UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

PROTÓTIPO DE UM ORM PARA A PLATAFORMA .NET: FRAMEWORK DE MAPEAMENTO OBJETO RELACIONAL

THIAGO BOUFLEUHR

BLUMENAU
2010

2010/1-25

THIAGO BOUFLEUHR

PROTÓTIPO DE UM ORM PARA A PLATAFORMA .NET: FRAMEWORK DE MAPEAMENTO OBJETO RELACIONAL

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 Sistemas de Informação— Bacharelado.

Prof. Adilson Vahldick, Mestre - Orientador

BLUMENAU
2010

2010/1-25

PROTÓTIPO DE UM ORM PARA A PLATAFORMA .NET: FRAMEWORK DE MAPEAMENTO OBJETO RELACIONAL

Por

THIAGO BOUFLEUHR

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

	_____
Presidente:	Prof. Adilson Vahldick, Mestre – Orientador, FURB
	_____
Membro:	Prof. Mauro Marcelo Mattos, Doutor - FURB
	_____
Membro:	Prof. Marcel Hugo, Mestre - FURB

Blumenau, 05 de julho de 2010

Dedico este trabalho especialmente à minha família e a todos que me ajudaram diretamente ou indiretamente na realização deste.

AGRADECIMENTOS

À Deus e ao Universo, proporcionando-me força e motivação para seguir em frente.

À minha família, que mesmo longe, sempre esteve presente.

Ao meu pai pelo incentivo constante a educação.

À minha mãe pela força espiritual e maternal.

Ao meu irmão pela ajuda incondicional na revisão ortográfica e gramatical.

À minha futura esposa Hellen Morás de Andrade na compreensão de minha ausência.

Ao meu orientador, Adilson Vahldick, por ter acreditado na conclusão deste trabalho.

Só me interessam os passos que tive de dar na vida, para chegar a mim mesmo.

Hermann Hesse

RESUMO

O acesso a dados tornou-se indispensável na construção de um sistema responsável por gerar qualquer tipo de informação. Face a isto, surge a necessidade de automatizar este processo a fim de minimizar tempo de desenvolvimento. Este trabalho apresenta o protótipo de um *framework* para mapeamento de objetos C# para o modelo relacional de banco de dados, visando à utilização de informações de mapeamento através de metadados, conhecidos no .NET como atributos customizados, sem a necessidade de arquivos físicos que contenham o mapeamento. Dentre as principais tecnologias utilizadas, destaca-se a Language INtegrated Query (LINQ) sendo responsável por realizar consultas nos objetos em uma base de dados visando a simplicidade na construção do comando. Todos estes recursos foram implementados utilizando o Microsoft Visual Studio 2008 como principal ferramenta de desenvolvimento.

Palavras-chave: Tecnologia. Microsoft. Orm. Mapeamento objeto relacional.

ABSTRACT

The data access became necessary in the software design responsible to generate any kind of information. Deal with this, arises the necessity to automate this process order to minimize development time. This work presents the prototype of a framework to map C# objects to the relational model database, seeking to use mapping information through metadata, known in .NET as custom attributes, without need for physical files containing the mapping. Among the main technologies used, there is the Language INtegrated Query (LINQ) that is responsible for query objects in a database in order to simplify the construction of the command. All these features have been implemented using Microsoft Visual Studio 2008 as primary development tool.

Key-words: Technology. Microsoft. Orm. Object relational mapping.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

DDL – *Data Definition Language*

DLL – *Dynamic Link Library*

DOM – *Document Object Model*

DSC – Departamento de Sistemas e Computação

HTML – *Hyper Text Markup Language*

LINQ – *Language INtegrated Query*

ORM – *Object Relational Mapping*

POCO – *Plain Old CLR Object*

SGBD – Sistema Gerenciador de Banco de Dados

SIS – Curso de Sistemas de Informação – Bacharelado

SQL – *Structured Query Language*

XML - *eXtensible Markup Language*

W3C – *Word Wilde Web Consortium*

SUMÁRIO

1 INTRODUÇÃO

Durante a fase de desenvolvimento de um software, percebe-se a repetição de um problema já resolvido anteriormente, que pode ser automatizado, desde que o mesmo faça uso de uma ferramenta especialista para solução do problema conhecido, como é o caso do acesso a dados, podendo-se utilizar qualquer *framework* que realize esta tarefa da melhor forma possível.

A construção de um *framework* é aceitável, quando o mesmo destina-se a resolver um problema conhecido, atuando especificamente em uma determinada tarefa, pois desta forma o mesmo acaba por dominar o assunto por completo.

Observando-se a plataforma .NET e os acessos a dados existentes, percebeu-se a necessidade de criar uma abstração da programação da camada de persistência de dados, de modo que o desenvolvedor não necessite se preocupar em como esta tarefa é realizada. Assim sendo, os *frameworks* são alternativas que podem ser acoplados em cenários comuns, mas em soluções diferentes, garantindo o aumento na produtividade (GAMMA; JOHNSON; HELM; VLISSIDES, 1994).

Segundo Ambler (1998), a persistência é uma questão que envolve a maneira na qual os objetos são armazenados permanentemente, para que possam estar disponíveis em momentos posteriores em seu estado original. Baseando-se nesse conceito, o protótipo de mapeamento de objetos para o modelo relacional implementa uma solução na qual os objetos escritos em C# são inseridos, recuperados, alterados e excluídos de uma base de dados, com alto nível de abstração do banco de dados utilizado.

As soluções de mapeamento objeto relacional existentes destinadas à plataforma .NET, em especial para linguagem C#, em sua maioria, são desenvolvidas baseadas em *frameworks* existentes para outras plataformas, como o Hibernate para o ambiente Java, por exemplo. Dessa forma, muitas vezes o desenvolvedor acaba por ocupar seu tempo estudando os padrões e metodologias de desenvolvimento adotadas pelas plataformas de origem do *framework* portado, que poucas vezes refletem a realidade da programação em .NET.

Assim sendo, a construção do protótipo vem por seguir os padrões propostos em MSDN (2007), garantindo que a curva de aprendizado seja menor para desenvolvedores habituados ao ambiente Microsoft .NET e nas tecnologias utilizadas na construção, como por exemplo o LINQ e os atributos customizados disponíveis no .NET Framework.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho foi desenvolver um protótipo de um *framework* para mapeamento objeto-relacional voltado para plataforma .NET.

Os objetivos específicos do trabalho foram:

a) mapear os objetos em C# para as entidades relacionais através do uso de atributos customizados;
b) simplificar o uso de chaves primárias compostas;
c) simplificar a utilização de relacionamentos entre as classes mapeadas;
d) eliminar a necessidade de mapeamento através de arquivos *eXtensible Markup Language* (XML);
e) permitir o uso da LINQ para realizar consultas ao banco de dados.

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado da seguinte forma: no primeiro capítulo é apresentada a introdução do assunto bem como os objetivos a serem alcançados em seu desenvolvimento.

O segundo capítulo apresenta a fundamentação teórica sobre os fundamentos do mapeamento objeto relacional, a tecnologia da Microsoft denominada LINQ e os atributos customizados no C#.

Já o terceiro capítulo descreve o desenvolvimento do protótipo, sua especificação, modelagem das classes e implementação.

Por fim, no capítulo quatro é apresentada a conclusão do trabalho desenvolvido, bem como sugestões para possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda assuntos a serem apresentados nas seções a seguir, tais como fundamentos do mapeamento objeto relacional, a tecnologia LINQ, atributos customizados no C#, além dos trabalhos correlatos.

2.1 FUNDAMENTOS DO MAPEAMENTO OBJETO RELACIONAL

A técnica de mapeamento de objetos para o modelo relacional consiste basicamente em representar uma classe, como se fosse uma tabela no banco de dados, sendo as colunas da tabela representadas através de atributos da classe, e as instâncias (objetos) desta classe representando os registros (linhas) contidos na tabela. Por sua vez um atributo pode representar um relacionamento existente com uma tabela (AMBLER, 1998).

As principais vantagens do mapeamento de objetos são a abstração quanto à utilização de diferentes SGBD's, a redução no uso da linguagem *Structured Query Language* (SQL) e o aumento considerado da produtividade, pois a camada de persistência e acesso a dados ficam encapsuladas no *framework*, eliminando a necessidade de desenvolvimento da mesma (AMBLER, 1998).

Dentre as diversas formas de mapeamento existentes, destacam-se o mapeamento através de arquivos XML, anotações de código (linguagem Java), atributos customizados (plataforma .NET), implementação de *interfaces*, dependência por herança, reflexão computacional, dentre outras. A utilização de qualquer uma destas formas tem como objetivo permitir que o *framework* construa os comandos SQL de forma dinâmica, baseado na definição dos atributos mapeados, nome e tipo, por exemplo, (FOWLER, 2002).

Conforme Ambler (2002), os três conceitos básicos de transformação do modelo OO para o modelo relacional são:

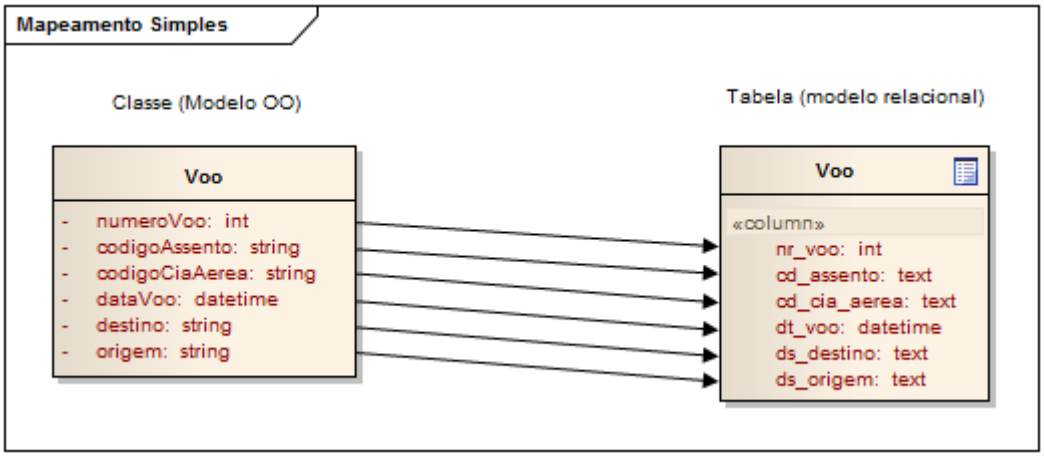
a) mapeamento básico;
b) mapeamento de herança;
c) mapeamento de relacionamentos.

2.1.1 Mapeamento Básico

O mapeamento simples de objetos tem como principal objetivo descrever a forma de persistir apenas as informações relevantes para o modelo relacional e de objetos, garantindo que apenas os atributos de valor primitivo (inteiro, ponto flutuante, *string*, entre outros) sejam mapeados, não levando em consideração os relacionamentos existentes entre duas ou mais classes, onde existe a necessidade de tratar recursivamente o relacionamento (AMBLER, 1998).

Uma forma prática para realizar o mapeamento entre os atributos de uma classe com as colunas da tabela no banco de dados é mapear um atributo para apenas uma coluna de mesmo tipo, ou similar, de dados em ambos os modelos, eliminando a necessidade de tratamento diferenciado para cada coluna, podendo ser preservado o nome da coluna na classe de mapeamento ou criando sinônimos para facilitar a identificação (AMBLER, 2002).

A figura 1 representa o mapeamento básico dos atributos da classe para as colunas da tabela, mantendo a correspondência entre os tipos de dados dos campos em ambos os modelos.



Fonte: Adaptado de Fowler (2002).

Figura 1 – Mapeamento básico um-para-um

2.1.2 Mapeamento de Herança

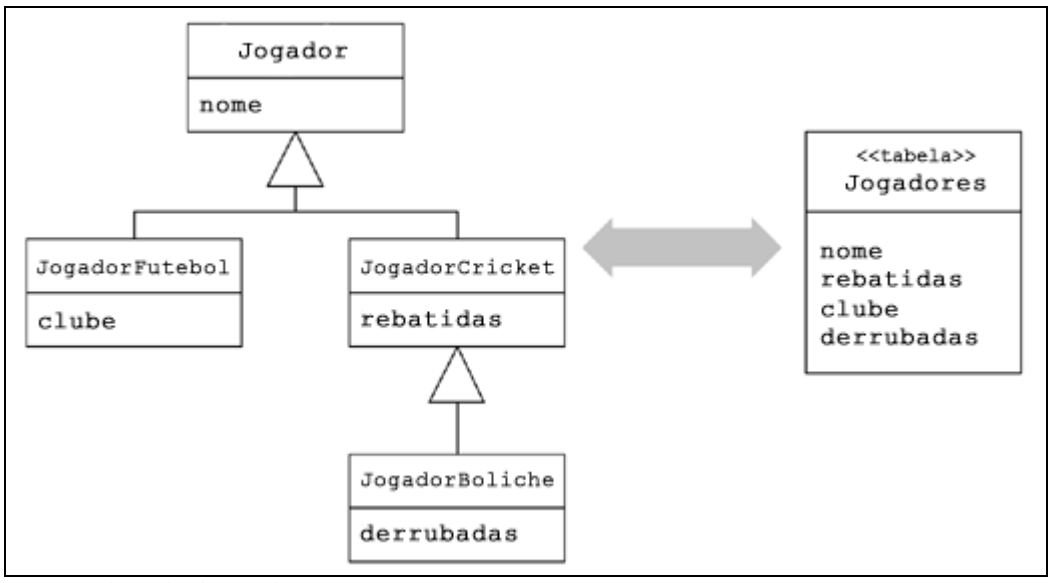
O modelo relacional não possui suporte nativo para utilização do conceito de herança conforme conhecido no modelo de objetos. Devido a isso, o mapeamento das classes que

utilizam esse conceito se torna complexo de realizar, pois o nível do encapsulamento dos atributos da classe ancestral inviabiliza o acesso através da classe concreta. Muitas vezes esse importante princípio (encapsulamento) de modelagem orientada a objetos é infringido para que o mapeamento possa ser realizado pela ferramenta de Object Relational Mapper (ORM) (AMBLER, 2002).

Conforme Fowler (2002), não existe uma forma padrão para realizar o mapeamento de classes que utilizam herança, sem ter que mapear a classe abstrata juntamente com as classes concretas, correlacionando seus atributos com as colunas. Para qualquer tipo de estrutura hierárquica, existem basicamente três opções:

a) uma tabela para todas as classes da hierarquia;
b) uma tabela para cada classe concreta;
c) uma tabela por classe (abstrata e concreta).

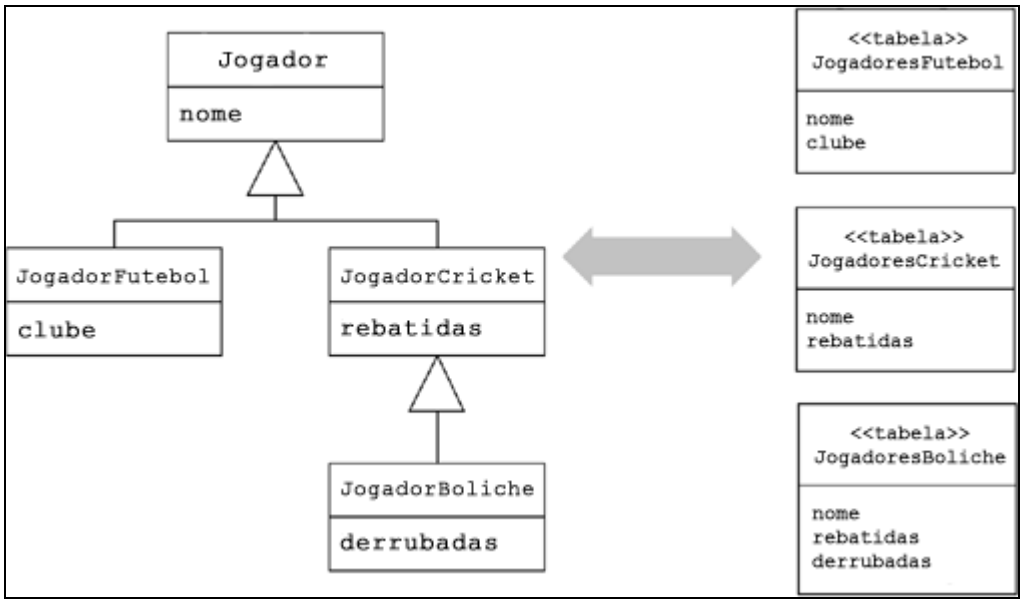
A figura 2 exemplifica um possível mapeamento utilizando uma tabela para todas as classes da hierarquia.



Fonte: Adaptado de Fowler (2002).

