



**FUNDAÇÃO EDSON QUEIROZ**  
**UNIVERSIDADE DE FORTALEZA – UNIFOR**  
**CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT**  
**Curso de Ciência da Computação**

**IMPLEMENTAÇÃO DE UM FRAMEWORK PARA MAPEAMENTO  
OBJETO-RELACIONAL (ORM) NA LINGUAGEM C#**

Yuri Marcel Holanda Barreto  
Matrícula nº 0720999/1

Fortaleza/CE  
2014

YURI MARCEL HOLANDA BARRETO

**IMPLEMENTAÇÃO DE UM FRAMEWORK PARA MAPEAMENTO  
OBJETO-RELACIONAL (ORM) NA LINGUAGEM C#**

Projeto de pesquisa a ser apresentado e submetido à avaliação para elaboração de Trabalho de Conclusão de Curso – TCC do Curso de Ciência da Computação do Centro de Ciências Tecnológicas da Universidade de Fortaleza, sob a orientação do Prof. Maicol Magalhães Rodrigues, M.Sc.

Fortaleza/CE  
2014

YURI MARCEL HOLANDA BARRETO

**IMPLEMENTAÇÃO DE UM FRAMEWORK PARA MAPEAMENTO  
OBJETO-RELACIONAL (ORM) NA LINGUAGEM C#**

Projeto de pesquisa a ser apresentado e submetido à avaliação para elaboração de Trabalho de Conclusão de Curso – TCC do Curso de Ciência da Computação do Centro de Ciências Tecnológicas da Universidade de Fortaleza, sob a orientação do Prof. Maikol Magalhães Rodrigues, M.Sc.

Fortaleza (CE), 07 de dezembro de 2014.

**PARECER:** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Maikol Magalhães Rodrigues  
**Prof. Orientador da Unifor**

**BANCA EXAMINADORA**

Maikol Magalhães Rodrigues  
**Prof. Examinador da Unifor**

Rafael Garcia Barbosa  
**Prof. Examinador da Unifor**

## RESUMO

Nos últimos anos o desenvolvimento de sistemas adotou uma prática que visa a utilização do paradigma de orientação a objetos e a adoção de banco de dados relacional para armazenamento de dados. Devido aos paradigmas orientado a objeto e relacional possuírem estruturas e características distintas, a comunicação entre os dois modelos deve ser adaptada, necessitando mapear as estruturas de cada paradigma para poder realizar a comunicação. Este trabalho visa apresentar os conceitos do modelo orientado a objeto e relacional, como também apresentar a implementação de um *framework* que possa resolver os problemas de impedância existentes na utilização dos dois paradigmas em um projeto. Também serão apresentados as características e implementação das soluções dos problemas de impedância encontradas na utilização do *framework* em um projeto que utiliza o modelo orientado a objeto e um banco de dados relacional.

**Palavra-chave:** orientação a objeto; orm; banco de dados relacional; mapeamento objeto relacional; problemas de impedância; framework;

## ABSTRACT

In recent years the development of systems adopted a practice that aims to use the object-oriented paradigm and the adoption of relational database for data storage. Due to the object-oriented paradigms and relational structures and their distinct characteristics, communication between the two models should be adapted, requiring mapping the structures of each paradigm in order to perform communication. This paper presents the concepts of object-oriented relational model, as well as present the implementation of a framework that can solve the impedance issues in the use of both paradigms in a project. The features and implementation of solutions of impedance problems encountered in using the framework in a project that uses the object-oriented model and a relational database will also be presented.

**Keywords:** object orientation; orm; relational database; object relational mapping; impedance problems; framework;

## LISTA DE FIGURAS

Figura 1: Três Objetos da Classe “Pessoa”.....	12
Figura 2: Três Objetos da Classe “Aluno” que herdam da Classe “Pessoa”.....	12
Figura 3: Exemplo de Polimorfismo no Método “EmitirSom” da Classe Animal.....	13
Figura 4: Representação de Conjuntos e Relacionamentos (Fonte: STAJANO, 1998).....	14
Figura 5: Instrução <i>SQL</i> e seu Resultado. (Fonte: KLINE, 2008).....	17
Figura 6: Entidade com <i>Shadow Information</i> .....	19
Figura 7: Arquitetura do <i>NHibernate</i> (Fonte: Definição de arquitetura do <i>NHibernate</i> ).....	21
Figura 8: Diferenças entre <i>XML Mapping</i> e <i>Fluent NHibernate</i> .....	22
Figura 9: Entidade de Negócio Produtos.....	29
Figura 10: Abordagens para Mapeamento de Herança.....	31
Figura 11: Associação e Especialização “um-para-um”.....	33
Figura 12: Associação “um-para-muitos” e “um-para-muitos” recursiva.....	34
Figura 13: Associação “muitos-para-muitos”.....	35
Figura 14: Operações <i>CRUD</i> .....	39
Figura 15: Operação Buscar com parâmetros. ....	40
Figura 16: Referências do <i>framework</i> no Projeto. ....	42
Figura 17: Arquivo de Configuração do Projeto.....	43
Figura 18: Diagrama <i>UML</i> de Caso de Uso. ....	45
Figura 19: Diagrama de Classes e Modelo Entidade Relacionamento. ....	46
Figura 20: Classes <i>POCO</i> .....	47
Figura 21: Tela de Cadastro.....	48
Figura 22: Código do Método Gravar.....	49
Figura 23: Tela de Busca Simples.....	51
Figura 24: Código do Método Busca Simples.....	51
Figura 25: Tela de Busca Avançada.....	52
Figura 26: Código do Método Busca Avançada.....	53
Figura 27: Código do Método Remover.....	54
Figura 28: Código do Método Editar.....	55
Figura 29: Exemplo do Problema de Granularidade.....	57
Figura 30: Exemplo de Arquivo de Configuração XML.....	59
Figura 31: Modelagem de Classe e de Tabelas aplicando o Conceito de Herança.....	59
Figura 32: Entidade Usuario e seu Mapeamento.....	60
Figura 33: Entidade Usuario já Mapeado.....	61
Figura 34: Mapeamento de Associação “um-para-um”.....	62
Figura 35: Mapeamento “um-para-um” na <i>YLibrary</i> .....	62

## LISTA DE QUADROS

Tabela 1: Requisitos Funcionais..... 44

# SUMÁRIO

INTRODUÇÃO.....	9
<b>1 REFERENCIAL TEÓRICO.....</b>	<b>11</b>
<b>1.1 Conceitos de Orientação a Objeto.....</b>	<b>11</b>
1.1.1 Herança.....	12
1.1.2 Polimorfismo.....	13
1.1.3 Encapsulamento.....	13
<b>1.2 Conceitos de Banco de Dados Relacional.....</b>	<b>14</b>
1.2.1 SQL (Structured Query Language).....	15
1.2.2 Tipos de dados.....	16
<b>1.3 Definição de ORM.....</b>	<b>17</b>
1.3.1 Mapeamento.....	18
<b>1.4 Soluções ORM para linguagem C#.....</b>	<b>20</b>
1.4.1 ADO.NET.....	20
1.4.2 NHibernate.....	21
1.4.3 Arquitetura do NHibernate.....	21
<b>2 YLIBRARY.....</b>	<b>24</b>
<b>2.1 Objetivos.....</b>	<b>24</b>
<b>2.2 Estrutura Interna.....</b>	<b>25</b>
<b>2.3. Mapeamento Objeto-Relacional.....</b>	<b>27</b>
2.3.1 Mapeamento de Herança.....	30
2.3.2 Mapeamento de Associações.....	31
2.3.2.1 Associação “um-para-um”.....	32
2.3.2.2 Associação “um-para-muitos”.....	33
2.3.2.3 Associação “muitos-para-muitos” .....	35
<b>2.4 Reflexão.....</b>	<b>36</b>
<b>2.5 Operações CRUD.....</b>	<b>37</b>
<b>3 ESTUDO DE CASO.....</b>	<b>41</b>
<b>3.1 Preparações do ambiente.....</b>	<b>41</b>
<b>3.2 Especificação.....</b>	<b>44</b>
<b>3.3 Implementação.....</b>	<b>46</b>
<b>3.4 Operacionalidade da Implementação.....</b>	<b>48</b>
3.4.1 Operação Cadastrar.....	48
3.4.2 Operação Buscar.....	50
3.4.3 Operação Remover.....	53
3.4.4 Operação Editar.....	54
<b>4 CONCLUSÃO E CONSIDERAÇÕES FINAIS.....</b>	<b>56</b>
<b>4.1 Problema de Granularidade.....</b>	<b>56</b>
<b>4.2 Problema de Herança.....</b>	<b>58</b>
<b>4.3 Problema de Identidade.....</b>	<b>59</b>
<b>4.4 Problema de Associações.....</b>	<b>61</b>
<b>4.5 Conclusão.....</b>	<b>62</b>
REFERÊNCIAS.....	65



## INTRODUÇÃO

A programação orientada a objetos fornece diversos benefícios no âmbito do desenvolvimento de sistemas. Tais benefícios são identificados como herança, polimorfismo e encapsulamento. A utilização desses conceitos de forma eficaz proporciona um desenvolvimento ágil e confiável de projetos complexos, garantindo, assim, um código bem estruturado e de fácil manutenibilidade.

Com o crescimento da demanda de desenvolvimento de sistemas, tornou-se essencial armazenar informações em um banco de dados. O banco de dados relacional é a solução para armazenamento de dados mais segura e robusta atualmente. A sua utilização, juntamente com a programação orientada a objeto, tornou-se um padrão utilizado pelas empresas e fábricas de *software*.

Apesar da crescente evolução do paradigma de desenvolvimento orientado a objetos, os Sistemas de Gerenciamento de Banco de Dados (*SGBD*) não se atualizaram, continuando a utilizar o modelo relacional ao invés do modelo orientado a objetos. Essa divergência entre os dois modelos é identificada como um problema de impedância. Com esta incompatibilidade, as inúmeras vantagens que o modelo orientado a objetos oferece não são utilizadas no escopo de armazenamento de dados.

Uma solução para suprir essa necessidade é a utilização de *frameworks* de persistências, conhecidos como *frameworks ORM* (*Object-Relational Modeling*), que consistem em técnicas ou abordagens para abstrair o trabalho de persistência de objetos em um banco de dados relacional. Uma característica dessa solução é prover uma camada de persistência que fará o trabalho de tradução de objetos para dados relacionais e vice-versa. Essa tradução permite com que desenvolvedores priorizem a resolução de problemas de negócio, ao invés de problemas de persistência de dados.

Existem várias soluções *ORM* disponíveis no mercado, algumas bem conhecidas e confiáveis. Assim, a implementação de uma nova solução não traz garantias de um projeto viável. Devido a esta incerteza, é preciso analisar as desvantagens da utilização de uma ferramenta já existente, para que a nova ferramenta a ser implementada possua como características principais a solução para desvantagens identificadas.

A implementação de uma solução *ORM* torna-se necessária quando o padrão ou modelo de desenvolvimento já utilizado na empresa é diferente do modelo utilizado pelo *framework*. Essa mudança acarreta em uma reestruturação nos processos da empresa e uma

adequação da equipe ao novo padrão. Algumas outras consequências poderão surgir, tais como a curva de aprendizagem a ser adquirida pela equipe no manuseio e utilização do *framework*, prazos longos nos projetos devido à escassez de documentação ou mão de obra indisponível. Todos esses contratempos devem ser considerados antes de desenvolver uma nova solução.

Este trabalho tem como principal objetivo apresentar uma solução ágil e confiável de um *framework* para mapeamento de objetos do modelo orientado a objetos para o modelo relacional. Serão apresentados os conceitos do paradigma relacional e de orientação a objetos, assim como a incompatibilidade entre os dois modelos, sendo possível analisar efetivamente a solução implementada. A solução que será implementada é a *YLibrary*. Serão reveladas suas principais características, estrutura e implementação. Além disso, será realizado um comparativo descritivo entre as soluções de impedância aplicadas no *framework YLibrary* e no *NHibernate*, que é, por sua vez, uma solução já existente, identificando desta forma as vantagens na utilização da nova ferramenta em frente as ferramentas já existentes.

## 1 REFERENCIAL TEÓRICO

Os conceitos de orientação a objetos e banco de dados relacional serão detalhados neste capítulo de forma a se obter um entendimento geral da utilização de um dos paradigmas para, assim, ser possível analisar a estrutura de uma solução ORM (Mapeamento Objeto-Relacional).

### 1.1 Conceitos de Orientação a Objeto

O uso do modelo de programação orientado a objetos passou a ser uma prática essencial para o desenvolvimento de sistemas, podendo ser aplicado na maioria das linguagens de alto nível e, assim, disponibilizando ao desenvolvedor recursos avançados para que a programação ocorra de forma mais natural, prática e de fácil manutenção.

Utilizando técnicas levantadas pela engenharia de software e consideradas os pilares do desenvolvimento orientado a objeto, estas características são identificadas como encapsulamento, herança e polimorfismo. Tais conceitos fornecem aos programadores capacidades de melhorar e simplificar a estrutura do sistema.

Será possível entender o conceito da utilização de objetos como um modo de criar uma nova variável ou entidade do sistema, sendo esta entidade composta por um conjunto de tipos primários ou de objetos. Uma classe do sistema unifica os novos objetos ou entidades criadas de uma forma, para que seus estados possam ser modificados por métodos ou funções desta classe. Essa utilização de classes e objetos torna possível a classificação do mundo real em objetos, vez que entidades do mundo real possuem estados e características, podendo, assim, ser naturalmente modeladas em objetos e atributos.

Essa abordagem de estruturar o universo real, como objetos e classes, torna-se a maior vantagem de programação orientada a objeto, possibilitando classificar o mundo ao nosso redor em objetos (FARINELLI, 2007). Essa classificação permite que todos os objetos mapeados possam ser manipulados de toda e qualquer forma. Na Figura 1 é possível visualizar um exemplo dessa classificação, onde são representadas pessoas do mundo real em objetos do sistema, estes objetos possuem seus atributos como nome e idade relacionados às características da pessoa no mundo real.

Figura 1: Três Objetos da Classe “Pessoa”.



### 1.1.1 Herança

Herança é um mecanismo que pode ser aplicado sobre os tipos de classe. Assim, quando uma classe herda de outra, fica subentendido que a classe filha usará a implementação da classe pai.

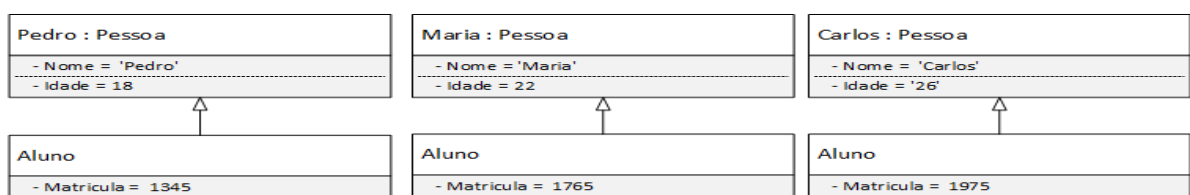
É possível perceber quando será necessária a utilização de herança e quando se deve empregar uma ou mais classes previamente definidas, para que seja possível a organização da estrutura por hierarquia. A principal vantagem da utilização de herança na programação orientada a objeto é a extensão e reutilização de código já existente, não precisando ter que repetir ou reescrever o código do início.

Com herança também é possível criar classes derivadas, que irão possuir a mesma implementação da classe original, mas com suas características próprias. Com isso, se pode interpretar o conceito de hierarquia de classes e subclasses existentes no mundo real.

