma chave estrangeira de uma tabela para a própria tabela, poderá ser feito um *join* de uma tabela para ela mesma. Isto não gerará problemas desde que elas recebam diferentes *aliases*.

Todas estas técnicas aplicadas à geração do código SQL de obtenção de objetos fizeram diminuir um pouco a sua legibilidade, porém foram necessárias para garantir o seu funcionamento desde consultas mais simples até consultas mais complexas.

3.5.2 Análise de desempenho

Foram realizados testes de desempenho com o *framework*, com finalidade de comparação entre diferentes formas de obtenção e persistência de objetos, bem como análise de desempenho com cargas maiores. Na obtenção de objetos, foram medidos os tempos com carregamento tardio e carregamento ansioso. Os testes foram realizados com o SGBD Firebird e as entidades que foram utilizadas são as classes `TCliente` e `TNotaFiscal` mostradas no Quadro 25. Na persistência de objetos, todos os objetos de um mesmo teste foram persistidos dentro de um mesmo ciclo de vida do gerenciador de objetos. Sabendo-se que o gerenciador de objetos armazena cada objeto persistido no mapa de identidade, estes objetos foram acumulando durante cada teste, tendo no pior caso um mapa de identidade com 20.000 objetos mapeados. Na obtenção de objetos, cada requisição foi feita para obter um objeto diferente, a fim de não utilizar o *cache* interno do *framework*. A Tabela 1 mostra os tempos medidos, em segundos.

Tabela 1 – Medição de tempos de persistência e obtenção de objetos

	Teste 1	**Teste 2**	**Teste 3**	**Média**
Persist de objetos avulsos				
100 x 1 Cliente	0,40	0,56	0,92	0,63
1.000 x 1 Cliente	6,69	6,55	6,81	6,68
10.000 x 1 Cliente	49,92	48,06	51,30	49,76
Persist de objetos associados				
100 x (1 NotaFiscal + 1 Cliente)	1,12	1,10	0,89	1,04
1.000 x (1 NotaFiscal + 1 Cliente)	13,53	9,29	12,33	11,72
10.000 x (1 NotaFiscal + 1 Cliente)	98,55	97,53	100,73	98,94
Find				
100 x 1 Cliente	0,43	0,45	0,48	0,45
1.000 x 1 Cliente	4,81	4,44	4,35	4,53
10.000 x 1 Cliente	40,58	40,96	41,34	40,96
Find - Eager				
100 x (1 NotaFiscal + 1 Cliente)	0,44	0,48	0,46	0,46
1.000 x (1 NotaFiscal + 1 Cliente)	6,06	4,60	4,37	5,01
10.000 x (1 NotaFiscal + 1 Cliente)	35,52	42,09	38,74	38,78
Find – Lazy				
100 x (1 NotaFiscal + 1 Cliente)	0,84	1,16	0,56	0,85
1.000 x (1 NotaFiscal + 1 Cliente)	7,09	8,69	8,92	8,23
10.000 x (1 NotaFiscal + 1 Cliente)	94,96	61,02	76,81	77,60

Como se pode constatar, em geral o tempo de persistência dos objetos cresce linearmente. Ou seja, quanto maior o número de objetos a persistir, maior o tempo de execução, proporcionalmente. O mesmo acontece com o tempo de obtenção dos objetos.

Verificou-se também que o tempo de persistência de dois objetos associados é em

geral duas vezes o tempo de persistência de um objeto avulso. Isso indica que a associação entre os objetos não aumenta o tempo de persistência no *framework*. Ou seja, associações não geram uma sobrecarga perceptível.

Foram feitos testes comparativos entre estratégias de carregamento `ftEager` e `ftLazy`. Como se pode constatar, o carregamento de dois objetos com estratégia `ftEager` demora em média apenas 2,2% a mais do que o carregamento de apenas um objeto. Já o carregamento dos mesmos objetos com estratégia `ftLazy` demora 88,9% a mais do que o carregamento de apenas um objeto, pois é necessário mais um acesso ao banco de dados. Com isto, verifica-se a importância de recuperar objetos interligados em apenas um acesso ao banco sempre que possível, desde que os objetos realmente sejam utilizados. Por outro lado, não é interessante mapear todas as associações como `ftEager`, pois desta forma um único acesso ao banco carregaria uma quantidade grande demais de objetos, e isto pode deixar o SGBD sobrecarregado, aparentando estar travado inclusive para outros usuários. As associações entre as entidades devem ser analisadas para determinar qual estratégia de carregamento é mais apropriada para cada caso.