Figura 2 – Uma tabela para todas as classes da hierarquia

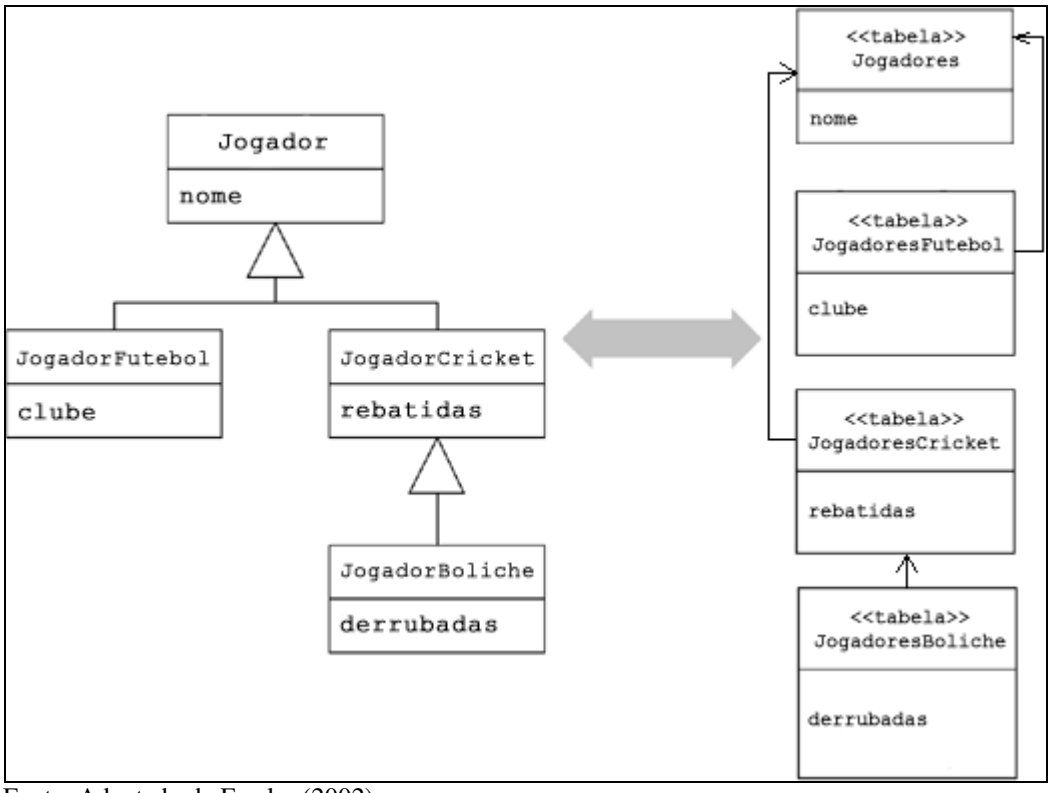
A figura 3 demonstra a modelagem das classes utilizando uma tabela para cada classe concreta.



Fonte: Adaptado de Fowler (2002).

Figura 3 - Uma tabela para cada classe concreta

A figura 4 exibe a construção do mapeamento fazendo uso de uma tabela por classe, tanto abstrata quanto concreta.



Fonte: Adaptado de Fowler (2002).

Figura 4 – Uma tabela por classe

2.1.3 Mapeamento de Relacionamentos

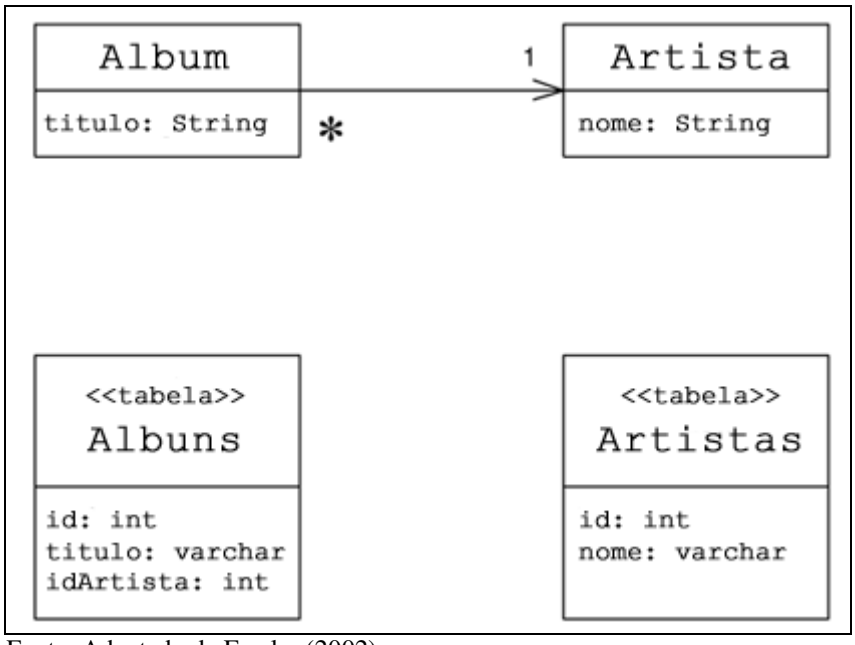
Para que a persistência de um objeto seja realizada por completo, devem-se mapear além dos atributos, os relacionamentos que possam existir com outros objetos, a fim de garantir a integridade do registro quando o mesmo for inserido ou alterado fisicamente no banco de dados.

Para isso, deve-se observar a diferença entre manter um relacionamento no modelo de objetos e no modelo relacional. No modelo de objetos o relacionamento é resolvido mantendo-se a referência em memória do objeto relacionado, enquanto que no modelo relacional o relacionamento é formado por uma chave de outra tabela (FOWLER, 2002).

De acordo com Ambler (2002), existem três tipos de relacionamentos que necessitam ser mapeados: associação, agregação e composição. Dentre as abordagens para mapeamento de relacionamentos existentes destacam-se:

a) relacionamento um-para-um;
b) relacionamento um-para-muitos;
c) relacionamento muitos-para-muitos.

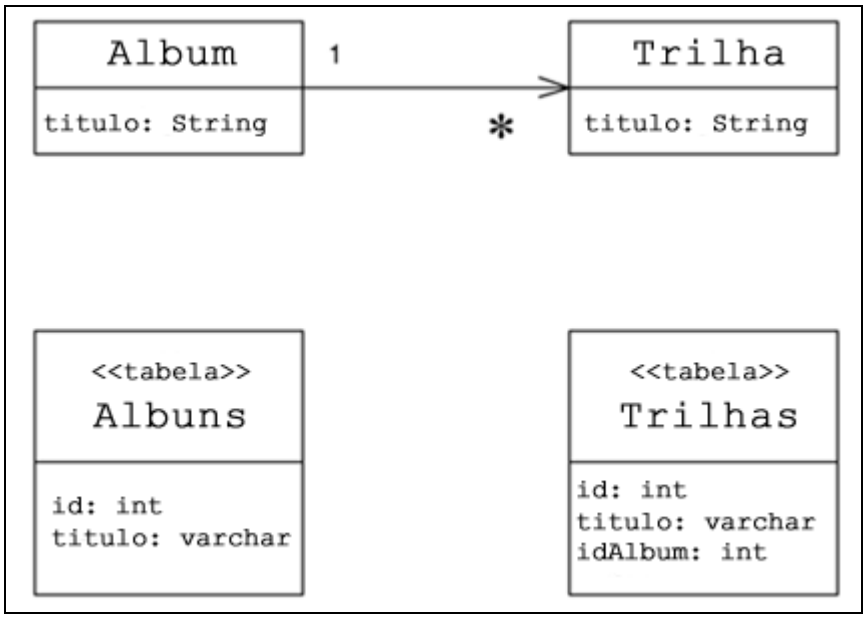
Na figura 5 a classe `Album` irá conter um atributo relacionado com a classe `Artista`, da mesma forma que a tabela `Albuns` contém um relacionamento para a tabela `Artistas`, assim sendo, consiste em um relacionamento de um-para-um.



Fonte: Adaptado de Fowler (2002).

Figura 5 – Mapeamento um-para-um

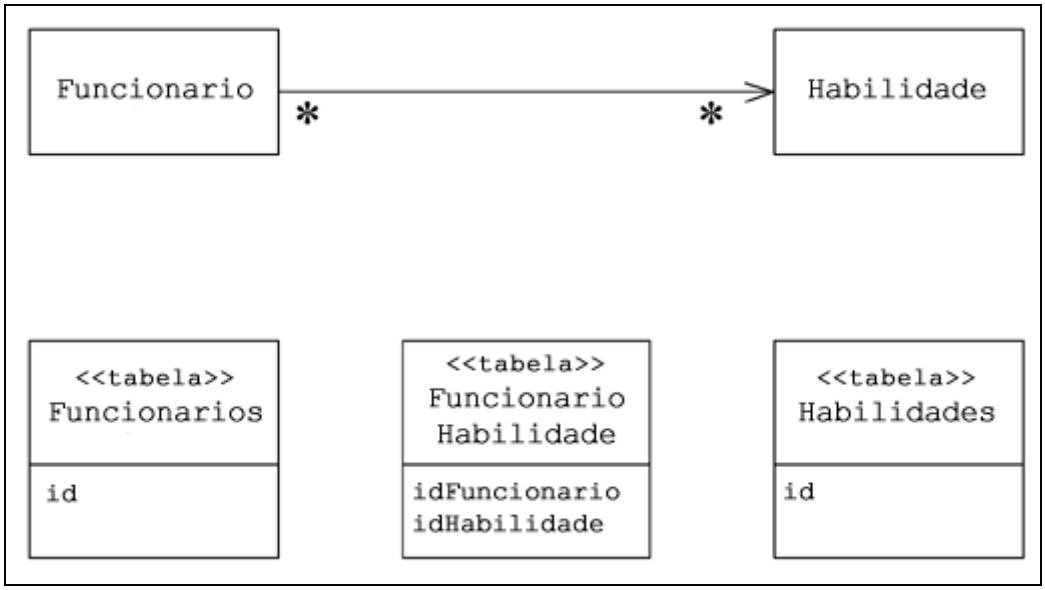
Na figura 6 existe um relacionamento um-para-muitos entre as tabelas `Albuns` e `Trilhas`, ou seja, um álbum contém várias faixas de músicas, gerando no modelo de classes uma coleção de `Trilha` como atributo de relacionamento na classe `Album`.



Fonte: Adaptado de Fowler (2002).

Figura 6 – Mapeamento um-para-muitos

A figura 7 expõe um exemplo de um relacionamento muitos-para-muitos onde um funcionário pode possuir várias habilidades e vice-versa. No modelo relacional este problema é resolvido criando uma tabela associativa, `FuncionarioHabilidade`, entre `Funcionarios` e `Habilidades`. Porém, no modelo de objetos é resolvido criando uma coleção em ambas as classes que possuem este relacionamento.



Fonte: Adaptado de Fowler (2002).

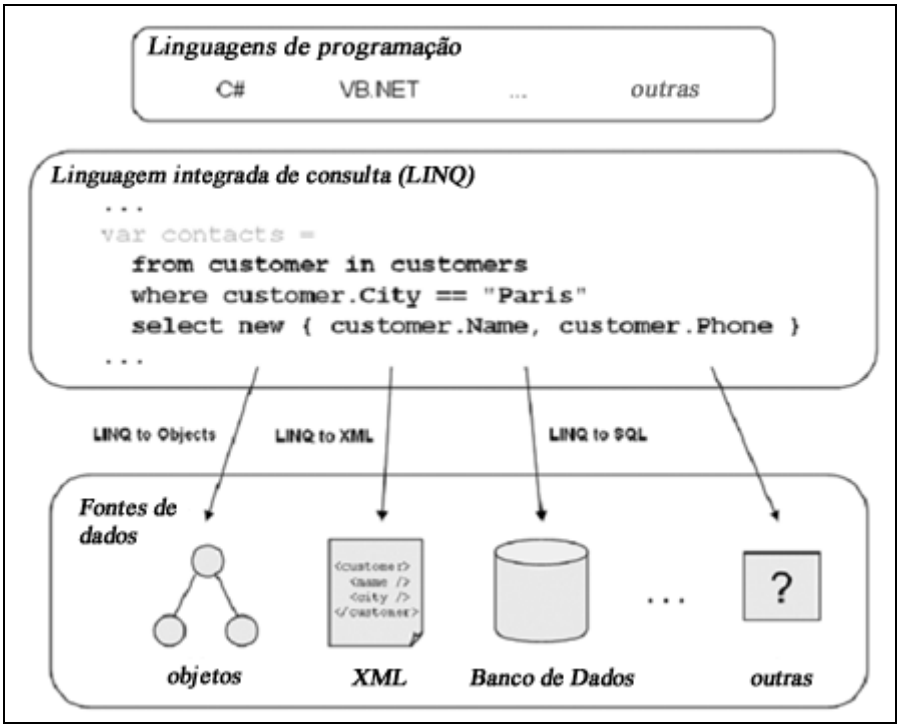
Figura 7 – Mapeamento muitos-para-muitos

2.2 TECNOLOGIA LINQ

De acordo com Marguerie, Eichert e Wolley (2009), a linguagem integrada de consulta da Microsoft, ou simplesmente LINQ, é uma extensão para as linguagens de programação C# e *Visual Basic for .NET* (VB.NET), disponível a partir da versão 3.5 do .NET Framework, utilizada para realizar consultas a quaisquer tipos de fontes de dados, arquivos XML (LINQ *to* XML), base de dados (LINQ *to* SQL, Entity Framework), coleções de objetos (`IEnumerable`) entre outros, usando uma escrita semelhante ao SQL tradicional.

Com o advento do LINQ para a plataforma .NET, algumas mudanças significativas irão ocorrer na forma em que é feita a manipulação e manutenção dos dados dentro das aplicações e componentes, visando um modelo de programação mais declarativo (MARGUERIE; EICHERT; WOLLEY, 2009).

O LINQ é totalmente flexível para ser utilizado em qualquer cenário. Possui métodos de extensão que os tornam automaticamente visíveis em objetos suportados, possibilidade de declarar métodos anônimos (*lambda expressions)* e tipos de dados anônimos (*anonymous type)* que são resolvidos no momento da execução do código, conforme demonstra a figura 8 (MARGUERIE; EICHERT; WOLLEY, 2009).



Fonte: Adaptado de Marguerie; Eichert; Wolley (2009, p. 9).

Figura 8 – Possibilidades de utilização do LINQ.

2.2.1 Vantagens

Uma das principais vantagens do LINQ é a possibilidade de trabalhar com os mais diversos tipos de objetos em um ambiente fortemente tipado, pois as consultas escritas são verificadas em tempo de compilação, reduzindo a probabilidade de erros em tempo de execução.

A seguir são descritas as características principais do LINQ no desenvolvimento utilizando a linguagem C# (MSDN, 2008 – Tradução nossa):

a) linguagem de consulta utilizando uma escrita familiar (SQL);
b) validação de sintaxe e tipos de dados em tempo de compilação;
c) suporte integrado de depuração;
d) suporte a descobrimento automático de membros de um tipo em tempo de desenvolvimento (*IntelliSense*);
e) habilidade de manipular diretamente elementos XML, ao invés da necessidade de utilizar um documento XML completo como *container*, conforme requerido pelo Word Wilde Web Consortium (W3C) Document Object Model (DOM);
f) método eficiente de manipulação de documentos XML em memória, mais simples do que XPath ou XQuery;
g) métodos eficientes de filtragem, ordenação e agrupamento de dados;
h) modelo consistente para trabalhar com manipulação de dados em vários tipos de fontes de dados e formatos.

2.2.2 Estrutura da Linguagem

Para que uma fonte de dados possa ser utilizada com a linguagem LINQ, que seja selecionável, a mesma deve implementar a interface `IQueryable` ou `IQueryable<T>`, contidas no *namespace* `System.Linq` (MEHTA, 2008, p. 40).

O quadro 1 a seguir, exibe um resumo das características da estrutura de utilização da linguagem LINQ.