Em algumas linguagens de programação também é possível utilizar a herança múltipla, permitindo que uma classe derivada possa herdar de uma ou mais classes. Essa característica permite que a classe derivada possa obter uma maior quantidade de características distintas de outras classes derivadas, mantendo uma variável de instância para cada classe herdada (FARINELLI, 2007).

Se pode observar um exemplo de aplicação de herança ao acrescentar uma nova classe na estrutura já definida na Figura 1. A nova classe denominada aluno herdará as características da classe pessoa, dessa forma existirá um vínculo de uma classe do tipo aluno para uma classe do tipo pessoa, tornando possível obter informações como nome e idade a partir de aluno, conforme é possível ver na Figura 2.

Figura 2: Três Objetos da Classe “Aluno” que herdam da Classe “Pessoa”.

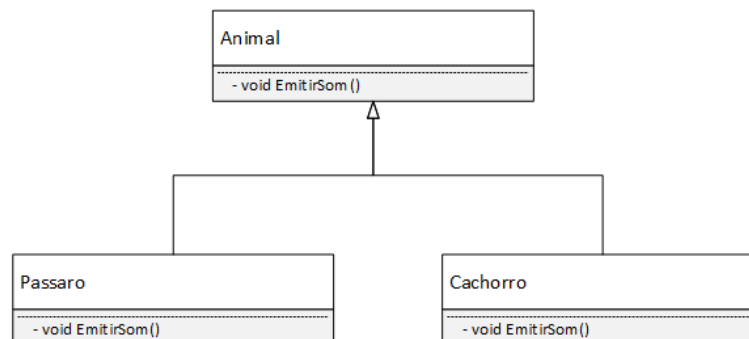


### 1.1.2 Polimorfismo

Com a utilização de herança os objetos sofreram uma especialização, já com a utilização de polimorfismo ocorre uma generalização. O princípio do polimorfismo ocorre quando duas ou mais classes, derivadas de uma mesma classe pai, invocam um método que possui a mesma assinatura, mas com comportamentos diferentes para a classe derivada. Dessa forma, um mesmo objeto pode ser processado de maneiras diferentes, dependendo da classe derivada.

A utilização de polimorfismo na programação orientada a objetos só é possível quando os métodos que irão ser executados ou disponibilizados nas classes derivadas possuem a mesma assinatura do método na classe pai, mudando apenas o comportamento do método na classe derivada, neste caso ocorre uma sobrescrita. Quando se alteram os parâmetros do método, ocorre uma sobrecarga do método, sendo um tipo diferente de polimorfismo, porém é mantido o principal conceito de comportamento diferente para uma mesma assinatura. Na Figura 3 se pode ver um exemplo de polimorfismo.

Figura 3: Exemplo de Polimorfismo no Método “EmitirSom” da Classe Animal.



### 1.1.3 Encapsulamento

Definir encapsulamento como a responsabilidade de informação para cada classe, ou seja, uma classe manipulará informações que são relevantes a ela. O encapsulamento é utilizado em conjunto com a técnica de ocultamento de informação, para manter parte da classe em um ambiente protegido, não permitindo ser visualizada sua implementação fora do escopo da classe (ZÜLLIGHOVEN, 2004).

O encapsulamento é bastante utilizado na programação orientado a objeto, não somente pelo fato de se manter informações ocultas, mas pelo fato da independência do objeto

no sistema. Essa independência dos objetos permite que eles possam ser reutilizados várias vezes em outras partes do sistema ou em outros sistemas, uma vez que a implementação da estrutura de dados dos objetos não precisa ser conhecida por quem utiliza os objetos.

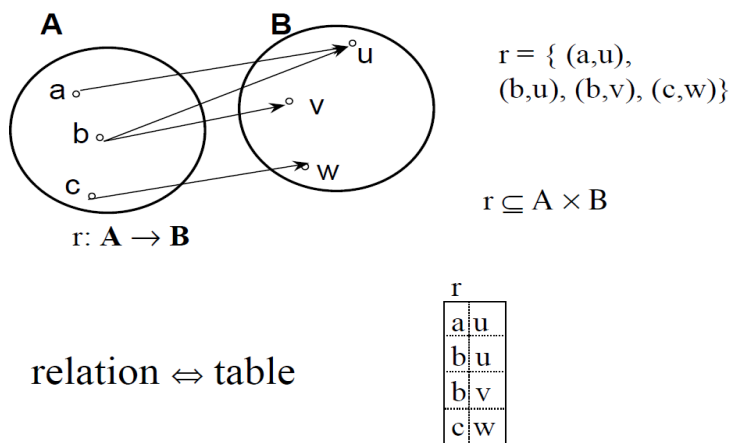
Uma vantagem na utilização de encapsulamento é a definição de nível de acesso, ou seja, até onde é permitido visualizar as informações protegidas da classe. Os níveis variam de *private* (Mais restrito) a *public* (Menos restrito), sendo que, geralmente, se utiliza o nível mais restrito para atributos, deixando a cargo de métodos com níveis menos restrito a possibilidade de alteração desses atributos privados. É possível existir atributos em classes com níveis menos restritos, geralmente atribuído às constantes do sistema, mas essa prática não é recomendada pelo fato que atributos públicos limitam a flexibilidade do código, causando uma dependência da classe ao sistema.

## 1.2 Conceitos de Banco de Dados Relacional

O modelo de armazenamento de dados relacional baseia-se na lógica dos predicados da álgebra relacional e da teoria dos conjuntos. O conceito básico do modelo relacional é que toda informação é descrita como predicados, podendo essas informações compostas de dados ser representadas por relacionamentos (STAJANO, 1998).

Um banco de dados relacional, diferente do modelo orientado a objeto, utiliza um conjunto de tabelas bidimensionais, nas quais informações são registradas em linhas dessa tabela. Para um entendimento mais claro, pode-se interpretar as tabelas como um conjunto e suas ligações como um relacionamento entre tabelas, conforme ilustrado na Figura 4.

Figura 4: Representação de Conjuntos e Relacionamentos.



Na Figura 4, se pode observar que existem conjuntos e tuplas, como também um subconjunto identificado pelas ligações entre os conjuntos. A relação entre o conjunto A e o conjunto B é identificada como um subconjunto r, logo este subconjunto é formado pelas ligações entre os elementos do conjunto A e B, essa ligação no modelo relacional é identificada como um relacionamento.

Além da teoria dos conjuntos, o modelo relacional utiliza a álgebra relacional, que é uma linguagem de manipulação de relações para criação de outras relações. Como na álgebra relacional, os bancos de dados também possuem operadores para manipular os dados, a exemplo dos operadores de união, inserção, subtração e seleção, necessária para criar os relacionamentos entre tabelas (ELMASRI, 2010).

A linguagem de consulta na teoria dos conjuntos não pode ser descrita para um banco de dados relacional, para isso foram realizadas implementações estruturadas baseadas na linguagem dos conjuntos, dentre estas implementações se tem o *SQL (Structured Query Language)*.

### 1.2.1 *SQL (Structured Query Language)*.

Na década de 1970, a IBM junto aos outros fornecedores de banco de dados relacional, queria padronizar o acesso e manipulação aos dados do banco relacional. A IBM desenvolveu então uma linguagem chamada *sequela* ou *Structured Query Language English*, que posteriormente foi denominada de *SQL* (KLINE, 2009).

O grande crescimento das indústrias de tecnologia da informação e a grande necessidade das pessoas em automatizar seus serviços e armazenar dados, exigiu o desenvolvimento de ferramentas para gerenciamento de banco de dados. No âmbito corporativo, a utilização de banco de dados relacionais é fundamental, já que as empresas necessitam armazenar informações como clientes, contas e produtos de uma maneira segura e robusta (KUATÉ, 2009). O SGBD (Sistemas de Gerenciamento de Banco de Dados) é um conjunto de ferramentas que permite manipular dados do banco, utilizando-se de linguagem *SQL* para efetuar operações de inserção, exclusão, alteração e consultas.

O *SQL* utiliza a lógica dos conjuntos para efetuar consultas ao banco de dados, utilizando-se de operadores definidos da teoria dos conjuntos para gerar condições baseadas em predicados. Como os bancos de dados relacionais são estruturados sobre uma lógica

bidimensional, utilizando uma estrutura de tabelas e linhas, cada coluna da tabela é representada por uma característica de uma entidade do mundo real, e a linha dessa tabela pode ser representada como a própria entidade, desde que a tabela também seja representada como um conjunto de entidades.

### 1.2.2 Tipos de dados

No modelo de orientação a objetos, uma classe pode conter vários tipos de dados e esses tipos podem ser representados em bancos de dados relacionais, mas essa representação só é possível se uma classe contiver somente tipos primitivos. Para tipos complexos ou classes que possuam objetos como atributos, não é possível armazenar a informação desse objeto para a entidade na estrutura de um modelo relacional. Esse tipo de armazenamento só é possível quando se utiliza uma ligação entre tabelas.

Para linguagens que utilizam o modelo orientado a objeto, os tipos primitivos não sofrem grandes diferenças entre as linguagens, ao contrário dos bancos de dados relacionais, onde cada fornecedor define os tipos no qual seu *SGBD* trabalhará. As diferenças de tipos de dados entre dois bancos de dados relacionais diferentes podem ser grandes, não somente na ausência de tipos, mas também nos tipos existentes, onde um mesmo tipo de dado pode possuir um tamanho diferente entre bancos diferentes. Um exemplo desse problema é identificado na utilização dos bancos de dados *Postgre* e *Oracle*, pois o primeiro possui o tipo de dado booleano, onde o valor pode ser verdadeiro (*true*) ou falso (*false*), porém não é possível representar esse tipo de dado no *Oracle*, tendo que ter adaptado uma coluna *Char* para armazenar valores de 0 ou 1, para representar verdadeiro ou falso, respectivamente.

Existe também um tipo de dado genérico denominado *Null*, utilizado pela maioria dos bancos de dados para qualquer um dos tipos de dados primários. O *Null* ou nulo não representa um valor vazio, mas sim a ausência de valor ou um valor indeterminado (CUSHMAN e TOLEDO, 2000). Essa diferença pode ser crucial para uma empresa, decorrente que a manipulação de dados com valores nulos pode resultar em valores diferentes do esperado. Esse tipo de problema geralmente ocorre porque valores nulos são ignorados em cálculos de valores agregados, como é no caso de consultas com *AVG*, *SUM* ou *MAX* (KLINE, 2008).

A sintaxe da linguagem SQL é muito fácil e didática. Essa facilidade trouxe um ganho produtivo nas empresas, em relação à simplicidade e custo para aprendizagem, sendo esse um



dos principais motivos para a utilização em grande escala desta linguagem para gerenciamento de banco de dados. Se pode observar na Figura 5 um exemplo de instrução escrita em SQL, onde são consultados dados como nome e estado de uma tabela chamada *authors*.

Figura 5: Instrução SQL e seu Resultado.

```
SELECT au_fname, au_lname, state
FROM authors
```

au_fname	au_lname	state
Johnson	White	CA
Marjorie	Green	CA
Cheryl	Carson	CA
Michael	O'Leary	CA

Fonte: KLINE, 2008

### 1.3 Definição de ORM

A utilização constante dos bancos de dados relacional como principal forma de armazenamento, tornou-se a forma mais confiável de armazenamento de dados, e que o paradigma orientado a objetos é uma boa prática de desenvolvimento de sistemas simples e complexos. A utilização desses dois modelos para elaboração de um sistema não se tornou somente uma boa prática, mas sim um modelo a ser utilizado pelos desenvolvedores (KUATÉ, 2009).

*Object-Relational Modeling (ORM)*, também conhecido como mapeamento objeto-relacional, é uma abordagem conceitual para modelagem de objetos de uma linguagem de programação orientada a objetos para o modelo relacional. Essa técnica realiza uma conversão e tradução de entidades entre os dois paradigmas, permitindo a interação entre um banco de dados relacional com um sistema orientado a objeto e vice-versa.

Um *ORM* também pode ser definido como uma camada de persistência para automatizar a persistência de dados, de forma transparente e eficaz. Essa solução resulta em ganho de tempo de desenvolvimento para o projeto, desde que o trabalho para mapear, traduzir e persistir os objetos em banco de dados relacional seja feita de forma automática pela camada de persistência.

A utilização de uma solução *ORM* torna-se essencial para projetos grandes e complexos, vez que a incompatibilidade entre os dois modelos exige um esforço significativo de tempo e codificação. Foi estimado que na maioria de projetos a serem desenvolvidos em uma linguagem de programação orientada a objeto, que necessitam persistir dados em um banco relacional, obteve-se um acréscimo de 30% de código no sistema, referente à manipulação manual de *scripts SQL* para persistência dos dados. Mesmo com a codificação manual para resolver o problema de incompatibilidade entre os dois modelos, esse problema de comunicação entre os modelos contribui-o para o fracasso da maioria desses projetos, devido à complexidade e inflexibilidade de suas camadas de abstração com o banco de dados (KUATÉ, 2009).

Uma solução *ORM* não só resolve os problemas de incompatibilidade entre os paradigmas orientados a objetos e relacional, como também torna o desenvolvimento mais produtivo. O trabalho pesado de codificação em um projeto é focado na solução de problemas de negócio, ao invés de problemas de persistência. A diminuição de linhas de códigos no sistema melhora o aspecto de manutenibilidade e legibilidade, consequentemente os desenvolvedores poderão realizar manutenções no código com maior agilidade e obter um entendimento mais sucinto das regras codificadas.

Para que seja possível uma integração entre os dois modelos, torna-se essencial realizar um mapeamento dos objetos para que, assim, possa ser possível criar uma camada de tradução para realizar a transformação de objeto em dados relacional. Embora realizar um mapeamento de classes ou entidades de um modelo orientado a objeto possa ser relativamente fácil, esse trabalho não pode ser reaproveitado para outros sistemas, ocasionado retrabalho para equipe de desenvolvimento. Uma alternativa para esse tipo de problema é utilizar uma solução que realize esse mapeamento de uma maneira automática.

### *1.3.1 Mapeamento*

Devido a necessidade de um mecanismo de tradução de objetos para dados relacional, o mapeamento, por sua vez, é uma parte essencial para qualquer solução *ORM* (KUATÉ, 2009).

Um mapeamento de entidade pode ser definido como uma tradução do objeto para um dado relacional. O objetivo de um mapeamento para uma solução *ORM* é poder definir uma classe específica do modelo orientado a objeto para uma tabela específica do modelo

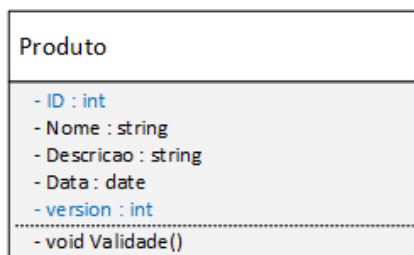
relacional. Essa tradução também reflete os atributos da entidade, que consequentemente estão relacionados às colunas da tabela (AGUIAR, 2009).

Para que seja realizado um mapeamento de uma entidade para o modelo relacional, é necessário ter o nome da entidade igual ao da tabela do banco de dados. Esse trabalho também deve ser feito com os atributos da entidade. Existem soluções *ORM* que fornecem um arquivo de configuração do mapeamento com o objetivo de definir parâmetros de configurações para as entidades, que serão mapeadas. Um desses parâmetros possibilita mapear entidades com nomes diferentes de tabelas.

Existem algumas regras que precisam ser adotadas quando se trabalha com o modelo relacional. Uma dessas regras é a exigência da criação de um campo identidade para todas as entidades, esse campo é relacionado à chave primária de uma tabela do banco. Normalmente, esse campo é um inteiro que deverá ser incrementado na tabela, tornando possível existir um dado relacional referente a uma instância do objeto. Esta criação de campos que não são necessárias no modelo orientado a objeto é chamada de *Shadow Information* (AMBLER, 2003).