3.5.3 Comparações com trabalhos correlatos

Para a comparação entre este trabalho e os *frameworks* correlatos, foi eleito um conjunto de características e funcionalidades a serem examinadas. Para tal, foram consultados os tópicos de ajuda de cada *framework* correlato, bem como suas documentações. Também foram analisadas partes dos seus códigos fontes. Os dados obtidos foram reunidos no Quadro 31. Embora os trabalhos correlatos possam ter outras funcionalidades importantes além de persistência e mapeamento objeto-relacional, são consideradas aqui apenas as características relevantes nesse contexto. A ferramenta de aplicação do padrão DAO foi analisada à parte por não ser um *framework* de persistência.

	Características	**Este trabalho**	**Hibernate**	**Instant Objects**
1	Linguagem	Delphi Win32	Java	Delphi Win32
2	Versões suportadas	2010	1.4 a 1.6	5 a 2006
3	Ano da última versão oficial	2010	2010	2006
4	Suporta mapeamento de herança (quantas estratégias?)	Sim (3)	Sim (3)	Não
5	Suporta mapeamento de associações	Sim	Sim	Sim
6	Mapeamento com anotações	Sim	Sim	Não
7	Mapeamento com comentários	Não	Não	Sim
8	Mapeamento com XML	Não	Sim	Sim
9	Permite consultas polimórficas	Sim	Sim	Não
10	Liberdade de herança na entidade raiz	Sim	Sim	Não
11	Implementa Carga Tardia	Sim	Sim	Sim
12	Implementa Carga Ansiosa com Joins	Sim	Sim	Sim
13	Suporta tipos anuláveis	Sim	Sim	Sim
14	Orientado a Datasets (Ex: `TTable`, `TQuery`)	Não	Não	Sim
15	Integração nativa c/ controles de tela	Não	Não	Sim
16	Possui editor gráfico de classes nativo	Não	Não	Sim
17	Possui linguagem específica de consulta	Não	Sim	Sim
18	Permite utilizar tipos comuns da linguagem	Sim	Sim	Não
19	Constrói a base de dados	Sim	Sim	Sim

Quadro 31 - Comparação entre características e funcionalidades dos *frameworks*

Tanto o Hibernate, segundo Bauer e King (2007, p. 192), quanto o *framework* desenvolvido neste trabalho implementam técnicas de Mapeamento Objeto-Relacional, como mapeamento de herança e mapeamento de associações, a fim de que a aplicação consiga utilizar com maior flexibilidade possível os conceitos do paradigma orientado a objetos. Para cada uma destas técnicas ainda é possível escolher entre um conjunto de estratégias. Porém, ambos os *frameworks* não possuem integração nativa com componentes visuais para apresentação dos dados na interface gráfica.

Instant Objects (INSTANT OBJECTS, 2006) é declarado pelos seus autores como um *framework* de persistência de objetos, não sendo citado o conceito de mapeamento objeto-relacional em sua definição, embora algumas técnicas de mapeamento sejam utilizadas. Algumas de suas construções são orientadas a *datasets*, de forma a se encaixarem mais facilmente em alguns componentes da VCL de apresentação dos dados na tela, como `DBEdit`, `DBGrid`, entre outros. Por outro lado, ele carece de algumas características de um *framework* puramente orientado a objetos como, por exemplo, consultas polimórficas. Um ponto positivo é que Instant Objects suporta desde versões antigas como Delphi 5 até as mais atuais. Na versão que está sendo desenvolvida atualmente, o *framework* deverá suportar Delphi 2010.

Hibernate e Instant Objects possuem linguagens de consulta específicas. Hibernate disponibiliza a Hibernate *Query Language* (HQL) enquanto Instant Objects disponibiliza a *Instant Query Language* (IQL). As duas linguagens são inspiradas na linguagem SQL, onde

as tabelas são identificadas pelos nomes das classes. O *framework* desenvolvido neste trabalho não disponibiliza uma linguagem específica para consultas. Esta funcionalidade é sugerida como extensão.