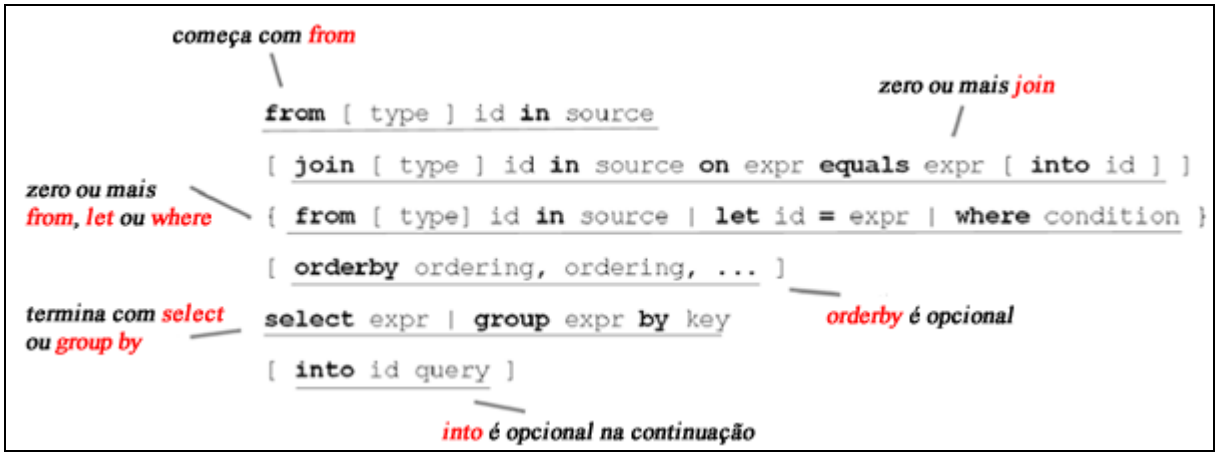
ESTRUTURA DA LINGUAGEM	DESCRIÇÃO
Expressões de consulta	Sintaxe de consulta declarativa utilizada para extrair dados de qualquer fonte de dados LINQ
Variáveis implicitamente tipadas	Uma variável precedida pela palavra chave *var* ativa o compilador a inferir seu tipo em tempo de compilação
Inicializador de objetos	Habilita a inicialização de um objeto sem a necessidade de executar um construtor
Tipos anônimos	Habilita o compilador a criar objetos sem especificar seu tipo. Disponível apenas em tempo de compilação
Extensões de métodos	Permite adicionar um método estático a um tipo existente
Expressões *lambda*	Expressão *inline* ou bloco de comandos, onde um *delegate* é esperado

Fonte: MSDN (2008 – Tradução nossa).

Quadro 1 – Estrutura da linguagem LINQ

2.2.2.1 Sintaxe Básica

Assim como em toda linguagem de programação, deve-se seguir uma sintaxe básica para que a mesma funcione corretamente. A figura 9 demonstra a estrutura e sintaxe básica de uma expressão LINQ.



Fonte: Adaptado de Marguerie; Eichert; Wolley (2009, p. 99).

Figura 9 – Estrutura sintática de expressão de consultas

A figura 10 ilustra um exemplo utilizando LINQ com a linguagem C#, para acessar uma fonte de dados relacional, selecionando os clientes (`customers`) cujo nome do cliente inicia-se com a letra “A” e o número de encomendas (`Orders`) seja maior que zero. Pode-se

observar que as palavras chaves *from, where*, *order by* e *select* são semelhantes à linguagem SQL.

```
from customer in customers
where customer.Name.StartsWith("A") && customer.Orders.Count > 0
orderby customer.Name
select new { customer.Name, customer.Orders }
```

Fonte: Marguerie; Eichert; Wolley (2009, p. 12).

Figura 10 – Exemplo de utilização do LINQ

2.3 ATRIBUTOS CUSTOMIZADOS NO C#

Segundo Troelsen (2007, p. 542), *custom attributes,* ou anotações de código, é uma forma disponibilizada pelo .NET Framework para que os programadores possam inserir informações no formato de metadados em um determinado tipo, classe, interface, estrutura, atributo, propriedade, método, entre outros.

Os atributos customizados são tipos de classes comuns que descendem da classe base `System.Attribute`. Algumas funcionalidades das anotações, por exemplo, informar se uma classe é *serializável*, se um método é executado sincronizado entre as *threads*, marcar um tipo como obsoleto, ou ainda pode-se criar atributos customizados derivando da classe base `Attribute` (TROELSEN, 2007, p. 546).

2.3.1 Desenvolvendo Atributos Customizados

Conforme Troelsen (2007, p. 546), o primeiro passo na construção de um atributo customizado é criar uma nova classe que derive de `System.Attribute`. Como segundo passo, toma-se como exemplo uma biblioteca de classes que contenha diversos tipos de veículos, no qual necessita-se incluir descrições em cada uma delas, no qual será utilizado um atributo customizado denominado `DescricaoVeiculoAttribute`.

A figura 11 exemplifica a criação do atributo customizado `DescricaoVeiculoAttribute`.

```
// Um Atributo Customizado
public sealed class DescricaoVeiculoAttribute : System.Attribute
{
    private String descricao;

    public DescricaoVeiculoAttribute()
    {

    }

    public DescricaoVeiculoAttribute(String descricao)
    {
        this.descricao = descricao;
    }

    public String Descricao
    {
        get { return this.descricao; }
        set { this.descricao = value; }
    }
}
```

Fonte: Adaptado de Troelsen (2007, p. 547).

Figura 11 – Atributo customizado `DescricaoVeiculoAttribute`

A classe `DescricaoVeiculoAttribute` possui um atributo privado denominado `descricao`, no qual armazena a descrição do veículo, a manipulação deste atributo dá-se através do construtor público ou da propriedade pública `Descricao`.

2.3.2 Utilizando Atributos Customizados

A figura 12 demonstra a utilização do atributo customizado `DescricaoVeiculoAttribute` criado no tópico anterior, e de mais dois que acompanham o .NET Framework, `Serializable` e `ObsoleteAttribute`, um ponto interessante a observar é que se pode utilizar apenas o início do nome da classe, excluindo o sufixo *Attribute*, quando se trata de atributos customizados (este sufixo é uma convenção para programação de atributos customizados na plataforma .NET).

```
// Adicionando uma descrição através de "propriedades nomeadas"
[Serializable]
[DescricaoVeiculo(Descricao = "Minha adorada Harley")]
public class Motocicleta
{

}

[Serializable]
[Obsolete("Utilize outro veículo!")]
[DescricaoVeiculo(Descricao = "Conduzida pela velha égua cinzenta")]
public class Carroca
{

}

[DescricaoVeiculo(Descricao = "Um carro longo, lento, mais com muitas qualidades")]
public class MotorHome
{

}
```

Fonte: Adaptado de Troelsen (2007, p. 547).

Figura 12 – Utilização de atributos customizados

Como pode-se verificar na figura 12, as classes `Motocicleta` e `Carroca` são decoradas com o atributo `Serializable` que é necessário para uma classe ser serializada. A classe `Carroca` é decorada com o atributo `Obsolete`, marcando o mesmo como obsoleto, dessa forma quando esta classe for utilizada o compilador irá gerar um *warning* exibindo a mensagem que foi configurada no atributo.

2.4 TRABALHOS CORRELATOS

Devido ao grande número de trabalhos correlatos encontrados com o tema sobre mapeamento objeto-relacional, seguem abaixo os que mais se destacam.

O produto NHibernate (MAULO, 2008) é uma versão para .NET do framework Hibernate para ambientes Java, com as mesmas funcionalidades da versão original, para a versão reescrita em C#, porém o mapeamento principal é mantido através de arquivos XML. Um *plugin* pode ser incorporado ao *framework* para gerar os XML's automaticamente.

Na monografia de Silva (2005), o mesmo expõe as problemáticas e possíveis soluções do mapeamento de objetos para o paradigma relacional, usando como base a ferramenta Hibernate. O mesmo sugere que o mecanismo de persistência deve ser desacoplado do mecanismo de mapeamento, evitando problemas de desempenho e tornando a solução simples

de manter. Dessa forma, o protótipo separa as camadas de acesso a dados e de mapeamento em componentes independentes, garantindo melhor portabilidade em possíveis expansões.

Dentre os trabalhos correlatos pesquisados, o que obteve maior destaque foi o projeto SubSonic (CONERY, 2009), sendo um *framework* que resolve o problema de mapeamento de objetos utilizando *templates* de classes mapeadas automaticamente de acordo com a conexão de banco de dados configurada no mecanismo. O trabalho proposto utilizou a técnica de repositório implementada pelo mesmo.

3 DESENVOLVIMENTO DO PROTÓTIPO

Esta seção tem por objetivo demonstrar as fases executadas para concepção, análise, *design* e desenvolvimento do protótipo, seguindo a seguinte ordem de apresentação: análise dos requisitos contemplando os requisitos funcionais e não funcionais, a especificação contendo os principais casos de uso, diagrama de classes, diagrama de atividades, seguidos pelo desenvolvimento (implementação) e dos resultados obtidos.

3.1 ANÁLISE DOS REQUISITOS

Com base na idéia proposta de construção de um *framework* responsável por mapear estruturas lógicas de classes para estruturas físicas de tabelas, a seguir são demonstrados os levantamentos dos requisitos funcionais, requisitos não funcionais e regras de negócio, estabelecidos pela análise final.

O quadro 2 apresenta os requisitos funcionais previstos para o protótipo e sua rastreabilidade, ou seja, vinculando com o caso de uso associado.

Requisitos Funcionais	Caso de Uso
RF01: O framework deverá permitir o mapeamento de classes C# para o modelo relacional utilizando atributos customizados.	UC01
RF02: O framework deverá permitir que seja informado em tempo de execução o SGBD que será utilizado.	UC02
RF03: O framework deverá suportar a operação de inserção de objetos mapeados em um banco de dados.	UC03
RF04: O framework deverá suportar a operação de alteração de objetos mapeados existentes em um banco de dados.	UC04
RF05: O framework deverá suportar a operação de exclusão de registros mapeados em um banco de dados.	UC05
RF06: O framework deverá disponibilizar o uso customizado do LINQ para consultas ao banco de dados através de classes mapeadas.	UC06
RF07: O framework deverá disponibilizar uma classe para o acesso	UC07

customizado ao banco de dados através da linguagem SQL.	

Quadro 2 - Requisitos funcionais

O quadro 3 lista os requisitos não funcionais previstos para o protótipo.

Requisitos Não Funcionais
RNF01: O framework deverá suportar os SGBD's SQLServer, MySQL e PostgreSQL.
RNF02: O framework deverá ser desenvolvido utilizando a linguagem C# juntamente com o .NET Framework 3.5 ou superior.
RNF03: O framework deverá manter um *cache* dos metadados da classe.
RNF04: O framework deverá gerar um *log* em memória dos últimos 10 comandos enviados ao SGBD.

Quadro 3 - Requisitos não funcionais

O quadro 4 lista as principais regras de negócio para o protótipo.

Regras de Negócio	**Caso de Uso**
RN01: O framework deverá disponibilizar um *custom attribute* a fim de mapear uma classe para uma tabela.	UC01
RN02: O framework deverá disponibilizar um *custom attribute* a fim de mapear uma propriedade da classe para uma coluna da tabela.	UC01

Quadro 4 – Regras de negócio

3.2 ESPECIFICAÇÃO

A especificação do protótipo foi realizada utilizando a ferramenta Enterprise Architect versão 7.5 para construção dos diagramas de casos de uso, diagramas de classes e diagramas de atividades, que são apresentados a seguir na respectiva ordem.

3.2.1 Diagrama de Casos de Uso

Esta seção apresenta os diagramas de casos de uso do protótipo de mapeamento de objetos para a plataforma .NET, sendo que o detalhamento dos principais casos de uso, quando necessário, estará disponível no Apêndice A – Detalhamento dos casos de uso.

Na figura 13 tem-se o diagrama de mapeamento das classes e configuração do SGBD.

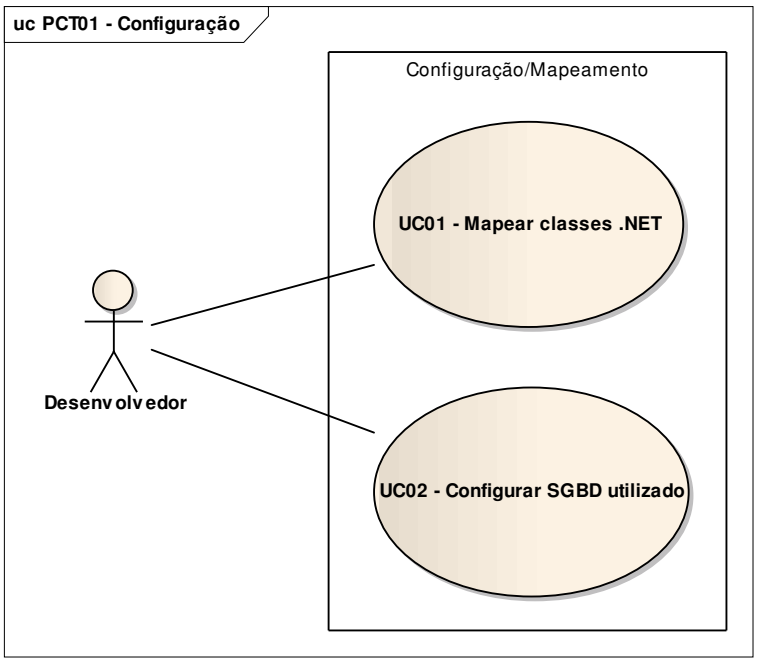


Figura 13 - Caso de uso de configuração

Na figura 14 apresenta-se o diagrama relacionado com as operações que poderão ser executadas pelo desenvolvedor após o mapeamento das classes e configuração do SGBD.

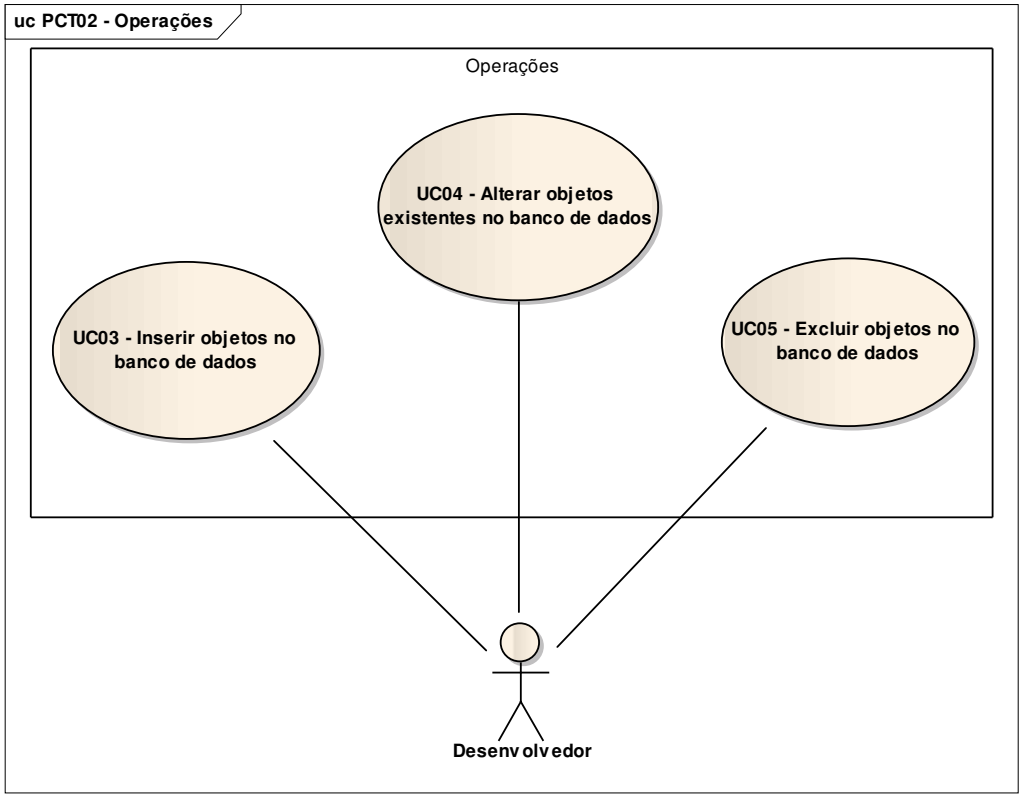


Figura 14 - Caso de uso das operações

Na figura 15 o caso de uso representa as formas que poderão ser utilizadas para recuperar um ou mais objetos persistidos em uma base de dados.

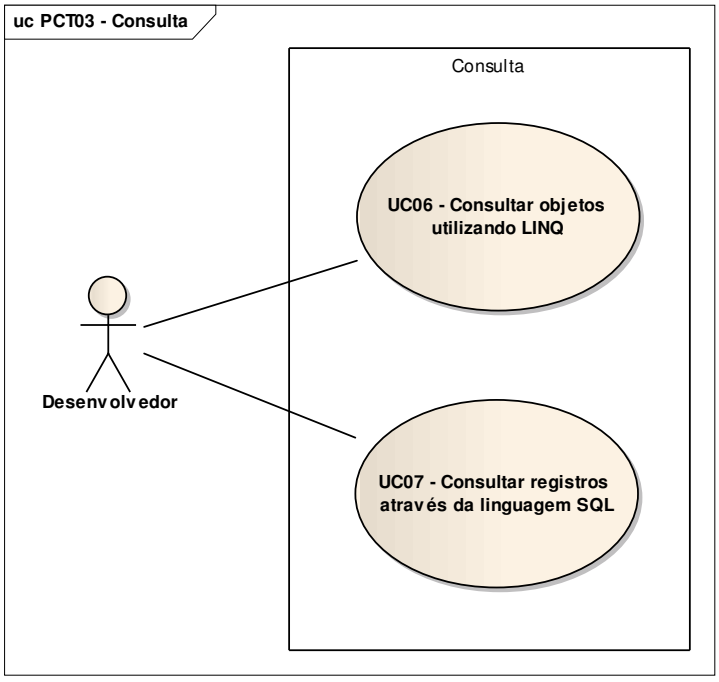


Figura 15 - Caso de uso de consultas

3.2.2 Diagrama de Classes

Os diagramas a seguir demonstram a arquitetura do protótipo, pois o mesmo não faz uso de informações persistidas no banco de dados para fins de controle da ferramenta.