*Shadow Information* ou informação oculta é uma prática adotada pela maioria dos mapeamentos objeto-relacional, decorrente de que o modelo relacional tem a necessidade de algumas regras para definir a integridade dos dados. Essas regras no modelo relacional refletem em novos atributos para as entidades do modelo orientado a objeto. Esses atributos, na maioria das vezes, não são necessários no domínio do negócio do sistema. Algumas das regras que podem ocasionar informações oculta no modelo orientado a objeto são a utilização de chave primaria, controle de concorrência e número de versão. Se pode observar um exemplo de entidade com informação oculta na Figura 6, na qual os atributos *version* e *ID* não são necessários para a entidade, mas foram criados para atender a necessidade do modelo relacional.

Figura 6: Entidade com *Shadow Information*.



## 1.4 Soluções *ORM* para Linguagem C#

A utilização de soluções *ORM* em projetos de desenvolvimento está se tornando uma prática bastante utilizada em empresas e fábricas de *software*. Muitas empresas optam por utilizar soluções já existentes e conhecidas no mercado de trabalho, atendendo em muitos aspectos às necessidades de informatizar seus processos.

Para uma aplicação que precise persistir dados em um banco de dados, é necessário existir uma camada de persistência, para poder converter objetos do paradigma orientado a objeto para dados relacionais. Existem muitas técnicas e soluções disponíveis para criar uma camada de persistência (KUATÉ, 2009).

Algumas soluções mais utilizadas para linguagem C# estão listadas logo abaixo:

- *ADO.NET*
- *NHibernate*

Cada solução ou técnica adotada possui suas vantagens e desvantagens para o projeto, sendo a solução *NHibernate* a alternativa mais apropriada para sistemas de grande porte.

### 1.4.1 *ADO.NET*

*ADO.NET* é uma tecnologia disponibilizada pelo *framework ASP.NET*, fornecendo diversas classes para persistências de dados. Essas classes permitem efetuar diversas operações de manipulação com banco de dados, utilizando-se de *provider* (Provedor de acesso ao banco) específico .NET, ou utilizando *driver ODBC* (MACDONALD, 2003).

O *ADO.NET* é uma evolução da antiga tecnologia *ADO* (*ActiveX Data Objects*), também fornecida pela *Microsoft* para conectar a um banco de dados, sendo usado em sistemas com linguagem *ASP* (LIMA, 2002). *ADO* é um conjunto de objetos que permite acesso a banco de dados, podendo executar instruções *SQL* para obter registros do banco de dados agrupados em um *RecordSet*, possibilitando utilizar mecanismos para navegação entre os dados agrupados.

A tecnologia *ADO.NET* possui algumas diferenças do seu antecessor *ADO*, sendo uma delas a utilização de dados em *cache* na máquina local, ou seja, a consulta ao banco é realizada e os dados são armazenados em uma estrutura desconectada do banco. Outra

característica é a possibilidade de utilizar várias técnicas para manipular os dados, mesmo estando desconectado do banco, diferentemente do *RecordSet* disponível pela antiga tecnologia *ADO* (LIBERTY, 2009).

#### 1.4.2 NHibernate

O *NHibernate* é um popular *framework* de código aberto para mapeamento objeto-relacional ou simplesmente uma ferramenta que cria uma “representação virtual” dos objetos do banco de dados dentro do código fonte (CURE, 2010). Criado em 2005 por Sergey Koshcheyev, o *NHibernate* foi uma adaptação para linguagem C# da solução *Hibernate*.

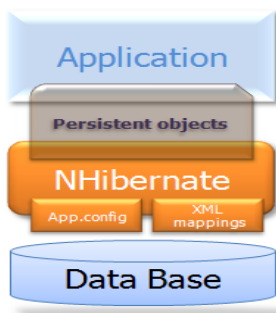
A estrutura do *NHibernate* é bastante similar ao *Hibernate* para *Java*, com a diferença que é utilizado para a tecnologia *.NET*. Com as características principais de uma solução *ORM*, este *framework* tem como responsabilidade abstrair do sistema a camada de persistência, lidando com a tarefa de persistir os objetos em um banco de dados relacional.

A camada de persistência é identificada como um conjunto de classes e métodos usados para fazer a tradução entre os dois modelos, garantindo a persistência de objetos no banco relacional. O *NHibernate* fornece todos os recursos necessários para construir uma camada de persistência completa para o sistema, sendo capaz de persistir em banco de dados relacionais os dados referentes a objetos, mantendo as relações e interligações destes objetos.

#### 1.4.3 Arquitetura do NHibernate

Existem três estruturas essenciais dentro da arquitetura do *NHibernate*, conforme se pode observar na Figura 7, identificadas como *Persistent Objects*, *App.config* e *Xml mappings*.

Figura 7: Arquitetura do *NHibernate*.



Fonte: Definição de Arquitetura do *NHibernate*.

Os *Persistent Objects* são identificados como as entidades do sistema e não fazem parte da estrutura do *NHibernate*, mas são necessários para fazer a comunicação da aplicação com o *framework* e posteriormente ao banco de dados. Essas entidades precisam ser modeladas seguindo as regras definidas no modelo *POCO* (*Plain Old CLR Object*). *POCO* é o padrão utilizado para identificar classes simples ou entidades do sistema, se faz necessário a utilização desse padrão para o funcionamento como classes de persistência ou *Persistent Objects*. Além disso, utilização do modelo *POCO* mantém as entidades do sistema desacopladas do mecanismo de persistência, ou seja, sem dependência com o *NHibernate* (AVANSINI, 2012).

*XML Mappings* é um arquivo de mapeamento de entidades, no qual, é escrito em linguagem *XML* (*Extensible Markup Language*). Foi adotada a utilização desse arquivo como padrão para mapeamento de entidades, decorrente de ser um arquivo que pode ser personalizado no momento de implantação ou até mesmo em tempo de execução. Esse padrão de *XML* não significa que é necessariamente a melhor prática, existem bibliotecas ou extensões para *NHibernate* que tornam o mapeamento mais produtivo e intuitivo, como é o caso do *Fluent NHibernate* e *Mapping Attributes* (SENDIN, 2010).

*Fluent NHibernate* é uma biblioteca que permite o mapeamento das entidades na própria aplicação, não sendo necessária a criação de um arquivo com linguagem *XML*. A vantagem de se utilizar essa biblioteca é a possibilidade de criar classes de mapeamento para cada entidade, tornando o código mais organizado. Outra vantagem é a não utilização de arquivos *XML* que exigiam um conhecimento prévio da linguagem e da estrutura a ser mantida na construção do arquivo, além da quantidade de linhas a serem codificadas. Na Figura 8 se pode observar a diferença entre as duas técnicas.

Figura 8: Diferenças entre *XML Mapping* e *Fluent NHibernate*.

Arquivo XML de Mapeamento	Código de Mapeamento Fluent NHibernate
<pre> &lt;hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"&gt;   &lt;class name="Teste" table="usuarios"&gt;     &lt;id name="Id"&gt;       &lt;column name="Id" /&gt;       &lt;generator class="identity" /&gt;     &lt;/id&gt;     &lt;property name="Nome"&gt;       &lt;column name="Nome" length="15" not-null="true" /&gt;     &lt;/property&gt;     &lt;property name="Email"&gt;       &lt;column name="Email" length="50" not-null="true" /&gt;     &lt;/property&gt;   &lt;/class&gt; &lt;/hibernate-mapping&gt; </pre>	<pre> public UsuarioMapping() {     Id(x =&gt; x.Id);     Map(x =&gt; x.Nome).Not.Nullable().Length(15);     Map(x =&gt; x.Email).Not.Nullable().Length(50); } </pre>

*Mapping Attributes* é uma biblioteca similar ao *Fluent NHibernate*, com a diferença de que o mapeamento das entidades é realizado na própria classe, feito através de *Annotations*. Essa técnica permite que o desenvolvedor faça o mapeamento da entidade no momento da sua criação, aumentando a produtividade do desenvolvimento. Outra vantagem é a legibilidade do código, pois será possível identificar a relação do objeto à tabela e dos atributos com às colunas, consultando a própria entidade, evitando do desenvolvedor consulta arquivos de mapeamento codificados em linguagem *XML*.

*App.config* é o arquivo de configuração do *NHibernate* responsável por manter informações de conexão ao banco de dados. Nesse arquivo é possível definir parâmetros como *ConnectionString*, *Provider*, *Dialect* e *Driver*, estes parâmetros são necessários para realizar o acesso do *NHibernate* ao banco de dados. Abaixo se pode obter a definição de cada parâmetro de configuração (DENTLER, 2010):

- ***ConnectionString***: É uma *String* de conexão com o banco de dados, possuindo informações como nome do servidor, nome do banco, id do usuário e senha.
- ***Provider***: É uma interface de gerenciamento de abertura e fechamento de conexão com banco.
- ***Dialect***: Cada SGBD possui seu próprio dialeto para consulta de dados, tendo pequenas diferenças entre banco de dados, o *NHibernate* necessita saber qual o dialeto a utilizar.
- ***Driver***: É a estrutura que será utilizada para conectar e interagir com a base de dados.

## 2 YLIBRARY

Será apresentado neste trabalho o desenvolvimento de um *framework* que faça a tradução dos objetos da aplicação para a estrutura de banco de dados relacional, mantendo os principais conceitos do paradigma orientado a objetos e eliminando o problema de impedância entre os dois paradigmas.

### 2.1 Objetivos

A utilização de uma ferramenta *ORM* desenvolvida para objetivos específicos da empresa permite reutilizar os mesmos padrões de desenvolvimentos já aplicados nos projetos existentes e em novos projetos que utilizarão o *ORM*, essa vantagem permite a fácil migração da camada de persistência dos projetos já construídos para a estrutura da nova ferramenta desenvolvida. O desenvolvimento de projetos utilizando um *framework* permite com que a equipe se dedique os esforços em codificação das classes ou nas regras de negócio, abstraindo onde e como esses objetos serão persistidos na base de dados.

Quando se trabalha em um projeto que utiliza a plataforma relacional para armazenamento de dados e o paradigma orientado a objetos, costuma-se utilizar dois modelos conceituais necessários para o entendimento da estrutura do projeto nesses dois paradigmas. O modelo de classes *UML* (*Unified Modeling Language*) representa os objetos da aplicação enquanto o modelo entidade-relacional (MER) é responsável para representar os objetos da base de dados. Esses dois documentos são necessários para obter o entendimento da tradução dos objetos em tabelas e vice-versa. Apesar de o modelo de classes e o modelo de entidade-relacional serem documentos bastante utilizados e de fácil elaboração, existe uma sobrecarga quando se associada ao tempo do projeto, considerando a necessidade de obter o conhecimento específico para elaborar estes documentos. Utilizando um *framework* capaz de traduzir e persistir os objetos do modelo orientado a objetos para a estrutura do modelo relacional, se pode projetar um sistema tomando como base o modelo de classes UML, independente do modelo entidade-relacional, pois a responsabilidade da tradução das entidades para o modelo relacional é do próprio *framework*.



## 2.2 Estrutura Interna

A estrutura do *framework* é subdivida em 2 (dois) projetos, um denominado *YLibrary* e outro com o nome do banco de dados no qual o *framework* trabalhará. Devido aos bancos de dados possuírem peculiaridades únicas, esse segundo projeto é necessário existir, dentro da estrutura interna, haja vista que alguns bancos de dados possuem ou não alguns tipos primitivos e outros manipulam um determinado tipo de dado de uma forma diferente. Desta forma é necessário existir um projeto específico para cada banco de dados, garantindo um bom nível de desacoplamento da solução, uma vez que um projeto é responsável por tratar das regras específicas para um determinado banco de dados e o outro é responsável por tratar os aspectos de manipulação dos objetos do modelo orientado a objeto.

O projeto *YLibrary* possui 3 diretórios, os detalhes de cada diretório serão resumidos abaixo:

- **Classes:** esse diretório é responsável por conter as entidades internas que irão ser utilizadas durante a execução de algum procedimento de persistência ou manipulação do banco. Também possui a classe de configuração do projeto, essa classe de configuração tem como responsabilidade obter as informações de customização do projeto, que será utilizada durante a execução dos procedimentos alterando os valores predefinidos do projeto.

- **Factory:** é o principal diretório do projeto, contendo as classes de gerenciamento do banco de dados, utilizando as *interfaces* disponíveis no *namespace System.Data* da *Microsoft*, possibilitando gerenciar uma transação e executar os scripts gerados pelo projeto do banco de dados, independente de qual banco de dados está sendo utilizado. Também possui as classes de reflexão, necessárias para obter as informações das entidades do negócio, possibilitando criar os nomes das tabelas e colunas como também definir o tipo da coluna através do tipo de atributo da entidade.

- **Util:** é o diretório auxiliar do projeto, nele existem as classes para criação do arquivo de mapeamento de entidades no modelo relacional.

O segundo projeto nomeado com o nome do banco de dados possui um diretório, os detalhes do diretório serão resumidos abaixo:

- **Classes:** esse diretório possui a classe Entidade, da qual as entidades de negócio do projeto deverão herdar. É com essa herança que o *framework* pode obter as informações de atributos por reflexão. Nesse diretório também existem as classes auxiliares para consultas a objetos no banco de dados.

Utilizando uma camada de persistência independente e desacoplado do projeto principal, permite ao *framework* centralizar o gerenciamento e criação dos *scripts* para consultas e manipulação de dados (*insert*, *update*, *delete*), como também isolar os acessos da aplicação diretamente ao banco. Essa característica permite com que o SGDB (Sistema Gerenciador de Banco de Dados) manipule as tabelas executando os *scripts* gerados pelo *framework*, independente da entidade de negócio ou regras da aplicação. Outras características da utilização do *framework* no projeto estão citadas abaixo:

- **Transações:** garante a consistência dos dados a serem gravados, como também possibilita desfazer (*Rollback*) as alterações anteriores em casos que ocorrer alguma exceção ou falha. Essa consistência é possível, pois é utilizada a *interface IDbTransaction* disponível na linguagem C#, garantindo um nível de isolamento da transação antes da execução da *query*, podendo ser feito o *commit* em caso de sucesso ou *rollback* em caso de exceção.

- **Queries SQL:** apesar das inúmeras vantagens ao se utilizar uma camada de persistência que abstrai as operações de gravação, atualização, exclusão e consulta, existira casos em que a ferramenta não atenderá a um determinado processo, desta maneira é preciso disponibilizar a possibilidade do desenvolvedor executar *queries* em linguagem *SQL*. Isso é possível, pois o *framework* trata todas as entidades como um tipo objeto, dessa forma ao executar uma *query* seu retorno é uma lista de objetos.

- **Controle de Concorrência:** em grandes aplicações nas quais exista uma grande quantidade de acessos, é preciso garantir a integridade dos dados para acessos simultâneos, prevenindo o sistema de inconsistências, desta forma a ferramenta deve proporcionar mecanismos para prevenir a utilização do mesmo registro do banco de dados em acessos simultâneos. Assim como o controle de transações o nível de isolamento, utilizando a interface disponível na linguagem C#, garante que uma transação aguarde o *commit* de outra que esteja utilizando a mesma tabela, não permitindo dessa forma a ocorrência de inconsistências.

- **Portabilidade:** decorrente do mercado prover diversos bancos de dados, a camada de persistência deve atender a estes bancos de dados existentes como também prover a possibilidade de adaptação a novos. Essa portabilidade é possível, pois o *framework* foi estruturado utilizando interfaces de conexão e transação, dessa maneira o *framework* pode ser adaptado para qualquer banco de dados, bastando apenas criar o projeto com as classes específicas para cada banco de dados.