O *framework* desenvolvido neste trabalho e o Hibernate são *frameworks* pouco intrusivos, ou seja, evitam ao máximo alterar ou descaracterizar o código fonte da aplicação para atingir seus objetivos. As classes de entidades não precisam descender de uma classe base do *framework*, o que possibilita que a entidade raiz de uma hierarquia tenha liberdade para descender de outra classe quando necessário. Além disso, os atributos podem usar tipos da própria linguagem ao invés de tipos do *framework*. Desta forma, o código fonte da aplicação torna-se mais desacoplado do *framework*.

No *framework* Instant Objects, as classes de entidades precisam descender de uma classe base chamada `TInstantObject`, e os atributos precisam ser de tipos especiais como `TInstantString`, `TInstantDateTime` e `TInstantReference`, entre outros. Para minimizar o impacto, o *framework* sugere que sejam criadas *properties* de tipos primitivos da linguagem e cujos *getters* e *setters* servem como adaptadores para os tipos do *framework*.

No *framework* desenvolvido neste trabalho, algumas construções precisam utilizar tipos especiais, como `Nullable` e `Proxy`. Porém, o uso de tipos genéricos nestas construções ajuda a minimizar a intrusão já que, quando bem aplicados, afetam somente as classes de entidades. O resto do código fonte da aplicação pode utilizar os atributos das entidades de forma transparente, sem preocupar-se com estas estruturas especiais, pois estas estruturas transpõem o tipo primitivo genérico utilizado na declaração dos atributos. No Hibernate, o problema dos tipos anuláveis é resolvido com utilização de objetos de valor ao invés de tipos primitivos. Usa-se a classe `Integer` ao invés do tipo `int`, por exemplo. A estrutura Proxy também não é necessária no Hibernate porque o padrão Lazy Load foi implementado utilizado orientação a aspectos, dispensando esta estrutura.

Nota-se que as anotações feitas com Atributos Customizados no *framework* desenvolvido neste trabalho são validadas em tempo de compilação pelo Delphi. Isto é uma vantagem sobre mapeamentos baseados em XML ou comentários, como é o caso do Instant Objects, pois tais mapeamentos são validados somente em tempo de execução pelo próprio *framework*.

A ferramenta de aplicação do padrão DAO de Sardagna (2007, p. 14) não é um *framework* e sim de uma ferramenta de análise e geração de código. Esta ferramenta adota um conceito comum que é organizar uma camada de persistência para aplicações orientadas a objetos construídas em linguagem Delphi. Tal ferramenta também se baseia no uso do padrão

de projeto DAO, e foi implementada em Delphi 2006.

A ferramenta de aplicação do padrão DAO, segundo Sardagna (2007, p. 64), foi implementada para suportar o SGBD Interbase, enquanto o *framework* do trabalho atual suporta Firebird e MySQL. Tal ferramenta analisa códigos fonte de projetos com componente de acesso DBExpress. O *framework* do presente trabalho também utiliza DBExpress como componente padrão de acesso, porém foi criada uma camada de independência para que facilmente possam ser implementados adaptadores para outros componentes de acesso.

4 CONCLUSÕES

Os resultados obtidos com o desenvolvimento do *framework* foram bons. Foi possível cumprir todos os requisitos propostos. Algumas técnicas como mapeamento de herança, associações, tipos anuláveis e carregamento tardio também foram implementadas com sucesso, sem que tivessem sido definidos como requisitos funcionais.

O objetivo principal do trabalho foi atendido, que era desenvolver um OPF em Delphi que utilize a técnica de ORM para fornecer a camada de persistência a uma aplicação. Acredita-se que esta camada de persistência seja eficiente e transparente. Os objetivos específicos do trabalho também foram atendidos com sucesso.

As ferramentas utilizadas foram adequadas para o desenvolvimento do trabalho. Os padrões de projeto utilizados também foram adequados e acredita-se que tornaram o código fonte do *framework* organizado e fácil de entender. Obteve-se produtividade com a exploração de recursos novos como tipos genéricos, a nova API do RTTI e Atributos Customizados, disponíveis na linguagem Delphi para Windows há menos de dois anos conforme Groves (2009). Tais recursos possibilitaram a construção de estruturas como Nullable e Proxy.

Vários tipos de aplicações podem utilizar este *framework* como camada de persistência, desde aplicações pequenas com banco de dados embarcado até sistemas maiores que rodam em arquitetura cliente-servidor. Porém, para ser considerado prático e eficiente na construção de grandes aplicações, são necessárias certas adaptações e extensões. O *framework* pode ser utilizado também para o desenvolvimento de *middlewares* para suportar arquiteturas mais elaboradas do que a cliente-servidor. Uma limitação é que a única plataforma suportada atualmente é Windows.