Na figura 16 são apresentadas as classes responsáveis por especificar uma implementação particular para os diferentes SGBD's dentro da camada de acesso a dados.

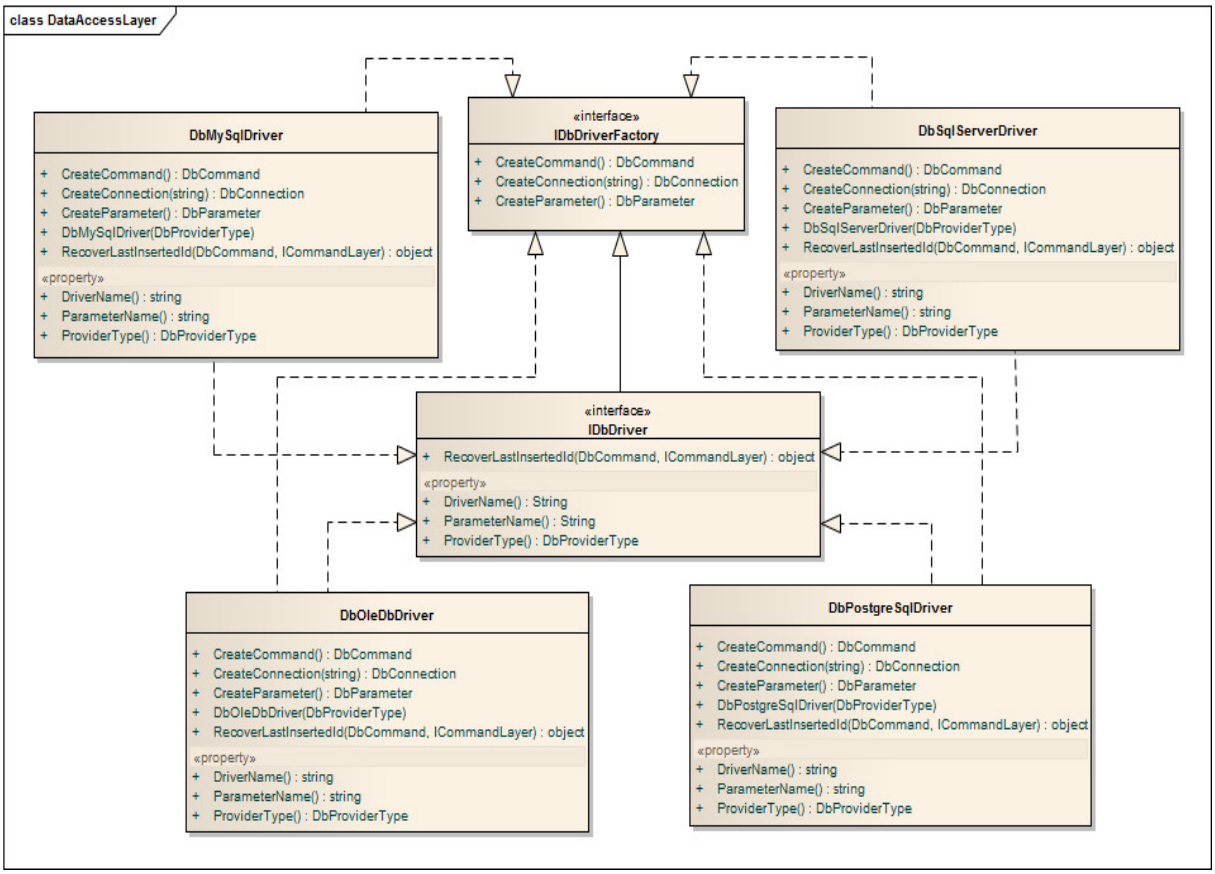


Figura 16 – Camada de acesso a dados específica de um SGBD

A figura 17 é uma continuação da modelagem da camada de acesso a dados demonstrando as classes responsáveis por abstrair o banco de dados que está sendo utilizado bem como auxiliares para criação e manutenção do clico de vida dessa abstração.

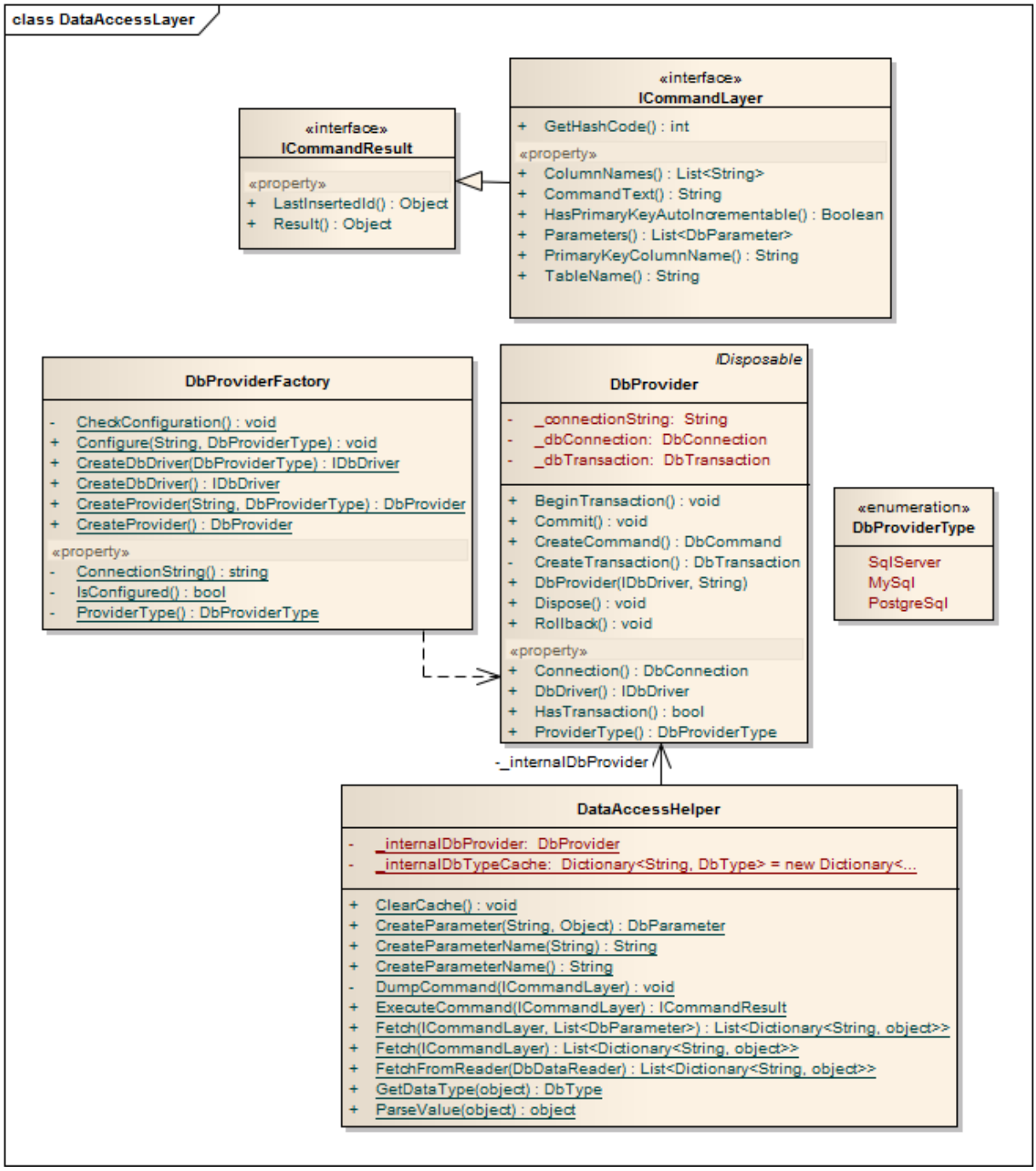


Figura 17 – Classes responsáveis por abstrair o acesso ao banco de dados

A figura 18 apresenta as classes utilizadas para realizar a montagem do comando que será enviado ao banco de dados, *insert*, *delete*, *update* e o comando customizado, bem como seus criadores (*factory*).

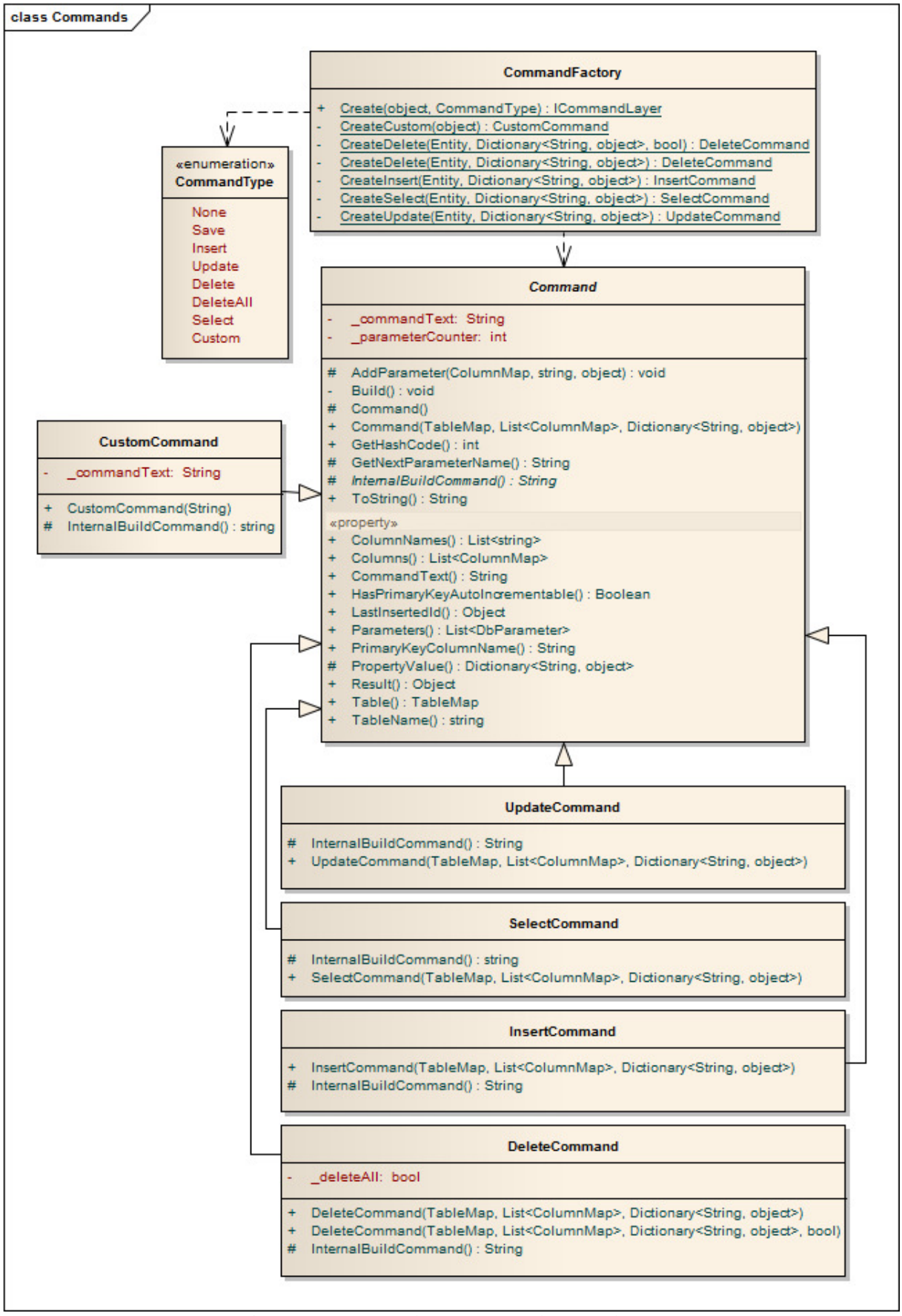


Figura 18 – Comandos representados por classes

Os atributos que serão utilizados para realizar o mapeamento das classes e propriedades bem como suas dependências e classes auxiliares são demonstrados na figura 19.

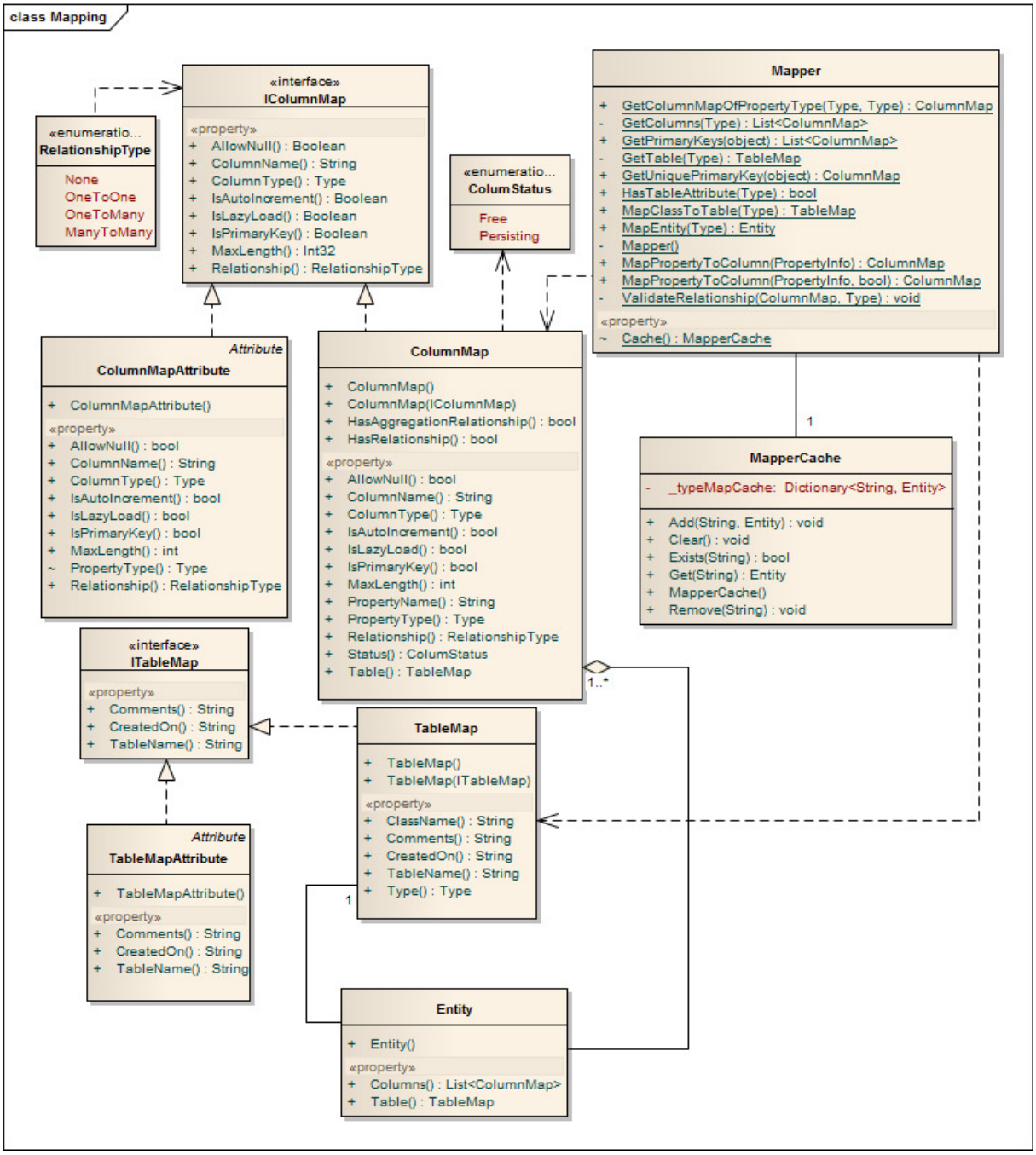


Figura 19 – Classes responsáveis pelo mapeamento

A forma como os dados são recuperados e armazenados através da interação com o objeto no banco de dados são controlados através das classes ilustradas na figura 20.