Ao compilar o projeto *YLibrary* são gerados os arquivos com extensão *.dll*, serão esses arquivos que deverão ser adicionados às referências do sistema, para que sejam disponibilizado as classes para persistência e configuração do *framework*. Uma vez que a referência esteja adicionada, as entidades do negócio devem herdar da classe Entidade, disponível no *namespace YLibrary*, essa classe já possui a implementação dos métodos necessários para persistir e consultar dados, dessa forma está entidade de negócio já estará pronta para persistir ou consultar dados no banco.

### 2.3 Mapeamento Objeto-Relacional

Em uma aplicação orientada a objetos o Mapeamento Objeto-Relacional (*ORM*) é a técnica responsável por permitir que um objeto possa sobreviver ao processo que o criou, tendo seu estado armazenado em disco para futuramente ser recriado com o mesmo estado. Essa técnica permite traduzir um objeto em paradigmas diferentes, tornando a persistência automatizada (e possivelmente transparente) de objetos em um sistema orientado a objetos para tabelas de um banco de dados relacional, utilizando-se de metadados que descreve o mapeamento entre os dois paradigmas (KUATÉ, 2009).

Devido ao problema de impedância existente ao elaborar um projeto usando 2 (dois) paradigmas diferentes, o mapeamento dos objetos se torna essencial para o funcionamento do *framework*. Na maioria dos *ORM* existentes no mercado o mapeamento é realizado em um arquivo de configuração, utilizando a linguagem *XML* para representar qual classe do sistema será persistido em qual tabela, o mesmo mapeamento se aplica para os atributos da classe relacionadas as colunas da tabela (KUATÉ, 2009). A tradução de uma classe para tabela exige a criação de alguns atributos no objeto que não são necessárias no modelo orientado a objetos, mas essencial para o modelo relacional (AMBLER, 2003).

Existem algumas regras da lógica relacional que precisam ser mantidas, como chave primária e versão do registro, essas informações não são relevantes no modelo orientado a objetos, uma vez que um objeto possui sua referência única no sistema, não precisando de um identificador para ele. É comum ao construir o diagrama de classes do objeto não exibir os atributos *shadow information*, como também não precisa necessariamente ser implementada

nos objetos de negócios, embora a aplicação precise de alguma maneira gerenciar essas informações ocultas fora do objeto.

A *YLibrary* não exige que seja criado um arquivo de mapeamento objeto-relacional manualmente, pois esse arquivo é criado automaticamente e dinamicamente pela própria aplicação que estiver utilizando o *framework*. Utilizando a API de reflexão disponível na linguagem C#, a *YLibrary* obtém as informações das classes em tempo de execução, mapeando as classes em tabelas e atributos em colunas, como também definindo os tipos de atributos em tipos de colunas, garantindo a integridade do dado a ser persistido. As tabelas que serão criadas no banco a partir de classes do sistema são nominadas com o alias “*tb\_*” no início do nome classe, podendo esse alias ser modificado nas configurações do *framework*, permitindo uma configuração customizada para cada tipo de projeto.

É possível customizar vários aspectos da *YLibrary*, permitindo assim que o *framework* se adapte dependendo do projeto no qual está sendo utilizado. Em todo projeto desenvolvido em linguagem C#, existe um arquivo de configuração de projeto nomeado *Web.config* para projetos *Web*, e *App.config* para projetos *Desktop*, nesse arquivo existe a possibilidade de adicionar parâmetros de configuração da aplicação, sendo esses parâmetros responsáveis pela customização das operações da *YLibrary*. Mesmo com a ausência de parâmetros de customização no arquivo de configuração, o *framework* aplica as configurações padrões, permitindo que o desenvolvedor somente adicione parâmetros de configuração se houver necessidades no projeto.

Uma configuração importante que pode ser customizada dependendo do projeto, é a possibilidade de permitir ou não que o *framework* se encarregue de criar ou alterar estruturas das tabelas baseadas nas entidades de negócios, ou seja, uma vez que uma classe, que já tenha sido mapeada para uma tabela, tenha um novo atributo adicionado, o *framework* se encarrega de alterar a estrutura dessa tabela adicionando esse novo campo automaticamente como uma nova coluna na tabela. Outra configuração existente é a chamada “*Entidade\_Completa*”, esse parâmetro atribuído como falso permite com que a *YLibrary* carregue apenas o identificador da entidade de negócio que possua um relacionamento com outra entidade, ou seja, se tivermos uma entidade cliente que possui uma conta corrente, existe um relacionamento entre essas duas entidades, dessa forma quando a *YLibrary* carregar o objeto cliente na memória, com esse parâmetro de customização estando como falso, é carregado somente o atributo identificador do objeto conta corrente, no lugar de carregar todos os atributos da entidade.

Esse tipo de customização permite um melhor desempenho do sistema, uma vez que nem sempre é necessário carregar em memória a entidade completa.

Todo mapeamento de um objeto para o modelo relacional se faz necessário criar o atributo identificador, sendo este atributo utilizado no modelo relacional como chave primária. No modelo orientado a objeto esse atributo não se faz necessário, pois a própria instância do objeto o define como único. Com a utilização da *YLibrary* a criação desse campo na entidade de negócio não é exigida, pois ela é feita dinamicamente pelo ORM, desta forma a entidade poderá possuir somente os atributos de sua responsabilidade, ficando livre de atributos do tipo *shadow information*. Um dos requisitos necessários para o mapeamento das entidades de negócio funcionar é a classificação dos atributos como anuláveis (*Nullable*), essa exigência é necessária para que o *framework* saiba quando uma informação de um atributo deverá ou não ser persistido, uma vez que um tipo primitivo não anulável ainda possui um valor como padrão. Na figura 9 se pode visualizar um exemplo de entidade de negócio pronta para ser persistida no banco, com exceção do tipo *String*, todos os tipos primitivos precisam ser declarados como anuláveis, no exemplo da figura 9 os atributos *Decimal* e *DateTime* foram declarados como anuláveis, uma vez que esses tipos primitivos não aceitam valores nulos por padrão.

Figura 9: Entidade de Negócio Produtos.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.DataAnnotations;