A proposta inicial deste trabalho previa resolver problemas de concorrência com outras instâncias da aplicação que interagem com o mesmo banco de dados. Porém, devido à grande complexidade dos controles de transações para tratar a concorrência, optou-se por não implementar este suporte neste trabalho. Em aplicações monousuário este problema simplesmente não existe. Em aplicações cliente-servidor de pequeno porte pode-se executar a interação com o gerenciador de objetos vinculada com uma transação do SGBD. Ou seja, inicia-se uma transação no banco, depois se realiza toda a lógica que envolve a persistência de uma só vez, depois se finaliza a transação. Desta forma, os objetos lidos ficarão bloqueados no banco enquanto a transação não for finalizada, impedindo modificações concorrentes. Esta

técnica, porém, pode deixar o sistema travado se houverem muitas transações concorrentes, por isto não é indicada para sistemas de grande porte. Além disso, é possível resolver este problema como extensão, criando uma camada intermediária para tratar a concorrência.

Foi encontrada uma dificuldade ao implementar o padrão de projeto Lazy Load. Estimava-se utilizar técnicas de orientação a aspectos para aplicar este padrão, mas o Delphi não suporta atualmente os mecanismos necessários para tal. Elaborou-se então uma solução alternativa que se mostrou eficiente: a estrutura `Proxy<T>`. O resultado esperado foi alcançado com sucesso.

Muitas funcionalidades do *framework* foram preparadas para serem estendidas, fazendo com que o projeto possa ser continuado, agregando mais funcionalidades com facilidade. Acredita-se que este trabalho contribua bastante para o desenvolvimento tecnológico, inovando em termos de persistência em Delphi para Windows. Algumas inovações são a utilização de anotações para mapeamento e a disponibilização de uma interface de persistência fortemente orientada a objetos. Com isto, abre-se um leque de possibilidades para futuros trabalhos.

4.1 EXTENSÕES

Como sugestões para extensões a este trabalho sugerem-se:

a) suporte a construção de consultas customizáveis pelo usuário, com recursos como filtros, junções, ordenação e agrupamentos. Para tal, poderia ser desenvolvida uma API orientada a objetos ou então uma linguagem específica de consulta, semelhante à HQL ou IQL;
b) suporte a associações muitos-para-muitos com ou sem classe associativa. Na primeira, o *framework* deve gerar uma tabela intermediária no banco de dados sem a necessidade de uma classe intermediária na aplicação;
c) suporte a identificadores compostos por mais de um campo. A finalidade é tornar o *framework* mais facilmente compatível com bases de dados de sistemas legados;
d) suporte a outros SGBDs de mercado, como Oracle, Microsoft SQL Server, PostgreSQL e IBM DB2, entre outros;
e) centralização das configurações de conexão como usuário, senha, porta e caminho da base de dados no próprio *framework*, para que o *framework* possa gerenciar

transparentemente o objeto de conexão;

f) controle avançado de transações e sessões no gerenciador de objetos;

g) suporte a mapeamento de campos de tipos especiais como BLOB, *text*, enumerações e *arrays*;

h) suporte ao mapeamento automático de certos atributos como `Column`, para que seja possível persistir uma classe sem mapear os atributos, visando a metologia Rapid Application Development (RAD) que faz parte da filosofia do Delphi;

i) suporte a atualização da estrutura de uma base de dados já criada pelo *framework* para uma nova versão, detectando as alterações feitas nos mapeamentos das entidades e refletindo as mudanças no banco de dados através de comandos DDL;

j) caso as próximas versões da linguagem Delphi venham a suportar orientação a aspectos, a implementação do padrão *Lazy Load* nas associações com estratégia de carregamento tardio pode ser aprimorada, eliminando a necessidade da estrutura Proxy;

k) integração com outros projetos como *frameworks* para geração de telas. A ideia é disponibilizar uma solução mais completa para construção de diversas camadas da aplicação.

REFERÊNCIAS BIBLIOGRÁFICAS

BAUER, Christian; KING, Gavin. **Java persistence with Hibernate**. Greenwich: Manning Publications, 2007.

BERNARDI, Diogo A. Técnicas de mapeamento objeto relacional. **SQL Magazine**, Rio de Janeiro, v. 6, n. 40, p. 46-51, 2006.