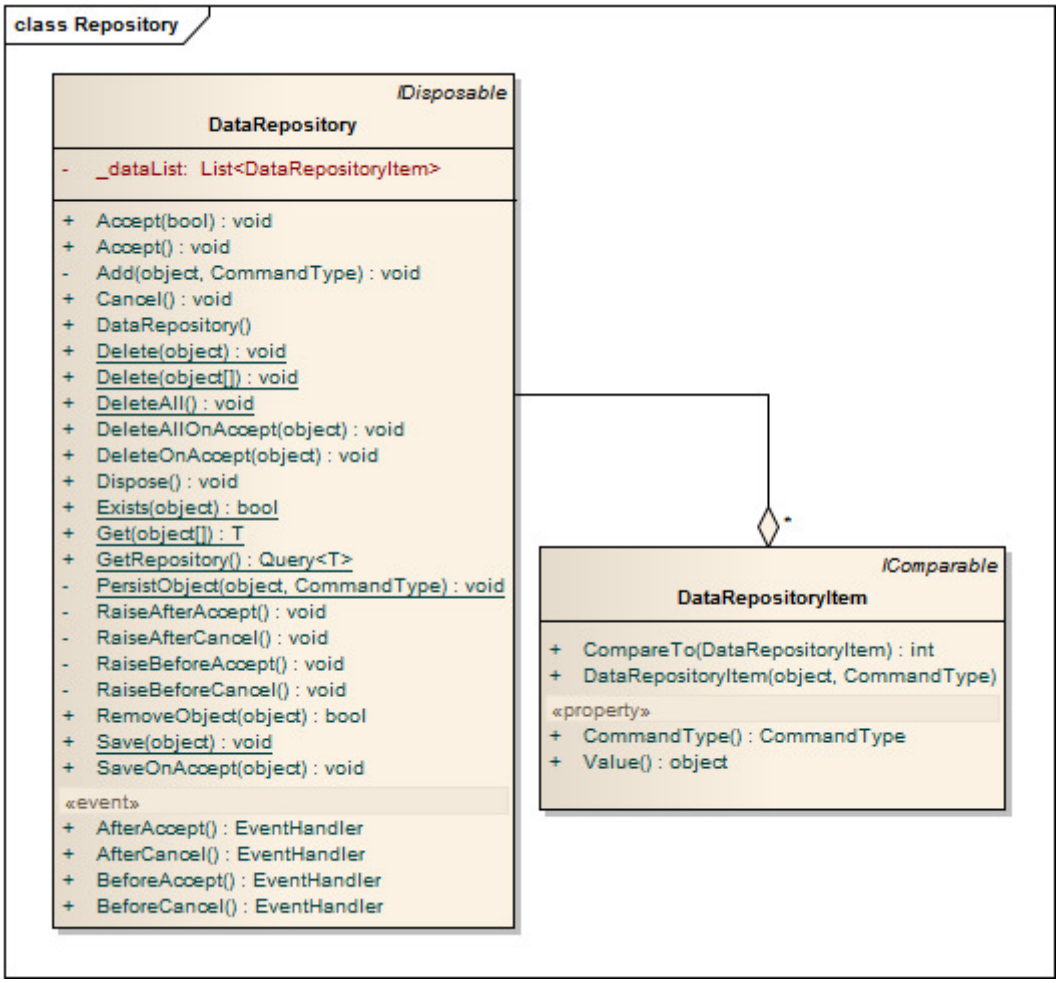


Figura 20 – Padrão *repository* para armazenagem e recuperação de objetos

Por fim, na figura 21 são apresentadas as classes auxiliares que são utilizadas largamente durante o ciclo de vida do protótipo.

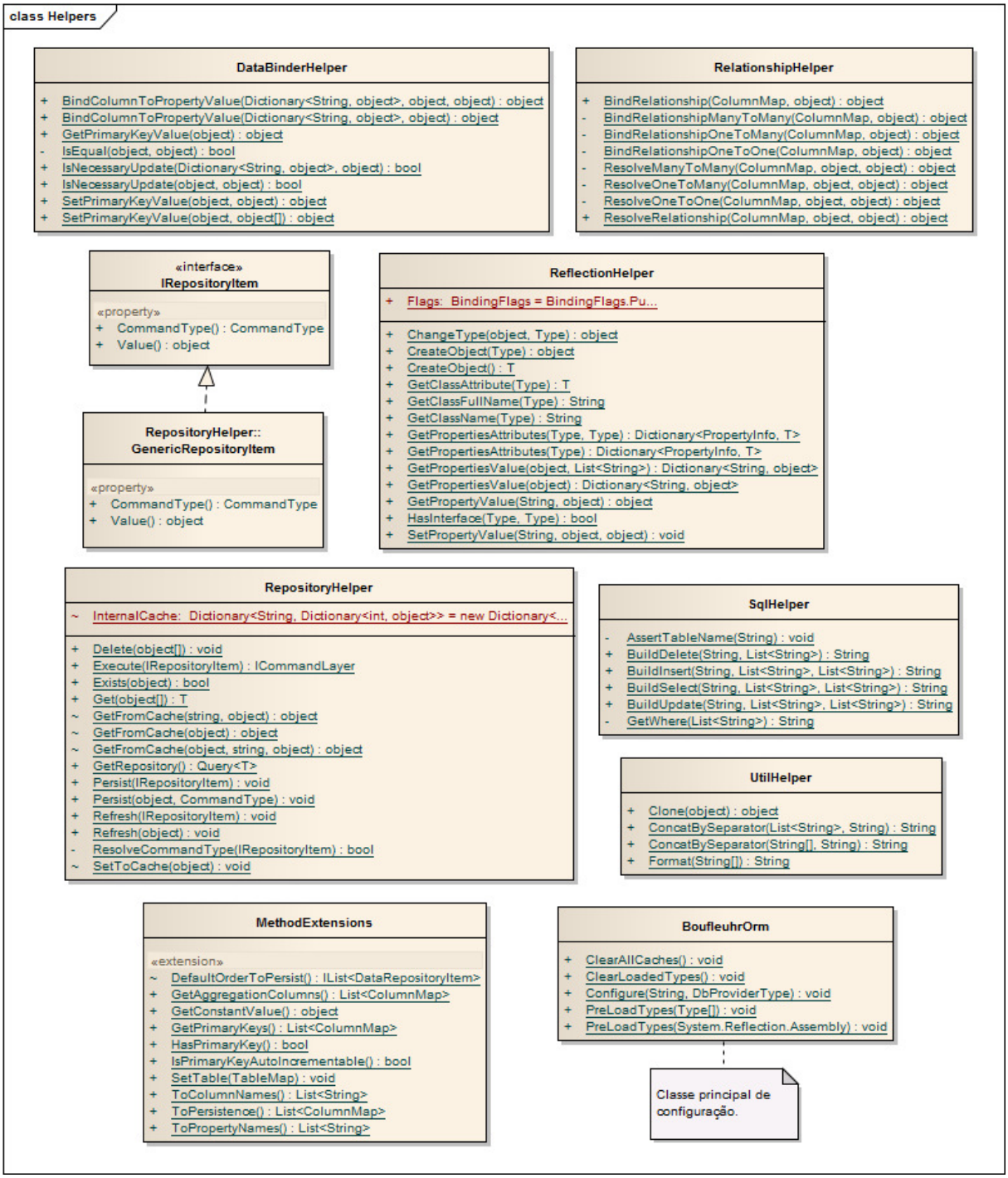


Figura 21 – Classes auxiliares e de configuração

O quadro 5 apresenta a listagem textual das classes/interfaces desenvolvidas e suas devidas responsabilidades.

NOME DA CLASSE/INTERFACE	**RESPONSABILIDADE**
`BoufleuhrOrm`	Ponto de entrada para configuração do ORM.
`ColumnMap`	Representa as informações de mapeamento de uma propriedade para uma coluna.
`ColumnMapAttribute`	Atributo customizado para mapeamento de propriedades para colunas.
`ColumStatus`	Representa o status da coluna no contexto de persistência
`Command`	Utiliza o padrão t*emplate method* para definir o comando enviado para o SGBD.
`CommandConstants`	Constantes contendo palavras chaves da linguagem SQL.
`CommandFactory`	Responsável por criar comandos de execução no banco de dados.
`CommandType`	Representa o tipo do comando de execução.
`CustomCommand`	Representa a implementação de um comando customizado.
`DataAccessHelper`	*Helper* para acesso a dados.
`DataBinderHelper`	*Helper* para *binding* de propriedades utilizando reflexão computacional.
`DataRepository`	Representa um repositório de dados.
`DataRepositoryItem`	Representa um ítem de repositório.
`DbMySqlDriver`	Classe controladora das particularidades do SGBD MySQL.
`DbOleDbDriver`	Classe controladora das particularidades do acesso OleDB .
`DbPostgreSqlDriver`	Classe controladora das particularidades do SGBD PostgreSQL.
`DbProvider`	Classe responsável por abstrair a conexão com o provedor de dados e agrupar funções específicas de cada SGBD.
`DbProviderFactory`	Responsável por criar o provedor de dados correto.
`DbProviderType`	Representa o tipo do provedor de dados.
`DbSqlServerDriver`	Classe controladora das particularidades do SGBD

	SqlServer.
`DeleteCommand`	Representa a implementação de um comando *delete*.
`Entity`	Representa informações da entidade mapeada.
`GenericRepositoryItem`	Classe utilizada para representar um item genérico de repositório.
`IColumnMap`	Interface padrão de mapeamento de colunas.
`ICommandLayer`	Interface padrão de envio de comandos customizados para o SGBD.
`ICommandResult`	Interface responsável por encapsular o retorno de um comando.
`IDbDriver`	Interface padrão para os *drivers* de comunicação com o SGBD
`IDbDriverFactory`	Interface padrão de criação dos objetos de acesso a dados de cada SGBD.
`InsertCommand`	Representa a implementação de um comando *insert*.
`IRepositoryItem`	Representa um objeto persistível.
`ITableMap`	Interface padrão de mapeamento de tabela.
`Mapper`	Responsável por extrair o mapeamento de classes que utilizam os atributos de mapeamento.
`MapperCache`	*Cache* de tipos mapeados.
`MethodExtensions`	Repositório de extensões de métodos.
`ReflectionHelper`	*Helper* para utilizar reflexão.
`RelationshipHelper`	*Helper* para trabalhar com relacionamentos entre objetos persistíveis.
`RelationshipType`	Tipo do relacionamento entre objetos.
`RepositoryHelper`	*Helper* para trabalhar com persistência de objetos.
`SelectCommand`	Representa a implementação de um comando *select*.
`SqlHelper`	*Helper* para tratar de comandos SQL.
`TableMap`	Representa a classe mapeada para uma tabela.
`TableMapAttribute`	Atributo customizado que representa uma classe mapeada.
`UpdateCommand`	Representa a implementação de um comando *update*.
`UtilHelper`	*Helper* para funções diversas.

Quadro 5 – Listagem das classes desenvolvidas no *framework*

3.2.3 Diagrama de Atividades

Quando um objeto é submetido a alguma operação de persistência (*save, delete*), o mesmo é avaliado pelo mecanismo interno do *framework* a fim de realizar algumas validações básicas como, por exemplo, se o objeto é proveniente de uma classe mapeada através dos atributos customizados disponibilizados pelo *framework*, entre outras. A figura 22 apresenta o ciclo de vida de um objeto que é enviado ao repositório de dados quando uma operação de persistência é disparada.

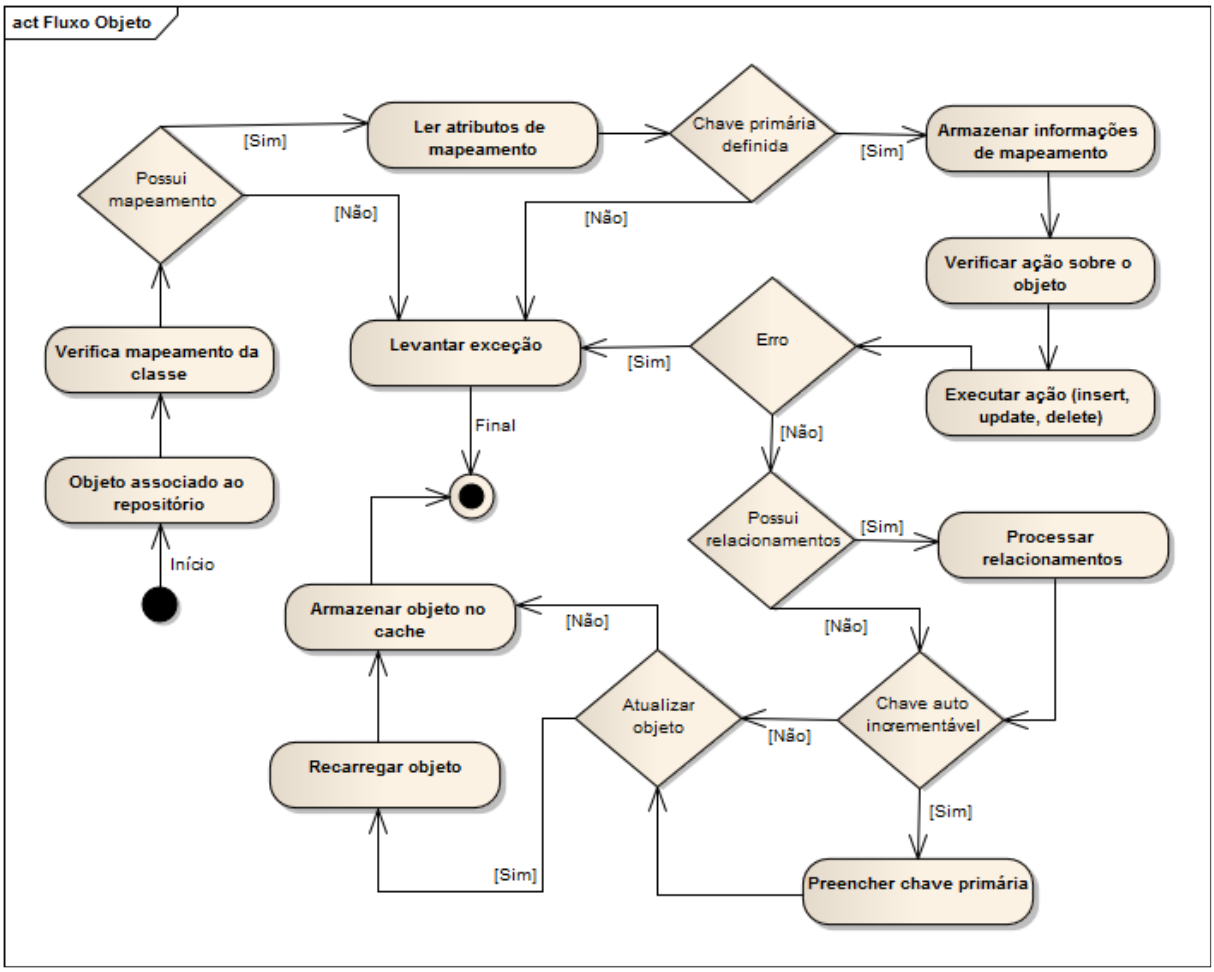


Figura 22 – Ciclo de vida de um objeto mapeado

A fim de utilizar corretamente os recursos disponibilizados pelo *framework*, a figura 23 exibe o fluxo necessário que o usuário (desenvolvedor) deve realizar para utilizar o protótipo de maneira correta.

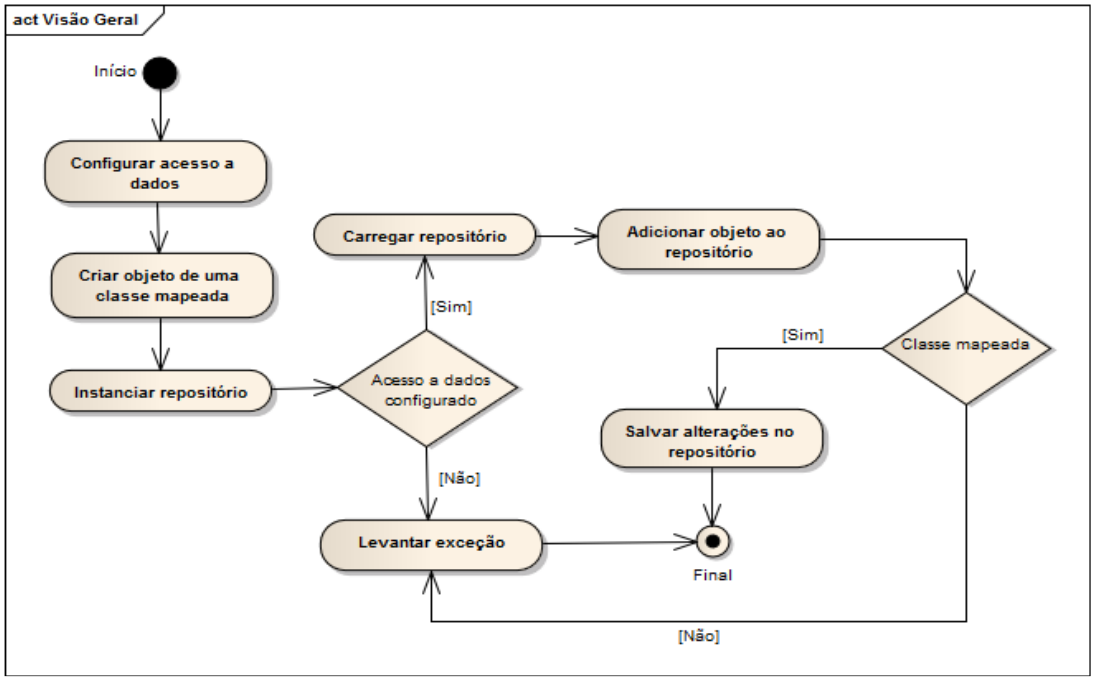


Figura 23 – Visão macro do uso do protótipo

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as ferramentas e técnicas utilizadas para construção do protótipo bem como a operacionalidade da implementação através de um estudo de caso do ponto de vista de um desenvolvedor de produtos comerciais.