namespace Entidade.Site
{
    public class Produtos : YLibrary.Entidade
    {
        public string DscNome { get; set; }
        public Decimal? DscValor { get; set; }
        public DateTime? DthAtualizacao { get; set; }
    }
}
```

### 2.3.1 Mapeamento de Herança

É devido à herança que existe a possibilidade de compartilhar métodos e atributos entre as classes em um projeto, tornando-se um mecanismo inteligente para aproveitar código, configurando-se a grande vantagem da herança no modelo orientado a objetos (FARINELLI, 2007). Em um projeto com estrutura hierárquica, a utilização de herança é fundamental, vez que as classes são inseridas em uma hierarquia especializada, permitindo que uma classe específica possa obter as características de uma classe mais geral. A utilização de herança é aplicada somente no contexto do paradigma orientado a objeto, não existindo nos atuais *SGDB*, necessitando, portanto, que o mapeamento efetue de alguma forma a persistência dos dados da entidade para que futuramente possa obtê-las no mesmo estado no qual foi armazenado no banco de dados.

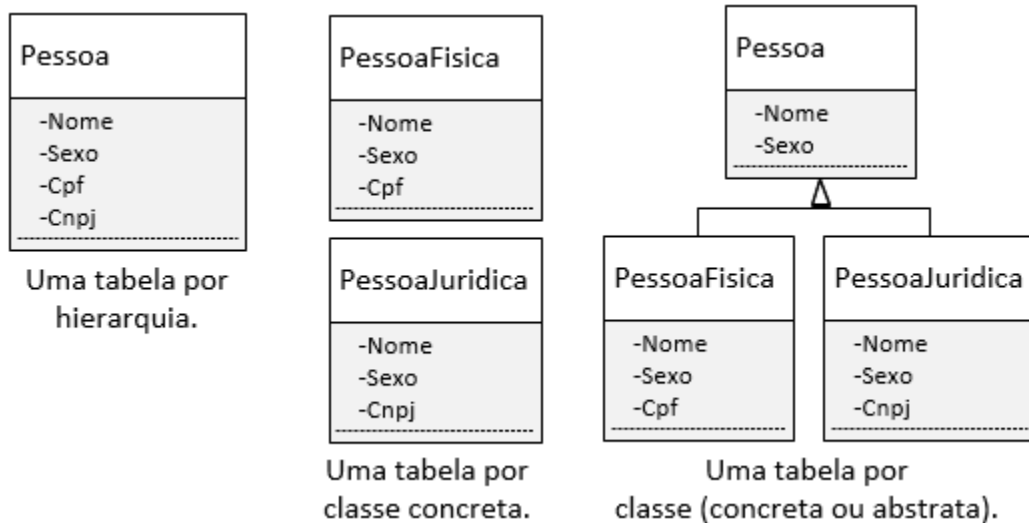
Existem três abordagens (Figura 10) que podem ser adotadas para mapeamento de herança em um *ORM*, suas características, como vantagens e desvantagens, serão citadas abaixo:

- **Uma tabela por classe concreta:** as classes concretas (não abstratas) serão mapeadas em tabelas e seus atributos juntamente com os atributos da superclasse abstrata. Essa abordagem tem como vantagem um mapeamento mais simples e de fácil manipulação dos dados, vez que todos os atributos podem ser encontrados em uma só tabela. Destaca-se como desvantagem o volume de colunas repetidas em tabelas que herdarem da mesma superclasse.

- **Uma tabela por hierarquia:** mapear em um só tabela a classe abstrata e seus atributos, juntamente com os atributos das classes especializadas. O acesso ao banco e a realização de consultas será mais rápido, vez que é possível obter todos os dados da hierarquia em uma só tabela. A desvantagem seria a quantidade de colunas não utilizadas na tabela. Esse mapeamento é o menos utilizado, devido à grande diferença que existirá entre os dois paradigmas, vez que as entidades concretas no modelo orientado a objeto não existirão no modelo relacional.

- **Uma tabela por classe:** todas as tabelas serão mapeadas, incluindo as abstratas. A maior vantagem desse mapeamento é a semelhança entre os dois modelos, vez que a estrutura de tabelas ficará semelhante com a estrutura de entidades. A desvantagem é a difícil implementação dessa abordagem na construção do *ORM* e uma diminuição na velocidade de acesso em relação às outras abordagens.

Figura 10: Abordagens para Mapeamento de Herança.



Devido à linguagem C# não possuir herança múltipla, e o mecanismo de mapeamento de entidades na *YLibrary* exige a utilização de herança para todas as entidades persistentes do sistema, tornou-se bastante complexo implementar a abordagem de uma tabela por classe (concreta ou abstrata). Desta forma foi aplicada a abordagem de uma tabela por classe concreta na construção do *framework*, uma vez que foi a solução intermediária entre a facilidade de implementação e velocidade de acesso. Como dito antes, a desvantagem de não utilizar uma abordagem que contempla toda a hierarquia de classes é devida ao não suporte de associações polimórficas no banco de dados, sendo essas associações comumente utilizadas com ligações de tabelas utilizando chaves estrangeiras, representando um mapeamento semelhante ao do modelo orientado a objeto (KUATÉ, 2009).

### 2.3.2 Mapeamento de Associações

Definir e gerenciar as associações entre as entidades de negócio e traduzir essa relação para tabelas é uma das principais responsabilidades de um ORM, desde que a maioria dos problemas difíceis envolvidos na implementação de um ORM dizem respeito à gestão de associações (KUATÉ, 2009). A associação é a possibilidade de estabelecer relações entre as entidades, assim como no mundo real existem os relacionamentos, no paradigma orientado a objeto as associações são responsáveis por estabelecer essa relação. Em muitos projetos a utilização de associação é fundamental, garantindo um nível de interoperabilidade entre os objetos existentes no sistema, permitindo elaborar consultas específicas e criar processos mais complexos. As associações possuem um conceito chamado grau de relacionamento, onde se

define a quantidade de objetos que podem ser interligados, essa característica define o número de entidades que podem participar de um relacionamento (JOHNSON, 1997).

O mapeamento de associações no modelo orientado a objeto é conceitualmente bastante similar ao do modelo relacional, permitindo assim que tais associações possam ser mapeadas sem grandes dificuldades. Enquanto em um modelo é possível ter uma instância de uma entidade atribuída a outra, garantindo desta forma uma relação entre as duas entidades, no outro modelo essa instância é substituída por uma chave estrangeira, que é utilizada para interligar duas tabelas. Existem 3 (três) tipos de associações utilizadas para representar relações no banco de dados, cada uma com suas peculiaridades e características específicas, sendo elas as associações “um-para-um”, “um-para-muitos” e “muitos-para-muitos”.

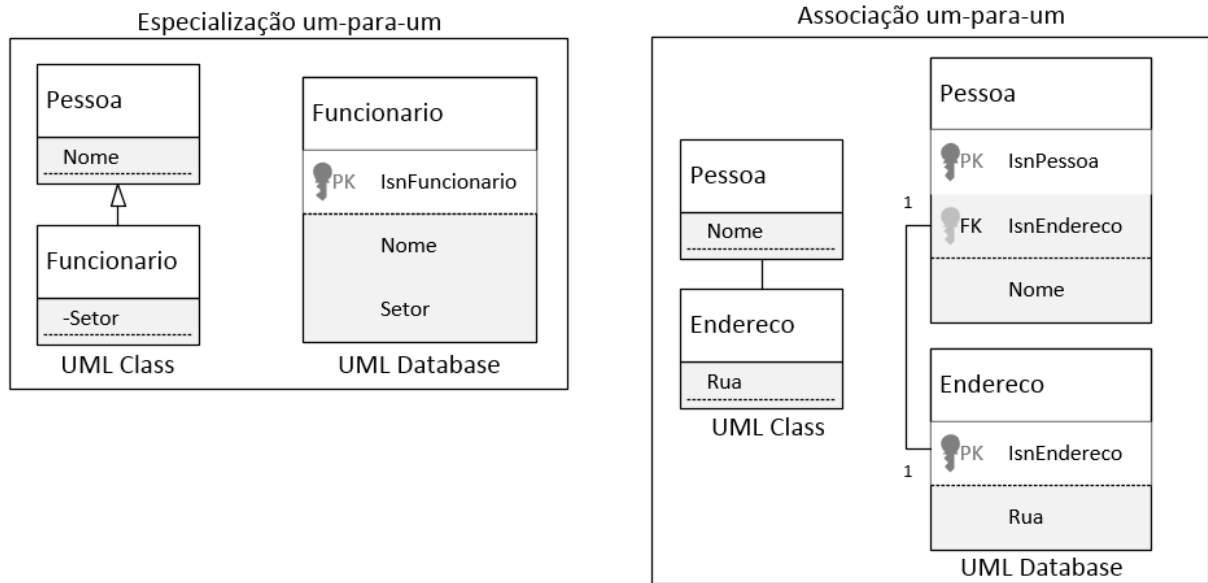
#### 2.3.2.1 Associação “um-para-um”.

Quando é preciso impor uma restrição à relação entre duas entidades, a qual deverá exigir que uma entidade somente possua uma instância de outra entidade, utiliza-se a associação “um-para-um”. Esse tipo de associação permite com que a dupla de entidades interligadas somente exista uma vez, impedindo, dessa forma, a existência de outra relação com as mesmas entidades (JOHNSON, 1997).

Um exemplo desse tipo de associação poderia ser aplicado em um sistema que exige um endereço por pessoa, não podendo duas pessoas ter o mesmo endereço, essa relação é representada por chaves estrangeira no modelo relacional ou uma instância da entidade endereço como atributo da entidade pessoa. Outra aplicação desse tipo de associação é para relacionamentos caracterizados como uma especialização, sendo esse tipo de relacionamento representado por um agrupamento das entidades, podendo a relação entre as duas entidades ser representada como uma única. Utilizando como exemplo a ligação entre a entidade Pessoa e Funcionário, dentro do contexto de uma empresa, essa relação é traduzida no mundo real como uma Pessoa é um Funcionário, desta maneira a associação criada tornou a entidade Pessoa especializada como Funcionário. Na figura 11 podemos observar o diagrama de classe e o modelo entidade relacionamento do exemplo de especialização “um-para-um” entre a classe Pessoa e Funcionário e do exemplo de associação “um-para-um” do exemplo de endereço por pessoa.

Figura 11: Associação e Especialização “um-para-um”.





Esse tipo de associação é mapeado no *framework* utilizando a anotação “UmParaUm” no atributo da entidade, disponível no projeto *YLibrary*, essa anotação permite que seja criada uma chave estrangeira única na tabela de referência, desta forma não permitindo existir dois registros associados ao mesmo registro da tabela referenciada. Para o tipo associação especializada o conceito é semelhante, bastando apenas no modelo orientado a objeto implementar a hierarquia de classes, onde as classes específicas irão herdar da classe mais geral, tendo esta última que herdar da classe entidade do projeto *YLibrary*, garantindo a persistência com a abordagem de uma tabela por classe concreta. A diferença entre esses dois tipos de associações não é identificada apenas no modelo orientado a objeto, sendo também refletida no banco de dados ocasionando a possibilidade de ser criado uma ou várias tabelas, dependendo do tipo de associação e da quantidade de entidades que se relacionam entre si.

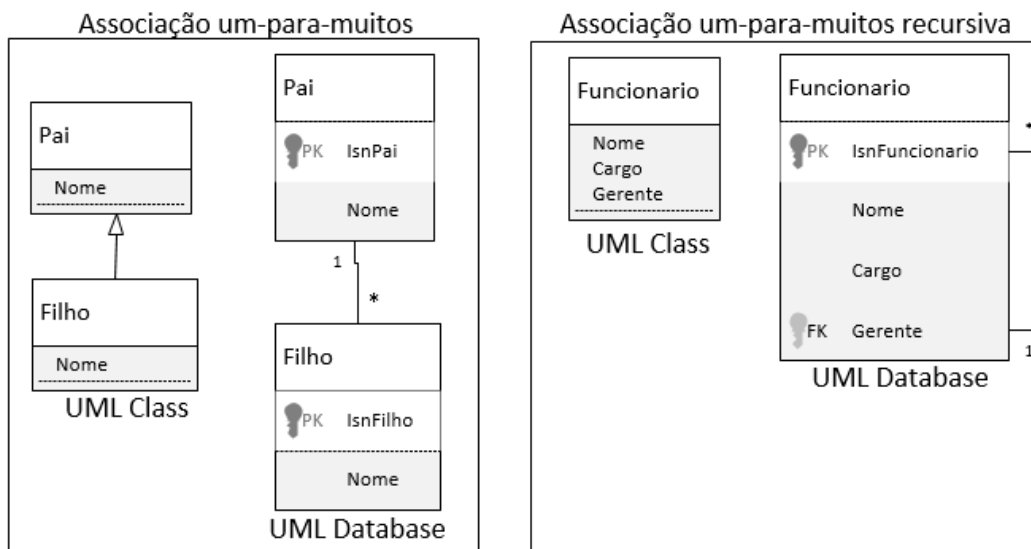
### 2.3.2.2 Associação “um-para-muitos”.

Esse tipo de associação é bastante utilizado, e pode ser aplicado em vários contextos diferentes, mantendo-se como regra a possibilidade de somente uma das entidades possuir relações com mais de uma entidade. Outra aplicação do conceito de associação “um-para-muitos” é poder ter uma entidade que referencia a si próprio ou uma classe do mesmo tipo, esse tipo de associação é chamada de relacionamento “um-para-muitos” recursivo. No modelo orientado a objeto temos como relacionamento recursivo quando uma entidade possui como

atributo um objeto do próprio tipo, enquanto no modelo relacional essa associação é traduzida quando uma chave estrangeira referênciava a sua própria chave primária (JOHNSON, 1997).

Tomemos como exemplo de tipo de associação “um-para-muitos” quando uma classe do tipo pai possui várias instâncias do tipo filho, no contexto do mundo real essa lógica existe e é bastante recorrente. No modelo orientado a objeto e no modelo relacional esse tipo de associação é aplicada da mesma forma que a associação “um-para-um”, podendo ser implementada do mesmo jeito. Já para associação recursiva deve ser feito de forma diferente, vez que é necessário que a entidade possua como atributo um objeto do mesmo tipo, traduzindo no modelo relacional como uma tabela autorreferenciada. Um exemplo para o tipo de associação recursiva existe quando, aplicado em um contexto de uma empresa, um gerente gerencia seus funcionários, sendo que o próprio gerente também é um funcionário, desta forma esse gerente também é gerenciado por ele mesmo. Podemos observar na figura 12 os exemplos citados.

Figura 12: Associação “um-para-muitos” e “um-para-muitos” recursiva.



O mapeamento dessa associação no *framework* é bastante similar com o mapeamento “um-para-um”, tendo como diferença a ausência da utilização da anotação e para modelagem que utilizam associação recursiva, faz-se necessário ter como atributo um objeto do mesmo tipo, permitindo dessa forma que a *YLibrary* faça a persistência da tabela com a chave estrangeira referenciando a chave primária da própria tabela.

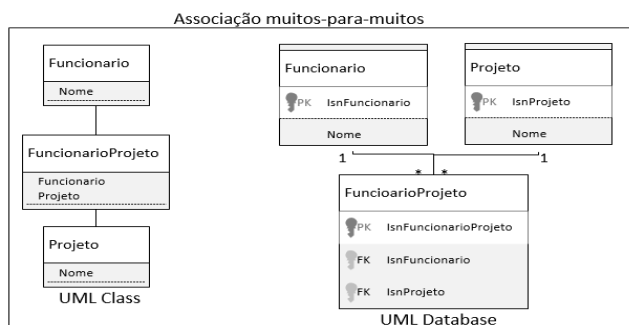
### 2.3.2.3 Associação “muitos-para-muitos”

Esse tipo de associação é menos utilizado do que as outras já citadas, vez que para implementar essa associação no modelo relacional é necessário criar uma tabela auxiliar. Esse relacionamento pode ser caracterizado como várias associações de “um-para-muitos”, onde essas associações são agrupadas em uma tabela, essas relações agrupadas podem ter uma entidade como origem e várias entidades como destino ou vice-versa, caracterizando a associação de “muitos-para-muitos”. Essa associação não possui o conceito de entidade dominante, como podemos identificar nas outras associações, decorrente disso o modelo relacional não possui uma solução direta para esse tipo de relação, sendo necessário um mapeamento de chaves estrangeiras entre as entidades que se relacionam, esse mapeamento é realizado em uma tabela auxiliar (JOHNSON, 1997).

Devido à exigência da criação de uma tabela auxiliar com o agrupamento de relações é no modelo relacional, essa mesma exigência se aplica no modelo orientado a objeto, devendo desta forma ser criada uma entidade auxiliar para mapeamento das relações entre as entidades. Um exemplo para esse tipo de mapeamento existe quando, aplicando-se o contexto de uma empresa, se tem um funcionário que trabalha em vários projetos, da mesma forma que se tem um projeto que possui vários funcionários, essa relação só poderá existir se for criada uma tabela auxiliar para mapear os funcionários pertencentes ao projeto e os projetos pertencentes ao funcionário, como podemos observar na figura 13 que apresenta o modelo entidade relacionamento e o digrama de classe do exemplo citado.

A implementação dessa associação no *framework* exige que seja criada a entidade auxiliar, contendo como atributos as entidades que irão se relacionar, vez que as entidades sejam criadas, a *YLibrary* efetuará a criação tanto das tabelas relacionadas com as entidades como também a tabela relacionada a entidade auxiliar.

Figura 13: Associação “muitos-para-muitos”.



## 2.4 Reflexão

Reflexão é uma *API* disponível na maioria das linguagens de programação modernas que permite a aplicação obter informações das classes em tempo de execução, dessa forma não só é possível obter o nome do método e sua assinatura, mas também permite invocar o método. Essa *API* permite com que a linguagem possa inspecionar o código compilado (*assembly*), podendo ser utilizado para diversos objetivos dentro do sistema, desde a introspecção em tempo de execução de objetos até a inspeção do diretório dos códigos compilados do sistema, no intuito de localizar a implementação de uma referência ou interface. A reflexão é particularmente útil em estrutura adaptáveis, porque elas podem usá-lo para adaptar o seu comportamento em tempo de execução com base na estrutura de seu código (Griffiths, 2012).

Diante do problema de impedância existente ao utilizar os dois paradigmas, se faz necessário realizar o mapeamento entre as entidades de negócio e as entidades do banco relacional, com isso o uso de reflexão se torna essencial para aplicar os conceitos de ORM e automatizar os processos de mapeamento e persistência.

É no diretório *Factory* do projeto *YLibrary* que contém as classes responsáveis por utilizarem a reflexão para obter as informações das entidades e mapeá-las, de forma a garantir a comunicação entre a aplicação e o banco de dados. Com a utilização da classe *PropertyInfo* do namespace *System.Reflection* obtemos todas as informações de um dado tipo de objeto, mesmo esse objeto sendo oriundo de arquivo compilado, dentre essas informações está o nome do atributo e tipo. O nome da tabela e os nomes das colunas são mapeados utilizando as informações extraídas através da reflexão e os parâmetros de customização do *framework*, caso não tenha nenhum parâmetro é utilizado os valores padrões. Uma vez que todos os atributos dos objetos e seus tipos foram mapeados, o projeto referente ao banco de dados se encarrega de gerar o script *DDL (Data Definition Language)*, persistindo esse mapeamento no banco de dados e criando a tabela e as colunas referentes a esse objeto. As responsabilidades de cada projeto não interferem na comunicação entre si, dentro da estrutura interna do *framework*, garantindo um bom nível de desacoplamento e permitindo a possibilidade de adaptar o *framework* a trabalhar com diversos bancos de dados.

A reflexão também é utilizada quando se manipula entidades de negócio que possuam alguma ligação com outras entidades, neste caso o mapeamento é tratado de forma diferente, uma vez que a *YLibrary* identifica que um dos atributos da classe não é um tipo primitivo,

desta forma é utilizado um conceito da reflexão chamada criação de instância (*CreateInstance*). Para atributos não primitivos é necessário instanciar a entidade referente ao este atributo, de uma forma a obter também seus atributos para que assim, através de recursão em pilha, a *YLibrary* possa mapear essa entidade interna, gerando o script *DDL* e criando a tabela dessa entidade. Esse tratamento é necessário, pois uma regra de integridade existente no paradigma relacional são as ligações por chaves estrangeiras (*Foreign Key*), que garante que uma coluna de tabela equivale a outra coluna de outra tabela, esse conceito de ligação por chave estrangeira do paradigma relacional é aplicado no modelo orientado a objeto quando uma determinada entidade possui outra como seu atributo. Essa aplicação do conceito da criação de instância no *framework* é necessária devido ao modelo relacional exigir que o valor persistido em uma coluna relacionada como chave estrangeira, exista na coluna da tabela referenciada, mesmo quando o valor da entidade for nulo, o conceito será aplicado para permitir a criação das tabelas e garantir a ligação entre elas.

## 2.5 Operações *CRUD*

Quando se está trabalhando com informações persistentes, se está preocupado com a persistência e recuperação dessas informações. O termo *CRUD* é um acrônimo dos termos criar, ler, atualizar e excluir na língua inglesa, essas operações primitivas são executadas para recuperar ou armazenar dados, sendo a execução dessas operações na maioria das vezes desencadeadas por eventos na camada de apresentação (KUATÉ, 2009). As operações *CRUD* podem ser feitas manualmente em uma aplicação, porém o trabalho repetitivo de codificar as instruções em *SQL* aumentará o prazo do projeto, com isso se torna essencial a utilização de uma camada de persistência que realizará o trabalho de gerar as instruções, executar e retornar em forma de objetos ou lista de objetos. Com a utilização da *IDE* (*Integrated Development Environment*) Visual Studio e o *framework* .Net, é possível executar operações *CRUD* utilizando a estrutura *Dataset*, essa estrutura é uma representação dos dados no modelo relacional residente na memória da aplicação, desta forma é possível manipular os dados em formas de tabelas na aplicação. Essa estrutura permite a manipulação de dados no banco, porém o conceito aplicado nessa estrutura é o de modelo relacional, dessa maneira não dispõe das vantagens da orientação a objeto.

Uma característica fundamental em uma ferramenta *ORM* é a resolução transparente do problema de impedância existente na utilização dos dois modelos em um projeto,

permitindo que cada operação possa ser realizada sem muitas dificuldades de implementação (KUATÉ, 2009). A execução de operações *CRUD* utilizando a *YLibrary* é muito simples, bastando apenas que a classe responsável pela entidade de negócio herde da classe Entidade, disponível no *namespace YLibrary*, essa associação permitirá que a entidade utilize os métodos Buscar, Inserir, Atualizar e Excluir. Uma vez que a entidade tenha acesso aos métodos *CRUD* basta apenas criar uma instância desse objeto, atribuir os valores que serão manipulados e executar os métodos disponíveis. Abaixo podemos ver um resumo de cada método disponível na classe Entidade:

- **Inserir:** responsável por inserir um novo registro referente ao objeto no seu estado atual. Todos os atributos serão persistidos na tabela, gravando nulo nos atributos que não tiveram valores preenchidos. Não possui parâmetro de retorno, porém o atributo Id localizado na classe Entidade é preenchida com o valor referente a chave primaria do registro na tabela que foi inserido.

- **Atualizar:** responsável por atualizar um objeto já existente no banco de dados. Será atualizado todos os atributos do registro que tiver o mesmo valor de chave primaria na tabela referente ao atributo identificador. Não possui parâmetro de entrada e nem de saída.

- **Excluir:** responsável por remover o objeto do banco de dados. Será removido o registro que tiver o mesmo valor de chave primaria na tabela referente ao atributo identificador. Não possui parâmetro de entrada e nem de saída.

- **Buscar:** responsável por retornar uma lista de objetos do banco de dados. Será localizado todos os registros que tiverem simultaneamente as colunas com os valores iguais aos dos atributos preenchidos da classe. Esse método possui como parâmetro de retorno uma lista de objetos e como esse método possui assinaturas diferentes, existem 3 (três) tipos de parâmetros de entrada para esse mesmo método, permitindo que seja realizada consultas mais elaboradas.

- **Sem parâmetro:** retornará uma lista de objetos que possuem o mesmo valor do atributo preenchido desse objeto. Se a entidade tiver mais de um atributo preenchido os registros que retornaram do banco de dados deverão ter os mesmos valores referente a cada atributo.

- **Com parâmetro *FlagPredicado*:** esse objeto é do tipo *enum (Enumeration)* e possui os identificadores necessários para elaborar consultas específicas para um ou todos os atributos preenchidos, tais como os comandos “*not in*” ou “*contains*” existentes na sintaxe da linguagem *SQL*.

○ **Com parâmetro *SelectCondition***: essa classe permite utilizar o *FlagPredicado* relacionado com algum atributo específico, podendo relacionar um, vários ou todos os atributos da classe com diferentes operadores ou conectores lógicos.

Mesmo que o *framework* possa efetuar todas as operações necessárias para manipulação de dados, ainda existirão casos em que a atual estrutura não poderá atender. Dessa forma é imprescindível permitir ao desenvolvedor a capacidade de executar *queries* nativa em linguagem *SQL*. Com isso a *YLibrary* dispõe de uma classe chamada *Execute*, disponível na instancia do objeto relacionado ao banco de dados, esse método permite que seja executado um script em linguagem *SQL*, retornando uma lista de objetos com os valores relacionado com o retorno da instrução *SQL*, caso seja uma instrução de consulta. Na figura 14 podemos ver um exemplo do código para cada operação *CRUD* e a explicação para cada método segue abaixo:

Figura 14: Operações *CRUD*.

<pre>public void Gravar(String nome, Decimal valor, DateTime data) {     Produtos produto = new Produtos();     produto.DscNome = nome;     produto.DscValor = valor;     produto.DthAtualizacao = data;     produto.Inserir(); }</pre>	<pre>public void Excluir(int id) {     Produtos produto = new Produtos();     produto.Id = id;     produto.Excluir(); }</pre>
<pre>public void Atualizar(int id, String nome, Decimal valor, DateTime data) {     Produtos produto = new Produtos();     produto.DscNome = nome;     produto.DscValor = valor;     produto.DthAtualizacao = data;     produto.Id = id;     produto.Atualizar(); }</pre>	<pre>public List&lt;Produtos&gt; Buscar(String nome) {     Produtos produto = new Produtos();     produto.DscNome = nome;     return produto.Buscar().Cast&lt;Produtos&gt;().ToList(); }</pre>

• **Gravar**: basta apenas instanciar a classe, preencher os atributos com os valores que serão persistidos e executar o método *Inserir*. O *framework* se encarregará de gravar os valores da entidade na tabela automaticamente, gravando nulo nos atributos que não foram preenchidos.

• **Excluir**: basta apenas instanciar a classe, preencher o atributo identificador da entidade e executar o método *Excluir*. O *framework* se encarregará de remover o registro que possua como chave primária o valor atribuído ao identificador.

• **Atualizar**: basta apenas instanciar a classe, preencher os atributos que terão o valor modificado, preencher o identificador da entidade e executar o método *Atualizar*. O *framework* se encarregará de atualizar os valores preenchidos no registro que possuir como

chave primária o valor atribuído ao identificador, deixando o mesmo valor para os atributos que não foram preenchidos.

- **Buscar:** basta apenas instanciar a classe, preencher os atributos que irão ser utilizados como parâmetro de pesquisa e executar o método `Buscar`. O *framework* se encarregará de localizar na tabela todos os registros que possuírem simultaneamente o mesmo valor preenchido nos atributos. O retorno desse método é uma lista de objetos, precisando ser convertida essa lista de objetos para lista de entidades.

Na figura 15 podemos ver um exemplo da utilização do método `buscar` com suas variações de parâmetros de entrada. No primeiro exemplo é possível ver que o parâmetro utilizado foi a classe *FlagPredicado*, essa classe é do tipo enumerador (*Enumeration*) servindo para definir qual o operador relacional será utilizado como critério na execução da consulta. No segundo exemplo podemos observar que foi utilizado como parâmetro um vetor de classes *SelectCondition*, essa classe permite criar uma maior variação ou uma variação mais específica da consulta, possibilitando utilizar vários operadores relacionais para um mesmo atributo. Esse segundo exemplo isenta o programador de preencher os atributos na classe, uma vez que a instância da classe *SelectCondition* exige como parâmetro a informação de qual o atributo, operador relacional e valor será utilizado como critério da consulta.

Figura 15: Operação Buscar com Parâmetros.