EMBARCADERO. **The new memory manager in BDS 2006**. [S.l.], 2006. Disponível em: <http://edn.embarcadero.com/article/33416#1Introduction>. Acesso em: 27 out. 2010.

EMBARCADERO. **VCL overview**. [S.l.], 2009. Disponível em: <http://docwiki.embarcadero.com/RADStudio/en/VCL_Overview>. Acesso em: 6 out. 2010.

EMBARCADERO. **Attributes and RTTI**. [S.l.], 2010. Disponível em: <http://docwiki.embarcadero.com/RADStudio/en/Overview_of_Attributes>. Acesso em: 1 out. 2010.

FOWLER, Martin. **Padrões de arquitetura de aplicações corporativas**. Tradução Mareci P. de Oliveira. Porto Alegre: Bookman, 2006.

GAJIC, Zarko. **A tour of Turbo Delphi IDE**. [S.l.], [2006?]. Disponível em: <http://delphi.about.com/od/turbodelphitutorial/ss/delphi_ide_tour_6.htm>. Acesso em: 2 mar. 2010.

GAMMA, Erich et al. **Padrões de projeto**: soluções reutilizáveis de software orientado a objetos. Tradução Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

GROVES, Malcolm. **RTTI and attributes in Delphi 2010**. [S.l.], 2009. Disponível em: <http://www.malcolmgroves.com/blog/?p=476>. Acesso em: 26 mar. 2010.

HEUSER, Carlos A. **Projeto de banco de dados.** 3. ed. Porto Alegre: Sagra Luzzatto, 2000.

INSTANT OBJECTS. **Instant Objects:** object persistence framework. [S.l.], 2006. Disponível em: <http://www.instantobjects.org/>. Acesso em: 26 mar. 2010.

LEME, Ricardo R. **Qual a diferença entre comandos DCL, DML e DDL**. [S.l.], 2008. Disponível em: <http://ricardoleme.blogspot.com/2008/06/qual-diferena-entre-comandos-dcl-dml-e.html>. Acesso em: 12 dez. 2010.

MARTINS, Ivan J. RTTI: tópicos avançados. **Clube Delphi**, Rio de Janeiro, v. 1, n. 42, p. 6-11, [2003?].

MENEZES, Eduardo D. B. de. **Princípios de análise e projeto de sistemas com UML**. 2. ed. totalmente rev. e atual. Rio de Janeiro: Campus, Elsevier, 2007. 369 p.

MOURÃO, Rodrigo C. RTTI: conheça as novidades na RTTI do Delphi 2010. **Clube Delphi**, Rio de Janeiro, v. 6, n. 111, p. 104-121, 2009.

MSDN. **Nullable generic structure**. [S.l.], 2006. Disponível em: <http://msdn.microsoft.com/en-us/library/b3h38hb0(v=VS.80).aspx>. Acesso em: 4 out. 2010.

QUICOLI, Paulo R. MVC: Model-View-Controller. **Clube Delphi**, Rio de Janeiro, v. 4, n. 90, p. 26-34, 2007.

SARDAGNA, Marcelo. **Ferramenta para aplicação do padrão Data Access Object (DAO) em sistemas desenvolvidos com a linguagem de programação Delphi**. 2007. 66 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Regional de Blumenau, Blumenau.

SASSE, Erick. **Convertendo arquivos DFM binários para texto**. [S.l.], 2005. Disponível em: <http://www.ericksasse.com.br/convertendo-arquivos-dfm-binrios-para-texto/>. Acesso em: 6 abr. 2010.

SILVA, Izalmo P.; SILVA, Samuel S. Desenvolvendo com Hibernate. **Java Magazine**, Rio de Janeiro, v. 4, n. 73, p. 113-114, 2009.

SOARES, Sérgio; BORBA, Paulo. **AspectJ**: programação orientada a aspectos em Java. Recife, 2002. Disponível em: <http://www.cin.ufpe.br/~phmb/papers/AspectJTutorialSBLP2002.pdf>. Acesso em: 20 out. 2010.

TIGRIS. **TortoiseSVN**: project home. [S.l.], 2010. Disponível em: <http://tortoisesvn.tigris.org>. Acesso em: 27 out. 2010.

TIOBE SOFTWARE. **TIOBE programming community index for March 2010**. [S.l.], 2010. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 26 mar. 2010.

TROTTIER, Alain. **Java 2 developer exam cram 2.** [S.l.]: Que Publishing, 2004.