3.3.1 Ferramentas e técnicas utilizadas

O protótipo foi desenvolvido utilizando a linguagem de programação C# sob o Microsoft .NET Framework 3.5 Service Pack 1 juntamente com o ambiente de desenvolvimento integrado Microsoft Visual Studio 2008 Professional da Microsoft, disponibilizado sob licença de uso acadêmica.

Foram utilizados os bancos de dados SQL Server 2008, MySQL Server 5.1 e PostgreSQL 8.4.2 para realizar testes com a implementação realizada na camada de acesso a dados bem como para mapear divergências estruturais específicas de cada gerenciador de banco de dados.

3.3.1.1 Acesso a dados

Em .NET o acesso ao banco de dados se dá por intermédio de *providers* disponibilizados pelas empresas que distribuem o SGBD. Um *provider* é uma Dynamic Link Library (DLL) específica de cada SGBD que é referenciada no projeto do Visual Studio para que suas classes possam ser utilizadas programaticamente a fim de conectar e efetuar comandos no banco de dados. Portanto, este protótipo faz uso dos *providers* do SQL Server (disponibilizados juntamente com .NET Framework), do MySQL e PostgreSQL que são obtidos separadamente através do *web site* de cada empresa.

3.3.1.2 Mapeamento de classes e propriedades

A extração dos metadados da classe que contém o mapeamento por meio de atributos customizados é realizada através do uso da técnica de reflexão computacional na qual obtém em tempo de execução o conteúdo preenchido nestes atributos, possibilitando dessa forma descobrir o mapeamento entre a classe e a tabela, propriedades e suas colunas.

A figura 24 exibe um trecho do código que obtém o mapeamento de uma classe para uma tabela por meio da técnica de reflexão computacional.

```
/// <summary>
/// Realiza a extração das informações de mapeamento de uma classe para uma tabela
/// </summary>
/// <param name="typeToMap">Type da classe mapeada</param>
/// <returns>TableMap</returns>
private static TableMap GetTable(Type typeToMap)
{
    // Recupera o atributo customizado utilizando reflexão
    var tableAttribute = ReflectionHelper.GetClassAttribute<TableMapAttribute>(typeToMap);
    if (tableAttribute == null)
        throw new InvalidOperationException(
            String.Format(Resources.ClassNotMapped,
            typeToMap.FullName,
            typeof(TableMapAttribute).Name)
        );

    // Preenche as informações de mapeamento
    var className = ReflectionHelper.GetClassName(typeToMap);
    TableMap table = new TableMap(tableAttribute)
    {
        ClassName = className,
        TableName = className.ToUpper(),
        Type = typeToMap
    };

    if (!String.IsNullOrEmpty(tableAttribute.TableName))
        table.TableName = tableAttribute.TableName.ToUpper();

    return table;
}
```

Figura 24 – Extração do mapeamento de uma classe para uma tabela

As colunas da tabela mapeada são representadas somente pelas propriedades (*get* e *set*) existentes na classe de origem, dessa forma, algumas informações serão necessárias para o *binding* da propriedade com a coluna correspondente fisicamente no banco de dados, como por exemplo, o nome da coluna (quando o nome da propriedade não corresponder igualitariamente ao nome da coluna), se é o caso de chave primária, dentre outras. Para isso, a propriedade que se deseja mapear deverá ser decorada com o *custom attribute* `ColumMap`, identificando cada particularidade quando necessária.

O quadro 6 apresenta a listagem dos atributos customizados desenvolvidos e suas responsabilidades.

NOME DO ATRIBUTO	**RESPONSABILIDADE**
`TableMap`	Marcar a classe como sendo uma classe mapeada, bem como armazenar o nome da tabela do mapeamento.
`ColumMap`	Marcar uma propriedade de uma classe como sendo uma propriedade mapeada para uma coluna, responsável por armazenar o nome da coluna no banco de dados, representa chave primária ou não, valor nulo ou não e se representa uma coluna auto incrementável.

Quadro 6 – Listagem dos atributos customizados desenvolvidos

Efetuado o mapeamento, a forma de persistência é resolvida implementando uma variação do padrão de projeto *Flyweight* proposto em Gamma, Johnson, Helm e Vlissides (1994), também conhecido como *Repository*, melhor explicado em Fowler (2002). Esta implementação atua como um *container* de objetos que são inseridos, alterados e/ou excluídos do repositório, conforme operação definida no contexto, evitando assim a necessidade da classe mapeada herdar um tipo específico do *framework*, assim o desenvolvedor fica livre para descender de suas próprias classes.

A fim de contemplar o mapeamento de herança, o protótipo faz uso do conceito de mapear todas as classes da hierarquia para uma única tabela no banco de dados, conforme explicado anteriormente na fundamentação teórica deste trabalho, dessa forma pode-se trabalhar a abstração das informações em diferentes níveis comportamentais.

A consulta ao banco de dados através de objetos mapeados, faz uso da tecnologia LINQ. Dentre os principais motivos da escolha, destacam-se a similaridade com o SQL, a possibilidade de realizar a consulta ao banco de dados sob demanda (dentro de um laço *foreach,* por exemplo), a redução de erros em tempo de execução, pois as consultas escritas são compiladas utilizando inferência de tipo.

Para o uso customizado do LINQ foram utilizados os conceitos e técnicas para a construção de um provedor customizado de LINQ, conforme demonstrado em Warren (2007).

3.3.1.3 Construção dos comandos SQL

Os comandos SQL de *insert*, *update* e *delete* são gerados automaticamente baseando-se na classe e em suas propriedades mapeadas. Essa geração é coordenada pelas classes envolvidas na figura 18, conforme apresentada anteriormente nos diagramas de classe. Devido aos bancos de dados utilizarem variações da linguagem SQL, adotou-se o padrão SQL American National Standards Institute (ANSI) para construção dos comandos a fim de garantir uma normatização entre diferentes SGBD's (ANSI, 2008).

Para os comandos SQL de consulta, *select*, a forma de construção é baseada nos metadados que o LINQ disponibiliza em tempo de execução através da expressão de consulta montada. Esta expressão de consulta é criada pelo LINQ utilizando uma árvore binária, onde cada nodo representa um objeto de consulta tipado. Dessa forma, o protótipo percorre esta árvore e realiza a tradução de comandos LINQ para comandos SQL de consulta. A figura 25 ilustra as classes envolvidas nesta transformação, bem como as classes especializadas para

cada SGBD quando se faz necessário otimizar consultas para o banco de dados configurado.

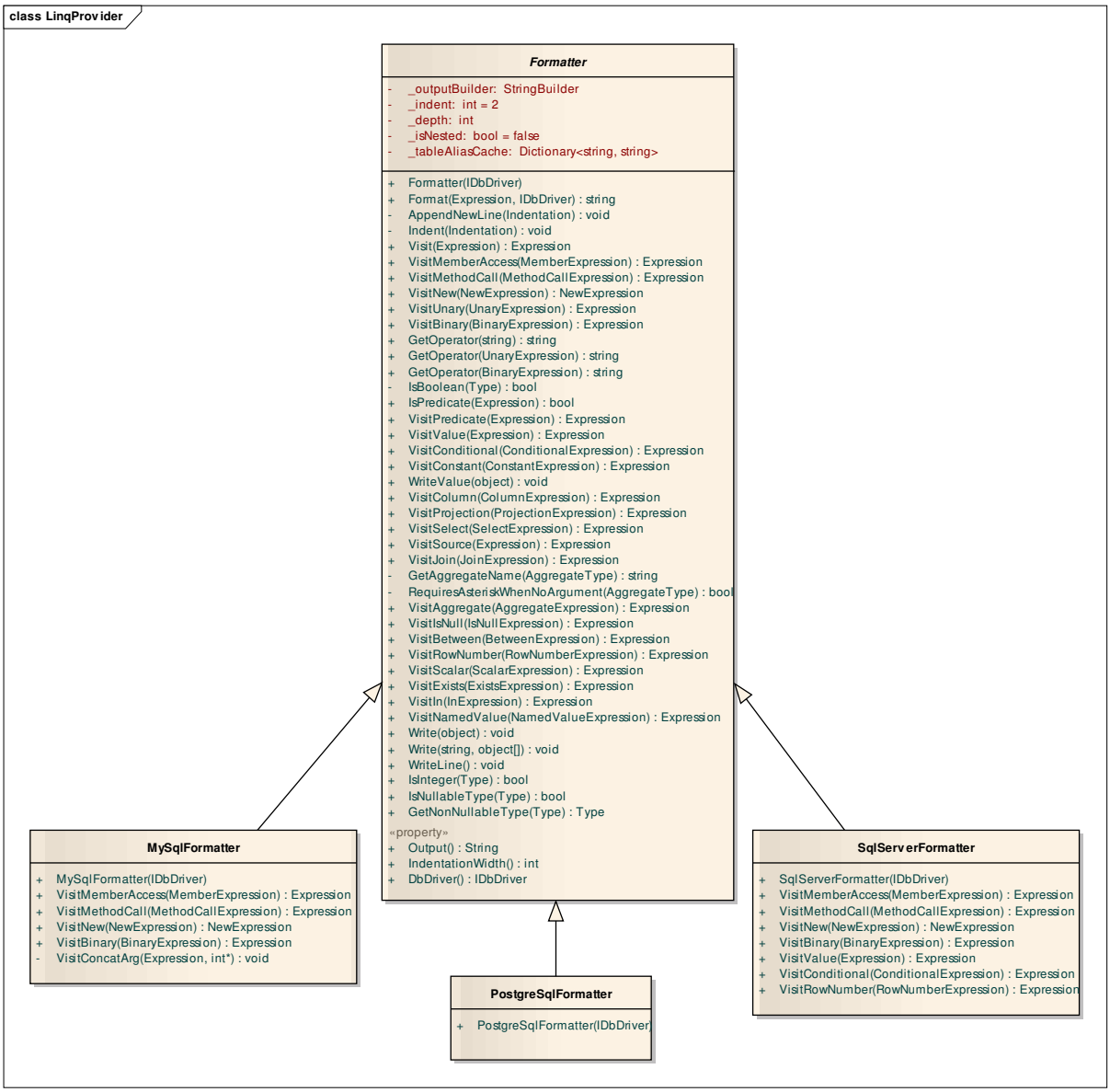


Figura 25 – Transformação de comandos LINQ em comandos SQL

3.3.2 Operacionalidade da implementação (estudo de caso)

A fim de demonstrar as principais funcionalidades implementadas no protótipo, toma-se como exemplo um estudo de caso de uma aplicação construída para simular a operacionalidade de uma Agência de Turismo fictícia. Onde o principal objetivo é o cadastro e controle de reservas para os clientes. Importante observar que as regras de negócio de uma agência de turismo não são o foco deste estudo de caso, servindo apenas para demonstrar a forma de utilização do *framework* construído.

Para a construção do estudo de caso, foi utilizado um *template* gratuito de Hyper Text Markup Language (HTML) denominado Travel Agency WebSite Template (TEM, 2009). O banco de dados utilizado foi o MySQL Server 5.1 com programação em C# e ASP.NET ambos partes integrantes do .NET Framework 3.5 Service Pack 1.

A figura 26 exibe o modelo entidade relacionamento utilizado pelo estudo de caso.

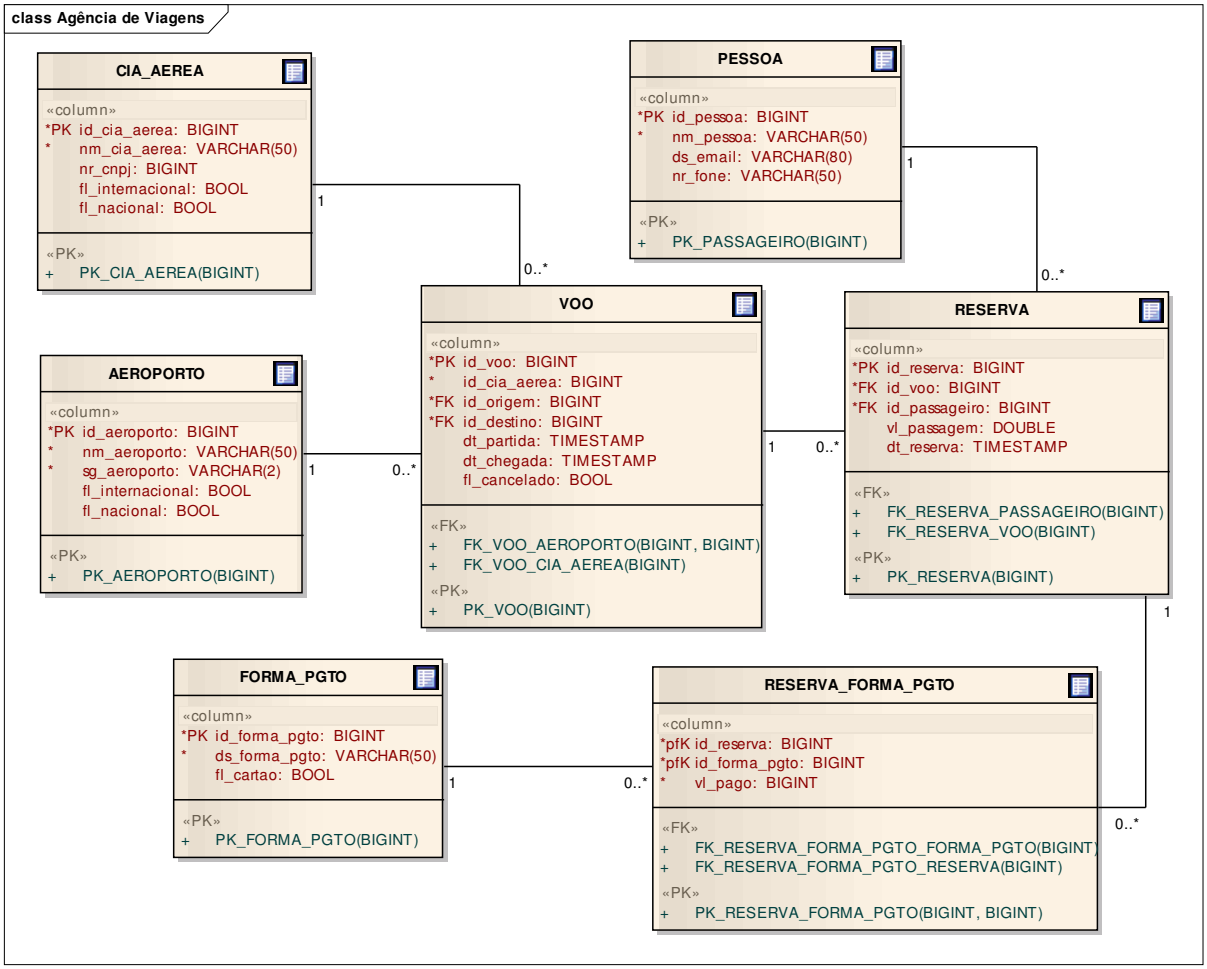


Figura 26 – Modelagem entidade relacionamento do estudo de caso

A figura 27 demonstra o diagrama de classes modelado para a agência de turismo, modelo esse que será utilizado para realizar o mapeamento da classe para com a tabela no banco de dados.