```
public List<Produtos> Buscar(Decimal valor)
{
    Produtos produto = new Produtos();
    produto.DscValor = valor;
    return new Produtos().Buscar(YLibrary.FlagPredicado.MaiorIgual).Cast<Produtos>().ToList();
}

public List<Produtos> Buscar(DateTime data, Decimal valormax, Decimal valormin)
{
    YLibrary.SelectCondition condicao = new YLibrary.SelectCondition("DthAtualizacao", YLibrary.FlagPredicado.MaiorIgual, data);
    YLibrary.SelectCondition condicao2 = new YLibrary.SelectCondition("DscValor", YLibrary.FlagPredicado.Menor, valormax);
    YLibrary.SelectCondition condicao3 = new YLibrary.SelectCondition("DscValor", YLibrary.FlagPredicado.MaiorIgual, valormin);
    return new Produtos().Buscar(condicao,condicao2,condicao3).Cast<Produtos>().ToList();
}
```



### 3 ESTUDO DE CASO

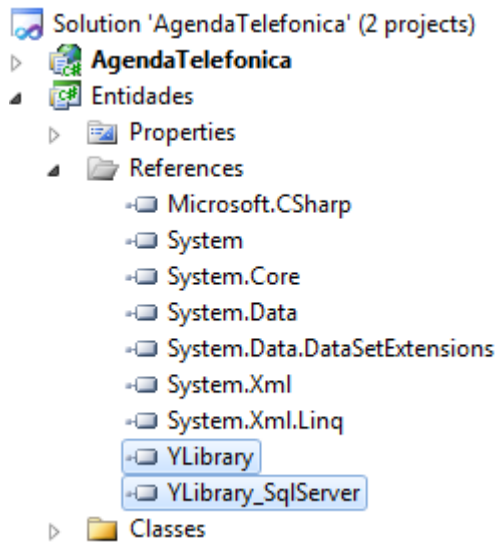
Nesse capítulo será demonstrado como efetuar operações *CRUD* utilizando o *framework YLibrary* em uma aplicação de registro de contatos telefônicos. A demonstração utiliza como banco de dados o *SQL Server 2012*, juntamente com o *Visual Studio 2010* para codificação. Essa aplicação foi escolhida, uma vez que é possível exemplificar os principais conceitos desenvolvidos no *framework*, como também a possibilidade de demonstrar os mapeamentos e sua aplicação na prática.

#### 3.1 Preparações do Ambiente

Uma exigência para qualquer sistema que for utilizar a *YLibrary* como *framework* de persistência é ter o projeto referente as entidades de negócio separadas do projeto referente a camada de visualização, assim como o padrão MVC, cada camada deve ser caracterizada por projeto dentro da solução do sistema, havendo uma camada para as entidades, outra para visualização e a terceira camada que será responsável por manipular o banco de dados. Essa separação entre os projetos é necessária uma vez que para efetuar o mapeamento das entidades o *framework* precisa saber qual o projeto no qual obterá as classes e seus atributos, podendo dessa forma efetuar o mapeamento utilizando reflexão. A camada relacionada a persistência de dados será de responsabilidade do *framework*, não precisando criar um projeto dentro da solução. Uma vez que a solução estiver criada é necessário adicionar nas referências do projeto os arquivos *dll* do *framework*, permitindo que possa ser feito o mapeamento das entidades de negócio.

Na figura 16, pode-se observar que o projeto possui a pasta *References* (Referências), essa pasta é criada automaticamente quando se inicia um projeto utilizando o *Visual Studio*, essa pasta é responsável por conter todas as referências que serão utilizadas no projeto e será nesta pasta que deverá ser adicionado os arquivos *dll* do *framework*. A pasta *Properties* (Propriedades) contém o arquivo *AssemblyInfo.cs*, que é o arquivo responsável por armazenar informações de versão do projeto, arquivo, nome do projeto e descrição. A pasta *Classes* deverá conter as entidades do sistema, como ela não é criada automaticamente, não se faz necessário utilizar esse mesmo nome ou até mesmo utilizá-la, podendo o desenvolvedor criar as entidades livremente no projeto, mas por critério de organização do projeto é sempre recomendado armazenar as entidades em pastas.

Figura 16: Referências do *framework* no Projeto.



A *YLibrary* possui diversos parâmetros customizados, permitindo dessa forma se adequar as necessidades do projeto. Esses parâmetros possuem por padrão valores predefinidos, sendo um desses parâmetros o nome do projeto referente às entidades de negócio, sendo o seu valor padrão o nome “Entidade”, caso o projeto relacionado às entidades de negócio possua um nome de projeto diferente, deverá ser especificado no arquivo de configuração da solução o parâmetro “CLASS\_LIBRARY\_ENTIDADE”, tendo como valor o nome do projeto relacionado as entidades. Abaixo segue todos os parâmetros de customização disponíveis na *YLibrary*:

- **SEMPRE\_VERIFICAR\_BANCO**: Responsável por definir se o *framework* realizará sempre uma consulta ao banco, antes de realizar uma operação *CRUD*, para checar se a tabela do banco de dados está mapeada conforme a classe. Seu valor padrão é “false”.

- **CLASS\_LIBRARY\_ENTIDADE**: responsável por definir o nome do projeto onde estão as entidades de negócio que irão ser mapeadas para tabelas. Seu valor padrão é “Entidade”.

- **ALIAS\_TABELA**: responsável por definir o alias para criação da tabela, sendo esse alias concatenada com o nome da entidade. Seu valor padrão “tb\_”.

- **ALTERA\_TABELA**: responsável por definir a criação ou alteração de tabelas dinamicamente pelo *framework*, utilizando como base o mapeamento das entidades realizadas por reflexão. Seu valor padrão é “true”.

- **TAMANHO\_PADRAO\_STRING**: responsável por definir qual o valor padrão que será aplicado aos atributos do tipo texto. Seu valor padrão é “255”.

- **ALIAS\_CAMPO\_TABELA\_AUTO\_RELACIONADA**: Responsável por definir alias da coluna referente a chave estrangeira de uma tabela auto referenciada. Seu valor padrão é “isn\_”.

- **ENTIDADE\_COMPLETA**: Responsável por definir se o *framework* irá popular toda a entidade associada com a entidade principal, podendo trazer todos os atributos preenchidos ou somente o atributo referente a chave primaria. Seu valor padrão é “false”.

Nenhum desses parâmetros exige que seja adicionado ao arquivo de configuração do projeto, porém a ausência desses parâmetros implicará na aplicação dos valores padrões de cada parâmetro. A aplicação dos parâmetros de customização do *framework* dependerá do tipo de projeto e da sua necessidade, podendo ter facilmente seus parâmetros adicionados no arquivo *Web.config*, para projetos web, ou no arquivo *app.config*, para projetos desktop.

Após a criação da solução e de seus projetos tiverem sido criados e as referências do *framework* adicionadas ao projeto, resta somente adicionar a conexão ao banco de dados. A conexão ao banco deve ser adicionada no parâmetro *connectionstring* do arquivo de configuração do projeto, esse parâmetro possui o atributo *name*, devendo esse atributo possuir o mesmo nome da referência que foi adicionada relacionado ao banco de dados, ou seja, como esse projeto trabalha com o banco *Sql Server*, a referência adicionada teve o nome de *YLibrary\_Sqlserver*. Uma vez que a conexão esteja com os valores corretos, o *framework* se encarregará de efetuar as operações *CRUD* das entidades mapeadas no banco. Na figura 17 se pode visualizar o arquivo de configuração do sistema de exemplo com os parâmetros adicionados e a explicação para cada trecho de código.

Figura 17: Arquivo de Configuração do Projeto.

```
<?xml version="1.0"?>
<configuration>
  <connectionStrings>
    <add name="YLibrary_Sqlserver" connectionString="Server=SERVER\SQLEXPRESS;Database=Agenda;Trusted_Connection=True;"
        providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <appSettings>
    <add key="SEMPRE_VERIFICAR_BANCO" value="false"/>
    <add key="CLASS_LIBRARY_ENTIDADE" value="Entidades"/>
    <add key="ALIAS_TABELA" value="tb_"/>
    <add key="ALTERA_TABELA" value="true"/>
    <add key="TAMANHO_PADRAO_STRING" value="255"/>
    <add key="ALIAS_CAMPO_TABELA_AUTO_RELACIONADA" value="isn_"/>
    <add key="ENTIDADE_COMPLETA" value="false"/>
  </appSettings>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>
</configuration>
```

1

2

3

O bloco 1 possui a informação de conexão com o banco de dados, sendo necessário especificar como nome da conexão o mesmo nome do projeto relacionando ao banco de dados, que foi adicionado como referência na criação do projeto.

O bloco 2 possui os parâmetros customizados de configuração da *YLibrary*, sendo atribuído ao campo *key* o nome do parâmetro e no campo *value* o seu valor que será utilizado no *framework*.

O bloco 3 possui o código responsável por definir qual a versão do *framework.Net* será utilizado para compilação do código no projeto.

### 3.2 Especificação

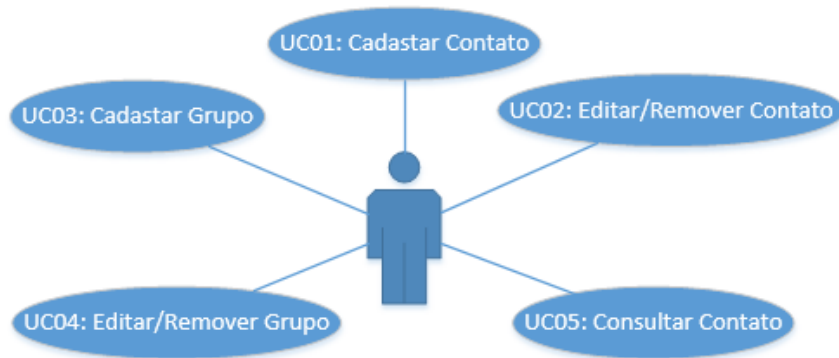
A partir do ambiente de desenvolvimento já configurado, segue-se para a especificação e implementação do sistema de agenda telefônica. Para demonstrar e exemplificar a utilização do *framework*, será construído um sistema de controle de contatos telefônicos, nesse sistema pode-se efetuar as operações *CRUD* (Cadastrar, Editar, Excluir e Consultar). Na tabela 1 se pode ver os requisitos funcionais do sistema de exemplo e os casos de uso ao que se refere cada requisito.

Tabela 1: Requisitos Funcionais.

Requisitos Funcionais	Casos de Uso
RF01: O sistema deverá permitir o cadastro de contatos	UC01
RF02: O sistema deverá permitir a edição e remoção de contatos	UC02
RF03: O sistema deverá permitir o cadastro de grupos	UC03
RF04: O sistema deverá permitir a edição e remoção de grupos	UC04
RF05: O sistema deverá permitir consultar contatos	UC05

Na figura 18 é apresentado o digrama UML de casos de uso do sistema de exemplo, exibindo todos os casos de uso que serão implementados.

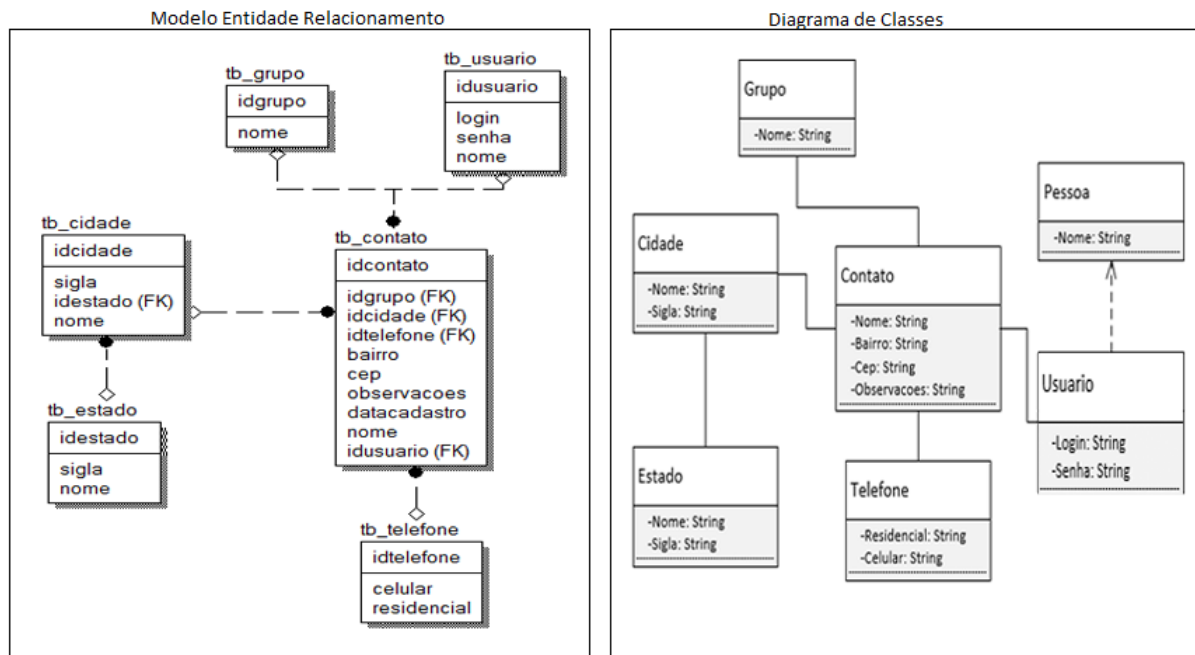
Figura 18: Diagrama UML de caso de uso.



Na figura 19 é exibido o modelo entidade relacional (MER) e o diagrama de classes do sistema, dessa forma podemos observar que as duas modelagens para esse sistema são bastante similares, com uma diferença para a associação entre a classe usuário e pessoa, que utiliza uma associação por herança. Essa similaridade entre os dois modelos pode ocorrer dependendo do tipo de sistema que está sendo implementado, sendo que para sistemas mais complexos os dois modelos podem se tornar bem diferentes.

Conforme é exibido no diagrama de classes, o sistema de agenda poderá armazenar vários contatos para um determinado usuário, da mesma forma com as entidades grupo e cidade, caracterizando uma associação um para muitos entre a entidade Contato e as tabelas Usuario, Grupo e Cidade. A entidade Telefone representará uma associação um para um, uma vez que ficará restrito a um contato possuir apenas um telefone residencial e um celular, não permitindo existir outro contato com o mesmo telefone. A entidade Cidade possui uma associação de um para muitos com a entidade Estado, uma vez que toda cidade possui apenas um estado, mas um estado possui várias cidades. A entidade Usuario possui uma associação por herança com a entidade Pessoa, desta forma os atributos da entidade Pessoa ficam disponíveis para ser usados na entidade Usuario, conforme visto anteriormente a persistência de entidades com herança utilizará a abordagem de persistência por classe concreta, desta forma a tabela que será persistida no banco de dados será a tabela Usuario, contendo como coluna os atributos da entidade Pessoa e da entidade Usuario.

Figura 19: Diagrama de Classes e Modelo Entidade Relacionamento.



### 3.3 Implementação

Com todos os requisitos levantados e definidos, poderá ser iniciada a etapa de codificação do sistema. A solução do sistema de exemplo conterá dois projetos, sendo um responsável pela camada de apresentação, ou seja, possuirá as páginas de interação com o usuário e o outro responsável pelas entidades do sistema.

Primeiramente, se deve criar as entidades de negócio, no qual deverão ser manipuladas dentro da aplicação para posteriormente persistir seu estado dentro do banco de dados. Essas entidades são criadas no projeto do sistema que foi definido, dentro do arquivo de configuração, como o projeto dotado das entidades de negócio, dessa forma o *framework* saberá qual o projeto que deverá mapear as entidades do sistema.

A criação das entidades segue o padrão *POCO* (*Plain Old CLR Object*), bastando apenas acrescentar uma associação por herança da classe Entidade localizado no *namespace YLibrary*. Essa herança fará com que a entidade receba os métodos *CRUD*, permitindo que a entidade possa realizar as operações de persistência no banco, também é com essa herança que a entidade receberá o atributo ID, necessário para obedecer à regra de integridade do modelo relacional, onde exige a utilização de uma chave primaria para identificação do registro, possibilitando dessa forma a recuperação do estado no objeto no qual foi persistido. Na figura 20 podemos visualizar a implementação da classe Contato do sistema de exemplo,

como também sua associação com outras classes do sistema (Grupo, Cidade, Telefone e Usuario).

Figura 20: Classes *POCO*.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using YLibrary.Annotations;

namespace Entidades.Classes
{
    public class Contato : YLibrary.Entidade
    {
        public String Nome { get; set; }
        public Grupo Grupo { get; set; }
        public Cidade Cidade { get; set; }

        [UmParaUm]
        public Telefone Telefone { get; set; }

        public String Bairro { get; set; }
        public String Cep { get; set; }
        public String Observacoes { get; set; }
        public DateTime? DataCadastro { get; set; }
        public Usuario Usuario { get; set; }
    }
}
```