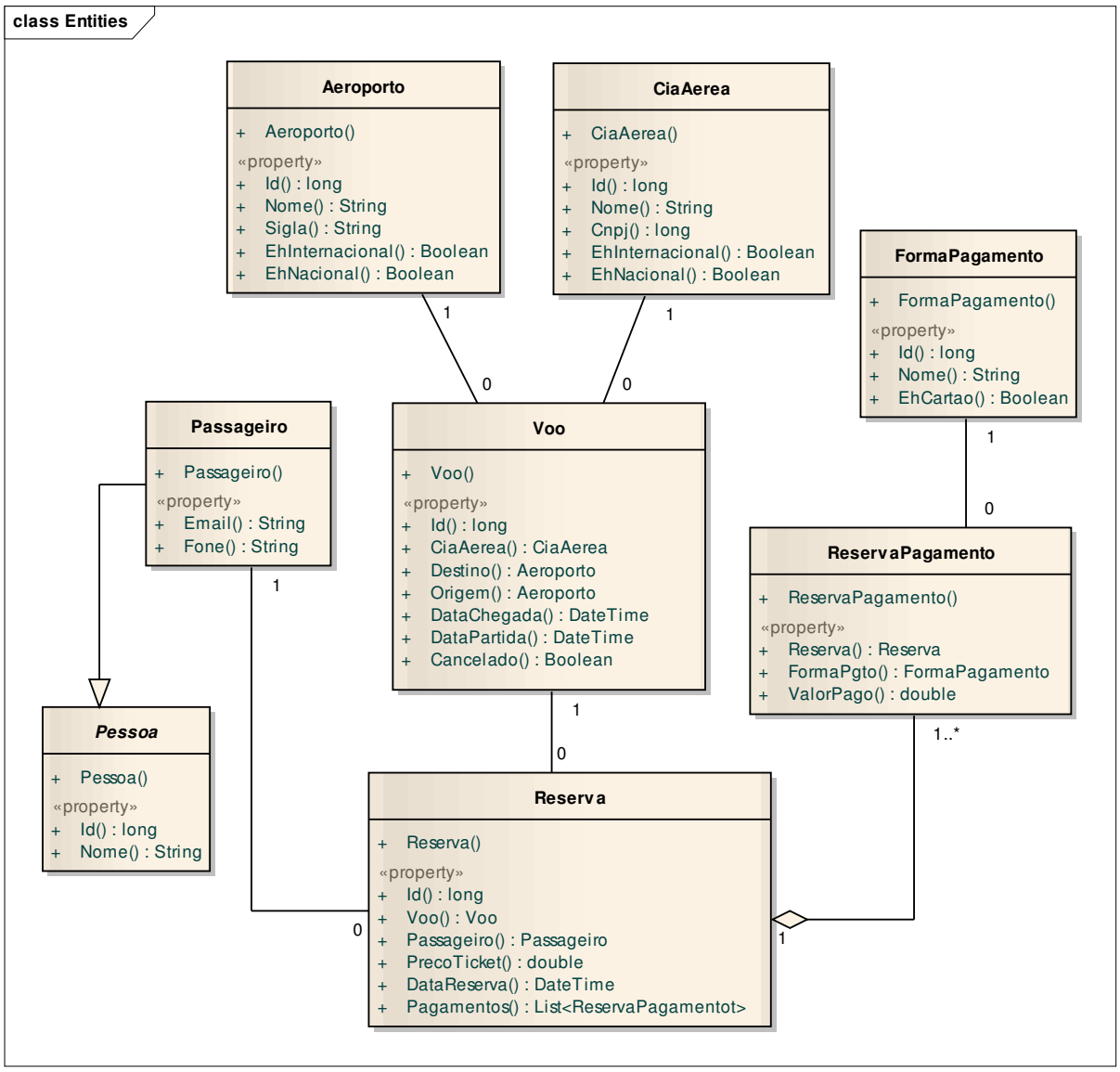


Figura 27 – Diagrama de classes do estudo de caso

A figura 28 exibe a utilização dos atributos customizados de mapeamento disponibilizados pelo protótipo, `TableMap` e `ColumnMap`, para mapear a classe abstrata `Pessoa` e sua especialização, a classe `Passageiro`, para a tabela PESSOA correspondente no banco de dados, percebe-se que o método de mapeamento de herança é através de uma única tabela para todas as classes. Pode-se observar que o nome da tabela PESSOA não necessita ser informado, pois o protótipo utiliza o nome da própria classe para localizar a tabela no banco de dados.

```
[TableMap]
public abstract class Pessoa
{
    [ColumnMap(ColumnName="ID_PESSOA", IsAutoIncrement=true, IsPrimaryKey=true, AllowNull=false)]
    public long Id { get; set; }

    [ColumnMap(ColumnName="NM_PESSOA", AllowNull=false)]
    public String Nome { get; set; }

    public Pessoa()
    {
    }
}

[TableMap(TableName = "PESSOA")]
public class Passageiro : Pessoa
{
    [ColumnMap(ColumnName = "DS_EMAIL")]
    public String Email { get; set; }

    [ColumnMap(ColumnName = "NR_FONE")]
    public String Fone { get; set; }

    public Passageiro()
    {
    }

    public override string ToString()
    {
        return this.Nome;
    }
}
```

Figura 28 – Mapeamento de herança entre `Pessoa` e `Passageiro`

A figura 29 demonstra o mapeamento da classe `CiaAerea` para a tabela CIA_AEREA.

```
[TableMap(TableName = "CIA_AEREA")]
public class CiaAerea
{
    [ColumnMap(ColumnName = "ID_CIA_AEREA", AllowNull = false, IsAutoIncrement = true, IsPrimaryKey = true)]
    public long Id { get; set; }

    [ColumnMap(ColumnName = "NM_CIA_AEREA", AllowNull = false)]
    public String Nome { get; set; }

    [ColumnMap(ColumnName = "NR_CNPJ")]
    public long Cnpj { get; set; }

    [ColumnMap(ColumnName = "FL_INTERNACIONAL")]
    public Boolean EhInternacional { get; set; }

    [ColumnMap(ColumnName = "FL_NACIONAL")]
    public Boolean EhNacional { get; set; }

    public CiaAerea()
    {

    }

    public override string ToString()
    {
        return this.Nome;
    }
}
```

Figura 29 – Mapeamento da classe `CiaAerea`

A figura 30 exibe o mapeamento da classe `Aeroporto` para a tabela AEROPORTO.

```
[TableMap]
public class Aeroporto
{
    [ColumnMap(ColumnName = "ID_AEROPORTO", AllowNull = false, IsAutoIncrement = true, IsPrimaryKey = true)]
    public long Id { get; set; }

    [ColumnMap(ColumnName = "NM_AEROPORTO", AllowNull = false)]
    public String Nome { get; set; }

    [ColumnMap(ColumnName = "SG_AEROPORTO", AllowNull = false)]
    public String Sigla { get; set; }

    [ColumnMap(ColumnName = "FL_INTERNACIONAL")]
    public Boolean EhInternacional { get; set; }

    [ColumnMap(ColumnName = "FL_NACIONAL")]
    public Boolean EhNacional { get; set; }

    public Aeroporto()
    {

    }

    public override string ToString()
    {
        return this.Sigla;
    }
}
```

Figura 30 – Mapeamento da classe `Aeroporto`

Com relação à figura 31, demonstra-se o mapeamento da classe `Voo` para com a tabela VOO, observa-se que as propriedades *CompanhiaAerea*, *Destino* e *Origem* possuem relacionamento definido pelo tipo da propriedade, não necessitando informar que se trata de um relacionamento pois o protótipo descobre o relacionamento de forma automática.

```
[TableMap]
public class Voo
{
    [ColumnMap(ColumnName = "ID_VOO", AllowNull = false, IsAutoIncrement = true, IsPrimaryKey = true)]
    public long Id { get; set; }

    [ColumnMap(ColumnName = "ID_CIA_AEREA", AllowNull = false)]
    public CiaAerea CompanhiaAerea { get; set; }

    [ColumnMap(ColumnName = "ID_DESTINO", AllowNull = false)]
    public Aeroporto Destino { get; set; }

    [ColumnMap(ColumnName = "ID_ORIGEM", AllowNull = false)]
    public Aeroporto Origem { get; set; }

    [ColumnMap(ColumnName = "DT_CHEGADA")]
    public DateTime DataChegada { get; set; }

    [ColumnMap(ColumnName = "DT_PARTIDA")]
    public DateTime DataPartida { get; set; }

    [ColumnMap(ColumnName = "FL_CANCELADO")]
    public Boolean Cancelado { get; set; }

    public Voo()
    {
    }

    public override string ToString()
    {
        return this.Id.ToString();
    }
}
```

Figura 31 – Mapeamento da classe `Voo`

A fim de descrever as formas de pagamento de uma reserva, a figura 32 exibe o mapeamento aplicado na classe `FormaPagamento` para com a tabela FORMA_PGTO.

```
[TableMap(TableName="FORMA_PGTO")]
public class FormaPagamento
{
    [ColumnMap(ColumnName = "ID_FORMA_PGTO", AllowNull = false, IsAutoIncrement = true, IsPrimaryKey = true)]
    public long Id { get; set; }

    [ColumnMap(ColumnName = "DS_FORMA_PGTO", AllowNull = false)]
    public String Nome { get; set; }

    [ColumnMap(ColumnName = "FL_CARTAO")]
    public Boolean EhCartao { get; set; }

    public FormaPagamento()
    {

    }
}
```

Figura 32 – Mapeamento da classe `FormaPagamento`

Conforme demonstrado na figura 26, uma reserva pode apresentar mais de uma forma de pagamento (muitos para muitos), para isso, necessita-se criar uma classe que componha as várias formas de pagamento com a reserva, esta situação é visualizada com o mapeamento da classe `ReservaPagamento` exibida na figura 33.

```
[TableMap(TableName="RESERVA_FORMA_PGTO")]
public class ReservaPagamento
{
    [ColumnMap(ColumnName="ID_RESERVA", AllowNull = false, IsPrimaryKey = true)]
    public Reserva Reserva { get; set; }

    [ColumnMap(ColumnName = "ID_FORMA_PGTO", AllowNull = false, IsPrimaryKey = true)]
    public FormaPagamento Payment { get; set; }

    [ColumnMap(ColumnName = "VL_PAGO", AllowNull = false)]
    public double ValorPago { get; set; }

    public ReservaPagamento()
    {

    }
}
```

Figura 33 – Mapeamento da classe `ReservaPagamento`

É possível observar na figura 33 que a classe possui uma chave primária composta pelas propriedades *Reserva* e *Pagamento* nas quais representam suas classes mapeadas.

Finalizando o mapeamento das classes, a figura 34 apresenta o mapeamento da classe `Reserva`, na qual possui relacionamentos simples com as classes `Voo` e `Passageiro`, e um relacionamento de muitos-para-muitos com a classe `ReservaPagamento` na qual o tipo de retorno da propriedade `Pagamentos` é uma lista de objetos da classe `ReservaPagamento`.

```
[TableMap]
public class Reserva
{
    [ColumnMap(ColumnName="ID_RESERVA", AllowNull=false, IsAutoIncrement=true, IsPrimaryKey=true)]
    public long Id { get; set; }

    [ColumnMap(ColumnName = "ID_VOO", AllowNull = false)]
    public Voo Voo { get; set; }

    [ColumnMap(ColumnName = "ID_PASSAGEIRO", AllowNull = false)]
    public Passageiro Passageiro { get; set; }

    [ColumnMap(ColumnName = "VL_PASSAGEM")]
    public double PrecoTicket { get; set; }

    [ColumnMap(ColumnName = "DT_RESERVA")]
    public DateTime DataReserva { get; set; }

    [ColumnMap(Relationship = RelationshipType.ManyToMany)]
    public List<ReservaPagamento> Pagamentos { get; set; }

    public Reserva()
    {

    }
}
```

Figura 34 – Mapeamento da classe `Reserva`

Para que o protótipo crie a conexão com o banco de dados é necessário informar qual SGBD será utilizado bem como as informações de necessárias para localização do banco de dados através de uma *connection string*. A figura 35 demonstra a inicialização e configuração do protótipo com as informações de conexão e o tipo do SGBD utilizado, observa-se que esta configuração é necessária apenas uma vez no ciclo de vida do protótipo.

```
public static class GerenciadorEntidades
{
    public static void ConfigurarOrm()
    {
        var connString = ConfigurationManager.ConnectionStrings["Default"];
        var provider = (Boufleuhr.DataAccessLayer.DbProviderType)Enum.Parse(
            typeof(Boufleuhr.DataAccessLayer.DbProviderType),
            connString.ProviderName,
            true);

        BoufleuhrOrm.Configure(connString.ConnectionString, provider);
    }

    static GerenciadorEntidades()
    {
        ConfigurarOrm();
    }
}
```

Figura 35 – Inicialização e configuração do protótipo

A tela inicial exibe a lista de reservas efetuadas pelo sistema bem como um filtro na parte superior para realizar uma busca pelas reservas. Nesta mesma tela o usuário pode incluir uma nova reserva através de um formulário localizado no canto inferior esquerdo, conforme demonstra a figura 36.

Figura 36 – Tela inicial do sistema de agência de turismo

Para o cadastramento de uma nova reserva após os dados inseridos no formulário, o sistema dispara o comando de persistência através das funcionalidades implementadas no protótipo, conforme exibidas pela figura 37.

```
public static Reserva SalvarReserva(
    long id,
    long voo,
    long passageiro,
    double precoTicket,
    List<ReservaPagamento> pagamentos)
{
    Reserva reserva = new Reserva()
    {
        Voo = DataRepository.Get<Voo>(voo),
        Passageiro = DataRepository.Get<Passageiro>(passageiro),
        DataReserva = DateTime.Now,
        PrecoTicket = precoTicket,
        Pagamentos = pagamentos
    };

    if (id > 0)
        reserva.Id = id;

    DataRepository repositorio = new DataRepository();
    repositorio.SaveOnAccept(reserva);
    repositorio.Accept();

    return reserva;
}
```

Figura 37 – Persistência de um objeto da classe `Reserva` utilizando o protótipo

A figura 38 apresenta a tela de cadastramento e manutenção dos vôos que estão inseridos no sistema.

Figura 38 – Cadastro e manutenção dos vôos no sistema

Como visto na figura 38, quando os vôos são listados existe a possibilidade de exclusão do mesmo. O trecho de código apresentado na figura 39 demonstra a exclusão do vôo pela chave primária da classe, recuperada através da ação do usuário em um vôo específico.

```
[DataObjectMethod(DataObjectMethodType.Delete)]
public static void ExcluirVoo(long id)
{
    if (id == 0)
        return;

    Voo voo = new Voo()
    {
        Id = id,
    };

    DataRepository repositorio = new DataRepository();
    repositorio.DeleteOnAccept(repositorio);
    repositorio.Accept();
}
```

Figura 39 – Exclusão de um vôo através da chave primária do objeto

Na tela de cadastro e manutenção dos passageiros inseridos no sistema, pode-se realizar uma pesquisa pelo código e/ou nome de um passageiro. A figura 40 demonstra uma pesquisa por um passageiro cujo nome contenha a palavra “THIAGO”.

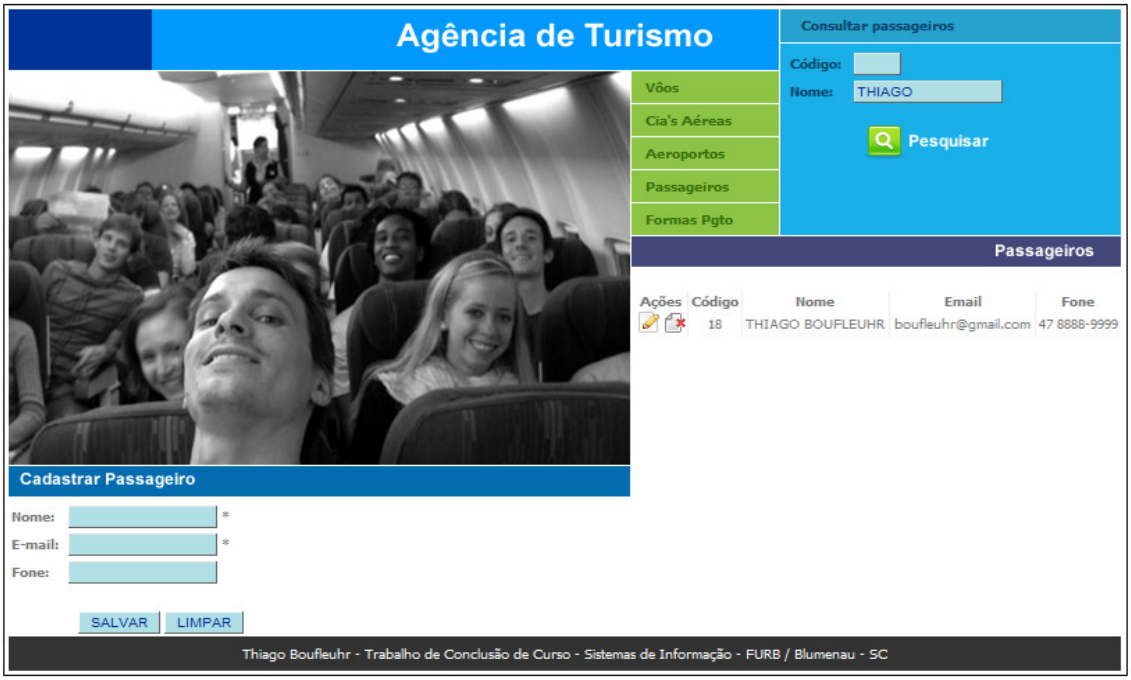


Figura 40 – Pesquisa por um passageiro

O trecho do código que é responsável pela pesquisa do passageiro é exibido na figura 41, nota-se que o código faz uso da tecnologia LINQ utilizada pelo protótipo.

```
[DataObjectMethod(DataObjectMethodType.Select)]
public static List<Passageiro> GetPassageiroEntidade(Int64 Id, String Nome)
{
    var passageiroRepositorio = DataRepository.GetRepository<Passageiro>();

    var where = passageiroRepositorio.Where(p => p.Id != null);
    if (Id != 0)
        where = where.Where(a => a.Id == Id);

    if (!String.IsNullOrEmpty(Nome))
        where = where.Where(a => a.Nome.Contains(Nome.ToUpper()));

    return where.ToList<Passageiro>();
}
```

Figura 41 – Código de pesquisa do passageiro utilizando LINQ

Dessa forma o estudo de caso de uma agência de turismo hipotética foi plenamente construído utilizando as funcionalidades originais do protótipo na camada de persistência dos dados.

3.4 RESULTADOS E DISCUSSÃO

Os recursos de mapeamento implementados no protótipo são de simples utilização e focados na clareza de leitura do código fonte, priorizando a abstração máxima durante a tarefa de mapear as classes para tabelas no banco de dados. Dessa forma, o ciclo de desenvolvimento torna-se orientado ao domínio do problema, reduzindo a necessidade de implementação local de recursos tecnológicos para problemas já conhecidos, como é o caso do acesso a dados e persistência de objetos em um sistema computacional.

A forma como foram projetados os recursos de mapeamento, no caso, utilizando atributos customizados que é parte integrante da linguagem C#, tornou-se uma maneira atrativa para realizar tal tarefa, pois em contraponto ao trabalho correlato NHibernate (MAULO, 2008), não necessita de arquivos adicionais de mapeamento que muitas vezes são incômodos durante o processo de desenvolvimento. Por exemplo, a manutenção de arquivos externos de mapeamento onde deve-se garantir a compatibilidade entre os atributos da classe as colunas da tabela, uma vez que são interpretados apenas em tempo de execução, postergando possíveis erros de mapeamento, entre outros.

A tarefa de realizar o mapeamento de chaves compostas muitas vezes torna-se incômoda ao desenvolvedor, pois as ferramentas citadas nos trabalhos correlatos utilizam de uma complexidade desnecessária do ponto de vista do programador. Por exemplo, no NHibernate existe a necessidade de criar objetos extras para representar tais chaves. Face a isto, o protótipo possui um grande diferencial neste quesito. Conforme exemplificado na figura 32 do estudo de caso, resolve-se este impasse marcando a propriedade *IsPrimaryKey* igual a *true* no atributo de mapeamento de colunas, sem restrição de limites para a quantidade de chaves envolvidas.

Um ponto interessante de se observar quanto ao mapeamento de relacionamentos do protótipo e a forma como tratam esta questão os trabalhos correlatos NHibernate (MAULO, 2008) e o projeto SubSonic (CONERY, 2009) é a não necessidade de explicitar o mapeamento em nenhum ponto, visto que o protótipo, ao contrário dos trabalhos citados, descobre automaticamente o relacionamento pelo tipo da propriedade garantindo a navegabilidade entre objetos de forma transparente para o usuário desenvolvedor. A informação de relacionamento do mapeamento existe apenas para melhorar a legibilidade do código fonte perante o time de desenvolvimento.

4 CONCLUSÕES

Segundo Lhotka (2009), a construção de uma ferramenta que realize o mapeamento lógico de estruturas de objetos convencionais para estruturas relacionais, deve ser voltada para as características da plataforma em que a mesma será utilizada, a fim de obter os melhores resultados em termos de desempenho e portabilidade, devido às diversas maneiras na qual o mesmo problema pode ser resolvido.

Com base nos pontos levantados acima, observou-se a que a plataforma .NET necessitava de um mecanismo reutilizável de acesso e manipulação de dados relacionais mapeados através de objetos, e que principalmente não transpareça para o desenvolvedor qual Sistema Gerenciador de Banco de Dados (SGBD) esteja-se utilizando.

Embora existam diversas tecnologias que solucionam o problema de mapeamento objeto-relacional em C# no mercado, muitas delas não seguem o padrão de desenvolvimento recomendado pela Microsoft para a plataforma .NET (MSDN, 2007), o que dificulta a compreensão das funcionalidades e dos conceitos particulares da ferramenta por parte dos desenvolvedores acostumados com a plataforma.

Dessa forma, surgiu a necessidade de desenvolver uma ferramenta genérica de mapeamento de objetos para o modelo relacional ou ORM para a plataforma .NET visando auxiliar na persistência e recuperação de objetos escritos na linguagem C# de forma transparente para o desenvolvedor. Assim sendo, torna o desenvolvimento de qualquer solução, que faça uso de banco de dados, mais simples de ser utilizada, pois conta com o suporte integrado ao LINQ no qual facilita a construção de consultas utilizando puramente objetos, atendendo completamente a um dos objetivos deste protótipo.

Dentre um dos principais objetivos deste protótipo, eliminou-se completamente a necessidade de mapear as classes escritas em C# utilizando qualquer tipo de arquivo secundário como, por exemplo, arquivos XML. O mapeamento é realizado decorando-se a classe e suas propriedades com atributos customizados disponibilizados pelo protótipo, atendendo a outro objetivo deste protótipo.

Para o objetivo de simplificar o uso de chaves primárias compostas, foi disponibilizado na classe `ColumMap` uma propriedade denominada *IsPrimaryKey*, na qual pode ser utilizada em quaisquer propriedades que reflitam tal comportamento na tabela no banco de dados.

Por fim, o objetivo de simplificar o uso de relacionamentos entre classes mapeadas foi atingido mediante a implementação de um mecanismo que valida o tipo de dados da

propriedade que representa o relacionamento a fim de verificar se o mesmo possui o atributo `TableMap` em seu *metadado*, dessa forma toma-se como parâmetro que o tipo representa um relacionamento.

4.1 EXTENSÕES

O protótipo construído possui as funcionalidades básicas de um mapeador de dados, porém, existem muitos pontos de extensão que se pode observar para acrescentar mais confiabilidade e tornar o mesmo mais aceitável em uma possível utilização comercial.

Para tanto, sugere-se:

a) aprimorar o sistema de *caching* de objetos em memória para evitar *pooling* desnecessário ao banco de dados;
b) geração da Data Definition Language (DDL);
c) implementar a comunicação com outros SGBD's, como por exemplo Oracle, DB2 entre outros;
d) utilizar o padrão *proxy* para retornar objetos mapeados a fim de aperfeiçoar o carregamento tardio (*lazy load*);
e) suportar objetos do tipo Plain Old CLR Objects (POCO) para desenvolvimento em várias camadas;
f) aperfeiçoar o suporte a *multi threading*;
g) construir uma fachada remota para transferência de objetos na rede.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott W. **Análise e Projeto Orientados a Objeto:** seu guia para desenvolver sistemas robustos com tecnologia de objetos. Rio de Janeiro: Infobook, 1998.

AMBLER, Scott W. **Mapping Objects to Relational Databases: O/R Mapping In Detail.** Toronto, [2002?]. Disponível em < http://www.agiledata.org/essays/mappingObjects.html>. Acesso em: 12 set 2009.

ANSI, ISO/IEC 9075-1:2008. New York, NY [2008?]. Disponível em <http://webstore.ansi.org/RecordDetail.aspx?sku=ISO/IEC+9075-1:2008>. Acesso em 10 jan 2010.

CONERY, Rob. **SubSonic.** [S.l.], [2009?]. Disponível em <http://www.subsonicproject.com>. Acesso em: 01 set 2009.

FOWLER, Martin. **Patterns of Enterprise Application Architecture.** Boston, MA: Addison-Wesley, 2002.

GAMMA, Eric; JOHNSON, Ralph; HELM, Richard; VLISSIDES, John M. **Design Patterns: Elements of Reusable Object-Oriented Software.** NJ: Addison-Wesley Professional Computing Series, 1994.

LHOTKA, Rockford. **Expert C# 2008 Business Objects**. New York: Apress, 2009.

MARGUERIE, Fabrice; EICHERT, Steve; WOOLEY, Jim. **LINQ em Ação.** Rio de Janeiro: Editora Ciência Moderna LTDA, 2009.

MAULO, Fabio. **NHibernate for .NET.** [S.l.], [2008?]. Disponível em <https://www.hibernate.org/343.html>. Acesso em: 04 set 2009.

MEHTA, Vijay P. **Pro LINQ Object Relational Mapping with C# 2008**. New York: Apress, 2008.

MSDN. **Microsoft patterns & practices Developer Center.** [S.l.], [2007?]. Disponível em <http://msdn.microsoft.com/en-us/practices/bb190357.aspx#get>. Acesso em: 16 set 2009.

MSDN. **Visual C# Developer Center.** [S.l], [2008?]. Disponível em < http://msdn.microsoft.com/en-us/library/bb655883.aspx>. Acesso em: 06 de março de 2010.

SILVA, Anderson G. **Mapeamento Objeto-Relacional.** 2005. 72 f. Monografia (Graduação em Ciências da Computação) – Curso de Graduação em Ciências da Computação, FAJ, Jaguariúna.

TEM, 8. **Custom Web Templates.** [S.l.], [2009?]. Disponível em <http://www.8tem.com/travelagency_website_template.html>. Acesso em: 10 mar 2010.

TROELSEN, Andrew. **Pro C# 2008 and the .NET 3.5 Platform.** 4th ed**.** New York: Apress, 2007.

WARREN, Matt. **LINQ: Building na IQueryable Provider.** [S.l.], [2007]. Disponível em <http://blogs.msdn.com/mattwar/archive/2007/07/30/linq-building-an-iqueryable-provider-part-i.aspx >. Acesso em: 02 fev 2010.

APÊNDICE A – Detalhamento dos casos de uso

No quadro 7 é apresentado o caso de uso “UC01 - Mapear classes .NET”.

Nome do Caso de Uso	UC01 - Mapear classes .NET
Descrição	Desenvolvedor realiza o mapeamento das classes escritas em linguagem *C#* refletindo a estrutura física de uma tabela no banco de dados.
Ator	Desenvolvedor
Pré-condição	A classe deve estar previamente escrita. A *Dynamic Link Library* (DLL) do protótipo deve estar referenciada no projeto.
Fluxo principal	1. Desenvolvedor realiza o mapeamento da(s) classe(s) através de *custom attributes*; 2. Desenvolvedor realiza o mapeamento das propriedades da classe através de atributos customizados de propriedade; 3. Desenvolvedor realiza o mapeamento dos relacionamentos entre as classes; 4. *Framework* realiza a validação do mapeamento através de reflexão computacional no momento da compilação.
Fluxo alternativo (a)	1. Mapeamento realizado sem definição de chave primária; 2. Exceção é levantada avisando que é necessário definir uma propriedade como chave primária.
Pós-condição	Classe mapeada para realizar operações de persistência.

Quadro 7 – Descrição do caso de uso UC01

No quadro 8 é apresentado caso de uso “UC02 - Configurar SGBD utilizado”.

Nome do Caso de Uso	UC02 - Configurar SGBD utilizado
Descrição	Desenvolvedor configura o protótipo informando uma string de conexão juntamente com o tipo de banco de dados utilizado (SqlServer, MySql ou PostgreSql).
Ator	Desenvolvedor
Pré-condição	O banco de dados deve estar configurado e acessível para o protótipo.
Fluxo principal	1. Desenvolvedor informa uma *string* de conexão para o banco de dados; 2. Desenvolvedor informa o tipo do banco de dados; 3. *Framework* inicializa o protótipo com as informações fornecidas.
Pós-condição	Desenvolvedor configurou o acesso ao banco de dados.

Quadro 8 – Descrição do caso de uso UC02

No quadro 9 é apresentado o caso de uso “UC03 - Inserir objetos no banco de dados”.

Nome do Caso de Uso	UC03 - Inserir objetos no banco de dados
Descrição	Desenvolvedor utiliza os objetos da classe mapeada para inserir um novo registro no banco de dados.
Ator	Desenvolvedor

Pré-condição	O acesso ao banco de dados deve ter sido configurado.
Fluxo principal	1. Desenvolvedor cria uma instância da classe mapeada, preenchendo as informações necessárias do objeto; 2. Desenvolvedor cria uma instância do repositório de dados; 3. Desenvolvedor adiciona o objeto ao repositório de dados marcando-o para inserção; 4. *Framework* recupera as informações de mapeamento da classe criando o vínculo com a estrutura relacional; 5. Desenvolvedor confirma a inserção do objeto no repositório de dados; 6. *Framework* persiste as informações contidas no objeto diretamente no banco de dados.
Pós-condição	Desenvolvedor inseriu um novo registro no banco de dados.

Quadro 9 – Descrição do caso de uso UC03

No quadro 10 é apresentado o caso de uso "UC04 - Alterar objetos existentes no banco de dados".

Nome do Caso de Uso	UC04 - Alterar objetos existentes no banco de dados
Descrição	Desenvolvedor utiliza os objetos da classe mapeada para alterar suas propriedades e persistir as alterações no banco de dados.
Ator	Desenvolvedor
Pré-condição	Os objetos que serão alterados devem possuir a sua classe previamente mapeada. O acesso ao banco de dados deve ter sido configurado. O objeto deve estar preenchido com a chave primária do registro.
Fluxo principal	1. Desenvolvedor cria uma instância da classe mapeada, preenchendo as informações que deseja alterar juntamente com a chave primária do objeto; 2. Desenvolvedor cria uma instância do repositório de dados; 3. Desenvolvedor adiciona o objeto ao repositório de dados marcando-o para alteração; 4. *Framework* recupera as informações de mapeamento da classe criando o vínculo com a estrutura relacional; 5. Desenvolvedor confirma a alteração do objeto no repositório de dados; 6. Sistema persiste as informações contidas no objeto diretamente no banco de dados.
Pós-condição	Desenvolvedor alterou um registro existente no banco de dados.

Quadro 10 – Descrição do caso de uso UC04

No quadro 11 é apresentado o caso de uso "UC05 - Excluir objetos no banco de dados".

Nome do Caso de Uso	UC05 - Excluir objetos no banco de dados
Descrição	Desenvolvedor utiliza os objetos da classe mapeada para excluir um objeto através

	da chave primária do mesmo.
Ator	Desenvolvedor
Pré-condição	Os objetos que serão excluídos devem possuir a sua classe previamente mapeada. O acesso ao banco de dados deve ter sido configurado. O objeto deve estar preenchido com a chave primária do registro para realizar a exclusão.
Fluxo principal	1. Desenvolvedor cria uma instância da classe mapeada, preenchendo-o com a chave primária do registro que deseja excluir; 2. Desenvolvedor cria uma instância do repositório de dados; 3. Desenvolvedor adiciona o objeto ao repositório de dados marcando-o para exclusão; 4. *Framework* recupera as informações de mapeamento da classe criando o vínculo com a estrutura relacional; 5. Desenvolvedor confirma a exclusão do objeto no repositório de dados; 6. *Framework* realiza a exclusão do registro conforme chave primária, diretamente no banco de dados.
Pós-condição	Desenvolvedor exclui um registro no banco de dados.

Quadro 11 – Descrição do caso de uso UC05

No quadro 12 é apresentado o caso de uso "UC06 - Consultar objetos utilizando LINQ".

Nome do Caso de Uso	UC06 – Consultar objetos utilizando LINQ
Descrição	Desenvolvedor utiliza um objeto da classe mapeada para realizar consultas ao banco de dados.
Ator	Desenvolvedor
Pré-condição	O acesso ao banco de dados deve ter sido configurado. A classe do objeto deve estar mapeada.
Fluxo principal	1. Desenvolvedor recupera uma referência para a classe mapeada através do repositório de dados; 2. Desenvolvedor escreve a consulta utilizando a sintaxe LINQ; 3. *Framework* transforma a consulta LINQ em consulta SQL; 4. *Framework* executa a consulta SQL no banco de dados configurado; 5. *Framework* recupera constrói os objetos da classe mapeada com o resultado da consulta; 6. *Framework* retorna uma lista contendo os objetos da classe mapeada.
Pós-condição	Desenvolvedor consultou objetos no banco de dados.

Quadro 12 – Descrição do caso de uso UC06

No quadro 13 é apresentado o caso de uso "UC07 - Consultar registros através da linguagem SQL".

Nome do Caso de Uso	UC07 – Consultar registros através da linguagem SQL
Descrição	Desenvolvedor utiliza uma classe disponibilizada pelo *framework* para realizar consultas/operações utilizando puramente a linguagem SQL.
Ator	Desenvolvedor
Pré-condição	O acesso ao banco de dados deve ter sido configurado.
Fluxo principal	1. Desenvolvedor realiza a montagem do comando SQL; 2. Desenvolvedor utiliza o método da classe `DataRepository` passando o comando montado; 3. *Framework* valida o comando; 4. *Framework* abre a conexão com o banco de dados e executa a consulta; 5. *Framework* retorna o resultado da consulta.
Pós-condição	Desenvolvedor consultou o banco de dados utilizando um comando SQL.

Quadro 13 – Descrição do caso de uso UC07