A entidade Contato possui várias associações com outras entidades do sistema, dentre elas existe a associação com a entidade Telefone, conforme detalhado na especificação do sistema, um determinado contato poderá possuir somente uma referência do tipo telefone, caracterizando uma associação um para um, desta forma se faz necessário utilizar a anotação “UmParaUm”, conforme podemos observar na figura 20.

Com as entidades criadas o *framework* realizará o mapeamento objeto relacional automaticamente, sem a necessidade de ser realizada manualmente em um arquivo de configuração ou no código. Esse mapeamento utilizará reflexão para obter o nome das classes, concatenado com o parâmetro de configuração padrão para criar os nomes das tabelas do banco de dados, o mesmo conceito é aplicado para obter o nome e tipo dos atributos, podendo dessa forma ser mapeado em colunas das tabelas. Essa característica permite com que a equipe de desenvolvimento aplique os esforços na implementação da solução, em vez de buscar soluções alternativas para resolver os problemas de impedância existentes entre os dois paradigmas utilizados no desenvolvimento do projeto, aumentando a qualidade do produto a ser construído.

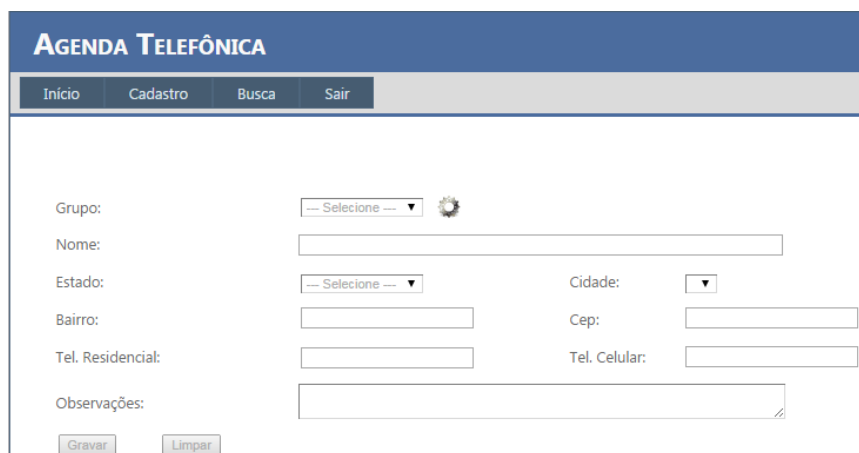
### 3.4 Operacionalidade da Implementação

Foi desenvolvido um sistema de exemplo para demonstrar a utilização na prática do *framework*, desta forma será possível visualizar suas características e vantagens. O sistema de exemplo é um sistema de agenda telefônica, onde um determinado usuário poderá se logar no sistema para cadastrar, editar, remover ou consultar seus contatos.

#### 3.4.1 Operação Cadastrar

Na figura 21 é apresentada a tela de cadastro, nesta tela é possível cadastrar um novo contato preenchendo os campos necessários disponíveis e pressionando o botão gravar.

Figura 21: Tela de Cadastro.



O evento do clique do botão gravar da tela de cadastro possui a chamada do método responsável por efetuar a persistência dos dados na tabela, como visto anteriormente no diagrama de classes, a entidade Contato possui várias associações, desta forma a persistência dos dados preenchidos na tela de cadastro ocasionará na criação de registros em várias tabelas. Na figura 22 é possível visualizar a implementação do método Gravar, sendo este o método responsável por preencher uma instância da entidade Contato com os dados informados pelo usuário. Será detalhado cada bloco de código do método.



Figura 22: Código do Método Gravar.

```

private void Gravar(int? idGrupo, int? idEstado, int? idCidade, String telResidencial,
String telCelular, String Bairro, String Cep, String Observacoes,
String Nome, DateTime Data)
{
    Contato contato = new Contato();|
    Grupo grupo = new Grupo();|
    Cidade cidade = new Cidade();|
    Estado estado = new Estado();|
    Usuario usuario = new Usuario();|
    Telefone tel = new Telefone();|
    tel.Residencial = telResidencial;|
    tel.Celular = telCelular;|
    tel.Inserir();|
    usuario = (Usuario)Session["Usuario"];|
    grupo.Id = idGrupo;|
    estado.Id = idEstado;|
    cidade.Id = idCidade;|
    cidade.Estado = estado;|
    contato.Usuario = usuario;|
    contato.Grupo = grupo;|
    contato.Cidade = cidade;|
    contato.Telefone = tel;|
    contato.Bairro = Bairro;|
    contato.Cep = Cep;|
    contato.Observacoes = Observacoes;|
    contato.Nome = Nome;|
    contato.DataCadastro = Data;|
    contato.Inserir();
}

```

O bloco 1 é responsável por conter a instância de todas as entidades no qual a entidade Contato possui associação, esse bloco é necessário pois as entidades associadas precisam ser instanciadas, preenchidas e atribuídas a entidade principal, antes de ser executado o método inserir.

O bloco 2 contém a persistência de uma entidade associada, neste caso a entidade Telefone, essa persistência é necessária para obedecer à regra de chave estrangeira do modelo relacional, uma vez que a entidade Contato só pode ser associada com a tabela Telefone se esta já possuir algum registro.

O bloco 3 possui o preenchimento das outras entidades associadas, uma vez que elas já existem no banco de dados não se faz necessário persistir a entidade associada antes da principal. Uma característica da *YLibrary* é não exigir a entidade completa para persistência, desta forma é necessário somente preencher o atributo identificador para poder realizar todas as associações existentes da entidade Contato. Outro detalhe desse bloco de código é que a entidade Cidade possui uma associação com entidade Estado, desta forma basta preencher o identificador da instância da entidade Estado e atribuí-la a entidade Cidade, desta forma ao se atribuir a entidade Cidade na entidade Contato, serão obedecidas todas as associações existentes. A entidade Usuario possui uma associação por herança com a entidade Pessoa,

porém a *YLibrary* torna a persistência dessa entidade no banco de dados transparente, não necessitando acrescentar linhas de códigos adicionais para manipular entidades com herança.

O bloco 4 possui o preenchimento da entidade Contato, vez que já se tem todas as instâncias preenchidas, basta apenas atribuir as instâncias nos atributos da entidade Contato. A última linha implementada do método é a execução do método Inserir, efetuando a persistência de todas as informações preenchidas pelo usuário.

Outra característica da *YLibrary* é a possibilidade de criação ou alteração de tabelas conforme a entidade, ou seja, no evento do gravar do formulário de cadastro de contato, o *framework* efetuará uma verificação da tabela se ela existe e se já está conforme o mapeamento, caso a tabela não exista no banco de dados, a *YLibrary* irá criá-la com todas as colunas e associações existentes no modelo orientado a objeto. Essa característica proporciona a criação de toda a hierarquia de tabelas existentes com a entidade principal, ou seja, no exemplo do gravar contato caso não seja atribuído nenhuma das entidades associadas com a entidade Contato, a *YLibrary* identificará que o valor das entidades é nulo, porém mesmo assim criará a tabela para cada entidade no banco de dados, caso ainda não tivesse sido criada anteriormente. Essa característica não diminui a *performance* de gravação, uma vez que a *YLibrary* cria um arquivo de mapeamento de alterações, onde será nesse arquivo que será verificado a estrutura da tabela criada ou alterada, não precisando ser feito uma consulta no banco de dados sempre que for realizar uma gravação. Em contrapartida se for realizado qualquer alteração de tabela, coluna ou banco de dados no modelo relacional manualmente, esse arquivo de configuração deve ser excluído, uma vez que a *YLibrary* precisará recriá-lo novamente fazendo as verificações necessárias para manter a consistência.

### 3.4.2 Operação Buscar

Na figura 23 podemos visualizar a tela de consulta de contatos, sendo possível o usuário efetuar busca por todos os seus contatos já cadastrados. No sistema de exemplo foi implementado a possibilidade do usuário efetuar dois tipos de pesquisa, uma simples e outra avançada, sendo que a diferença entre as duas pesquisas é que a simples pesquisa o valor digitado pelo usuário no atributo nome da entidade contato e a avançada permite efetuar uma busca por todos os atributos disponíveis da entidade.

Figura 23: Tela de Busca Simples.

Detalhes	Id	Nome	Grupo	Bairro	Cep	Cidade	Estado	Tel1	Tel2
<a href="#">Ver Detalhes</a>	2	João Lima	Amigos	Messegana	60000000	Fortaleza	CE	(85) 3333-3333	(85) 9999-9999

O evento do botão Enviar, localizado ao lado do campo de texto, é responsável por efetuar a operação de consulta ao banco de dados, efetuando uma pesquisa para localizar qualquer ocorrência do valor digitado no atributo nome da entidade Contato. Na figura 24 pode-se ver o código responsável por efetuar a consulta no banco de dados, será detalhado cada trecho de código para se ter um entendimento completo da operação.

Figura 24: Código do Método BuscaSimples.

```
private void BuscaSimples(String nome)
{
    Contato contato = new Contato();
    contato.Nome = nome;
    List<Contato> lst = contato.Buscar(YLibrary.FlagPredicado.Like).Cast<Contato>().ToList();

    foreach (Contato ct in lst)
    {
        ct.Cidade = ct.Cidade.Buscar().Cast<Cidade>().ToList()[0];
        ct.Cidade.Estado = ct.Cidade.Estado.Buscar().Cast<Estado>().ToList()[0];
        ct.Telefone = ct.Telefone.Buscar().Cast<Telefone>().ToList()[0];
        ct.Grupo = ct.Grupo.Buscar().Cast<Grupo>().ToList()[0];
        ct.Usuario = ct.Usuario.Buscar().Cast<Usuario>().ToList()[0];
    }
}
```

1  
2

O bloco 1 contém a chamada do método Buscar da classe Contato, passando como parâmetro o enumerador *FlagPredicado.Like*. Ao se utilizado o método Buscar utilizando como parâmetro um enumerador *FlagPredicado*, ocasionará na busca por registros que possua todos os atributos preenchidos, ou seja, todos os atributos que não tiverem valor nulo, utilizando como operador relacional da consulta o enumerador passado por parâmetro. Como é possível observar na figura 24 o atributo nome da entidade Contato foi o único atributo preenchido na instancia do objeto, desta forma a consulta pesquisará todas as ocorrências no banco que possuírem o valor do atributo utilizando o operador *Like*. Após a chamada do método Buscar é feita a conversão do objeto para a entidade, para que seja possível trabalhar com a entidade no lugar de objeto.

O bloco 2 é responsável por preencher as entidades associadas, devido ao valor padrão do parâmetro `ENTIDADE_COMPLETA` ser falso, esse parâmetro permite com que não seja efetuada uma consulta completa na entidade, obtendo somente o atributo identificador das entidades associadas, no lugar de obter todos os atributos. Em alguns *frameworks*, como o próprio Hibernate para java, possui uma implementação semelhante ao do parâmetro `ENTIDADE_COMPLETA`, denominada *Lazy* e *Eager Loading*, essa implementação permite com que a entidade carregue ou não todas os dados da entidade, incluindo seus relacionamentos. Como a entidade Contato possui associações com as entidades Cidade, Telefone, Grupo e Usuario, esse bloco de código se torna necessário para que sejam exibido as informações das entidades associadas, desta forma para que seja exibido o nome da cidade ou o telefone do contato cadastrado é necessário preencher a classe associada.

A busca avançada permite que o usuário possa localizar seus contatos de diversas formas, podendo utilizar de todos os campos disponíveis para gerar sua consulta. Na figura 25 podemos visualizar a tela de busca avançada e assim como é feita na busca simples, na busca avançada basta apenas o usuário informar os dados necessários e clicar no botão Pesquisar.

Figura 25: Tela de Busca Avançada.

Detalhes	Id	Nome	Grupo	Bairro	Cep	Cidade	Estado	Tel1	Tel2
<a href="#">Ver Detalhes</a>	2	João Lima	Amigos	Messejana	60000000	Fortaleza	CE	(85) 3333-3333	(85) 9999-9999

O evento do botão Pesquisar contém a implementação do código responsável por obter a consulta avançada. Diferente como é feita na busca simples, a busca avançada não pesquisa somente pelo atributo nome da entidade Contato, precisando dessa forma identificar quais os campos foram preenchidos para somente assim criar os critérios da consulta. A pesquisa avançada é basicamente a mesma implementação que a pesquisa simples, bastando apenas atribuir os valores preenchidos pelo usuário aos atributos da entidade Contato, como também não utilizar o enumerador `Flagpredicado` como parâmetro do método `buscar`, pois essa consulta localizará registros com os valores exatos, não precisando utilizar o operador *Like*.

Na figura 26 podemos visualizar ao código que será necessário para gerar a consulta avançada.


Figura 26: Código do Método BuscaAvancada.

```
private void BuscaAvancada(int? IdEstado, int? IdCidade, int? IdGrupo,
                          String Nome, String Bairro, String Cep)
{
    Contato cont = new Contato();
    Grupo grupo = new Grupo();
    Estado estado = new Estado();
    Cidade cidade = new Cidade();

    estado.Id = IdEstado;
    cidade.Id = IdCidade;
    grupo.Id = IdGrupo;
    cidade.Estado = estado;

    cont.Nome = Nome;
    cont.Bairro = Bairro;
    cont.Cep = Cep;
    cont.Cidade = cidade;
    cont.Grupo = grupo;

    List<Contato> lst = cont.Buscar().Cast<Contato>().ToList();
}
```



O bloco 1 é responsável por conter a instância de todas as entidades no qual a entidade Contato possui associação, esse bloco é necessário pois as entidades associadas precisam ser instanciadas, preenchidas e atribuídas a entidade principal, antes de ser executado o método inserir.

O bloco 2 possui o código responsável por atribuir nas instâncias criadas no bloco 1 os valores referentes aos identificados das entidades e os valores passados por parâmetros na entidade Contato. Como a entidade estado é um atributo da entidade Cidade e não da entidade Contato, a instancia da entidade Estado é atribuída a Cidade, sendo está atribuída a entidade Contato.

O bloco 3 é responsável por acionar o método buscar, disponível na entidade após herdar os métodos da classe Entidade, disponível na *YLibrary*. A *YLibrary* efetuará a pesquisa por registros no banco de dados que possuam todos os valores atribuídos na classe Contato.


### 3.4.3 Operação Remover

A operação remover é possível acessando a tela de consulta de contatos, conforme podemos ver na figura 23, ao pesquisar por um nome de contato os resultados da consulta serão exibidos na tela, sendo possível acessar todos os detalhes do registro ao pressionar o

link “Ver Detalhes”. O evento desse link redirecionará para a página de cadastro enviando como parâmetro o identificador do registro selecionado, desta forma a página de cadastro pesquisará no banco de dados a entidade Contato que possuir o identificador recuperado no parâmetro. Após a entidade ser carregada do banco, os valores dos atributos dessa entidade serão distribuídos nos campos disponíveis na tela de cadastro. Como a intenção é remover o registro, basta apenas o usuário pressionar o botão Remover, que no evento do botão será executado a implementação de remoção de registro do banco de dados. Na figura 27 podemos visualizar o exemplo dos métodos editar e remover sendo detalhada posteriormente cada trecho de código.

Figura 27: Código do Método Remover.

```
private void Remover(int id)
{
    Contato contato = new Contato();
    contato.Id = id;
    contato.Excluir();
}
```



O bloco 1 do método remover é responsável por instanciar a entidade Contato e atribuir o valor recebido por parâmetro ao identificado da entidade, desta forma ao executar o método Excluir o *framework* executará a operação de remoção do registro na tabela de Contatos, excluindo o registro que possuir como valor na coluna de chave primaria o identificador atribuído na instância da entidade.

#### 3.4.4 Operação Editar

A operação editar é acessado de forma idêntica ao da operação remover, sendo também redirecionada para a página de cadastro para carregar os dados do contato a partir do identificador recuperado por parâmetro. Após os dados do contato serem carregados na tela de cadastro, ficará disponível para usuário alterar quaisquer informações previamente cadastrada no sistema de exemplo. Na figura 28 podemos observar a implementação do método responsável por efetuar a operação editar, sendo detalhado cada trecho de código logo em seguida.

Figura 28: Código do Método Editar.

```

private void Editar(int? id, int? idGrupo, int? idEstado, int? idCidade, String telResidencial,
String telCelular, String Bairro, String Cep, String Observacoes,
String Nome, DateTime Data)
{
    Contato contato = new Contato();
    Grupo grupo = new Grupo();
    Cidade cidade = new Cidade();
    Estado estado = new Estado();
    Usuario usuario = new Usuario();

    Telefone tel = new Telefone();
    tel.Residencial = telResidencial;
    tel.Celular = telCelular;
    tel.Inserir();

    usuario = (Usuario)Session["Usuario"];
    grupo.Id = idGrupo;
    estado.Id = idEstado;
    cidade.Id = idCidade;
    cidade.Estado = estado;

    contato.Usuario = usuario;
    contato.Grupo = grupo;
    contato.Cidade = cidade;
    contato.Telefone = tel;
    contato.Bairro = Bairro;
    contato.Cep = Cep;
    contato.Observacoes = Observacoes;
    contato.Nome = Nome;
    contato.DataCadastro = Data;

    contato.Id = id;
    contato.Atualizar();
}

```

O bloco 1 da implementação da operação editar é basicamente a mesma implementação da operação cadastrar, com a diferença que não possui o trecho de código responsável por executar o método Inserir. Como uma atualização de contato pode sofrer alteração em qualquer de suas informações, o *framework* possibilita que seja preenchida todos os campos da entidade.

O bloco 2 é o que diferencia a operação cadastrar do atualizar, devendo ser necessário preencher o campo identificador da entidade. Uma vez que os campos necessários foram preenchidos, basta apenas acionar método Atualizar, possibilitando dessa forma que o *framework* execute a operação de atualização do contato no banco de dados para o registro que possuir como chave primária o valor do atributo identificador da instancia do objeto. A *YLibrary* atualizará todos os campos da tabela contato que possuírem valores nos atributos da entidade.

## 4 CONCLUSÃO E CONSIDERAÇÕES FINAIS

O desenvolvimento de sistema é uma pratica em constate mudança, surgindo novas técnicas e tecnologias todos os dias. Uma equipe de desenvolvedores de software deve se comprometer em escolher as melhores ferramentas e práticas disponíveis na atualidade, visando atender e concluir da melhor maneira possível o projeto. Uma dessas escolhas é como gerenciar a persistência de dados ou, mais simplesmente, como carregar e salvar dados (KUATÉ, 2009).

A utilização de uma ferramenta que possa efetuar o mapeamento objeto relacional e gerenciar a camada de persistência do projeto, trará um ganho significativo no esforço de desenvolvimento, uma vez que será possível focar os esforços no desenvolvimento do projeto, ao invés de resoluções de problemas de impedância, como também na reutilização de código, uma vez que a ferramenta encapsula toda a lógica para persistência no banco de dados.

Existem diversas ferramentas e práticas para gerenciar a persistência de dados, uma dessas ferramentas é o NHibernate. O NHibernate é um popular, maduro e *open source framework* para mapeamento objeto relacional (ORM), com base no projeto Hibernate do Java (DENTLER, 2010). O NHibernate traz várias características e soluções para problemas existentes devido a incompatibilidade entre os paradigmas orientado a objeto e relacional, desta forma será apresentado um comparativo descritivo entre a ferramenta NHibernate e o *framework YLibrary*, mostrando suas características e soluções para os problemas de impedância na persistência de dados.

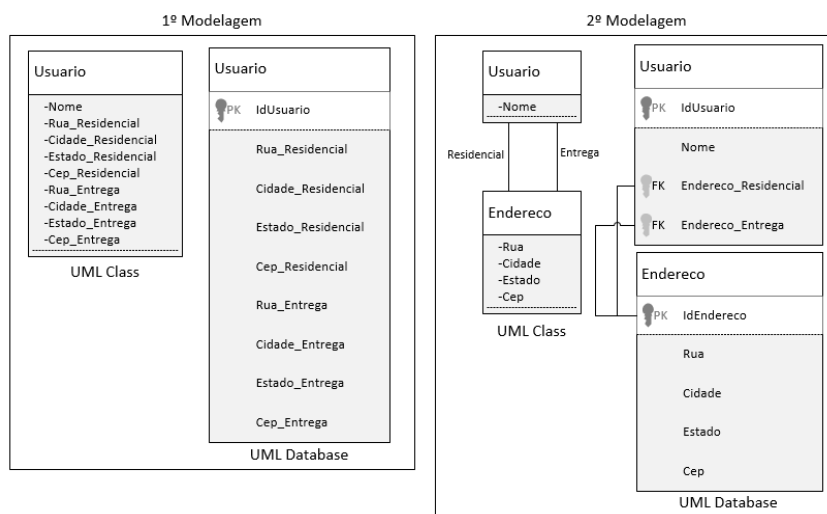
### 4.1 Problema de Granularidade

O paradigma orientado a objeto e o modelo de banco de dados relacional são bem diferentes em termos de granularidade dos dados, devido a mais essa diferença entre os dois paradigmas uma boa ferramenta *ORM* deve prover uma solução para esse problema. O problema da granularidade consiste em que nem sempre o número de classes será igual ao número de tabela do banco de dados (KUATÉ, 2009). Tomemos como exemplo um processo no qual o usuário possui dois endereços, sendo um residencial e outro de entrega, cada endereço armazenará as informações de Rua, Cidade, Estado e Cep, na realização da modelagem de classes poderá ser criado os atributos referente ao endereço na própria entidade, nesse caso serão 8 atributos de tipo primitivo para armazenar as informações de



endereço residencial e de entrega, ou poderá ser criado uma entidade endereço, no qual a entidade usuário possuirá duas instâncias da entidade endereço, uma para residencial e outra para entrega. A segunda solução é mais legível no modelo orientado a objeto e aproveita a vantagem de reutilização de código, porém a primeira possui um desempenho maior, uma vez que não precisa fazer relacionamentos para cada operação. Na figura 29 podemos visualizar a modelagem de classes e de tabelas para cada exemplo citado.

Figura 29: Exemplo do problema de granularidade.



O *NHibernate* utiliza o conceito de composição para resolver esse problema de granularidade, necessitando que seja especificado no arquivo XML de mapeamento da entidade usuário que o atributo endereço residencial e endereço de entrega é um componente da classe endereço, desta forma o *NHibernate* persistirá no banco de dados somente a tabela referente a entidade usuário, acrescentando todas as colunas da entidade endereço para cada instância de endereço na entidade usuário.

A *YLibrary* utiliza o conceito de associação para solução desse problema, diferentemente da solução aplicada pela ferramenta *NHibernate*, que utiliza composição para persistir somente uma tabela referente a duas entidades, a solução por associação persiste a entidade usuário com seus relacionamentos com a entidade endereço, neste caso a tabela referente a entidade usuário possuirá dois relacionamentos com a tabela referente a entidade endereço. Como a *YLibrary* utiliza reflexão para obter as informações da entidade, não é preciso especificar no código ou em arquivo de configuração essas associações, bastando apenas seguir o padrão POCO para criação da entidade e de seus atributos.

## 4.2 Problema de Herança

Uma estratégia utilizada em todos ORM é proporcionar uma solução de mapeamento de classes para tabelas utilizando o conceito de uma tabela por classe, essa abordagem pode atender a grande parte do projeto, mas não atenderá quando se tratar de herança. Uma das principais características do paradigma orientado a objeto é a utilização de herança na modelagem das entidades, porém essa característica não é comum quando se trata do modelo de banco de dados relacional, ocasionando em um dos problemas de impedância entre os dois modelos (KUATÉ, 2009). Tomemos como exemplo um processo de pagamento, onde existe um pagamento por cartão de crédito ou cheque, sendo que ambas possuem as mesmas características da entidade pagamento, caracterizando essa associação como uma herança, uma vez que cartão de crédito ou cheque é um tipo de pagamento, desta forma a entidade cartão de crédito e cheque herdarão as características da entidade pagamento. Conforme foi visto no capítulo 2.3.1, existem três formas para resolver esse tipo de problema, sendo essas soluções a aplicação do conceito “Uma tabela por classe”, “Uma tabela por classe concreta” ou “Uma tabela por hierarquia”.

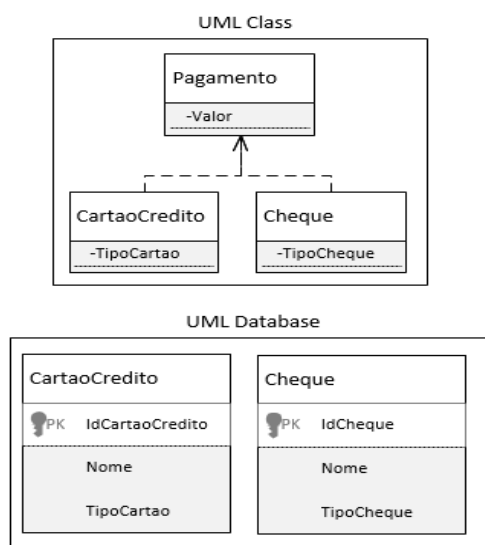
O *NHibernate* permite com que o desenvolvedor possa aplicar o conceito no qual melhor se adapta ao projeto, ou seja, para uma modelagem de classes utilizando herança poderá ser utilizado qualquer um dos conceitos para mapeamento de herança. Independente de qual conceito será aplicado no mapeamento de herança utilizando o *NHibernate*, ainda se faz necessário mapear essa solução no arquivo de configuração XML de mapeamento da entidade, especificando para cada entidade que utilizará herança qual o tipo será aplicado. Na figura 30 podemos visualizar um exemplo de arquivo de configuração XML de mapeamento da herança, necessária para *NHibernate* aplicar o conceito de herança por classe concreta no banco de dados.

Figura 30: Exemplo de Arquivo de Configuração XML.

```
<class name="Pagamento">
  <id name="id" type="long" column="IdPagamento">
    <generator class="sequence"/>
  </id>
  <property name="Valor" column="Valor"/>
  ...
  <union-subclass name="CartaoCredito" table="CartaoCredito">
    <property name="TipoCartao" column="String"/>
    ...
  </union-subclass>
  <union-subclass name="Cheque" table="Cheque">
    <property name="TipoCheque" column="String"/>
    ...
  </union-subclass>
</class>
```

Diferentemente do *NHibernate* que permite que possa ser escolhido qual o conceito será aplicado para o mapeamento de herança das entidades, a *YLibrary* permite que seja aplicado somente o conceito de “Uma tabela por classe concreta”, para resolução de mapeamento com herança. Essa abordagem para permite somente um conceito para mapeamento de herança foi utilizado para manter um dos principais objetivos do desenvolvimento do *framework*, que consiste em garantir para a equipe a aplicação do conceito de desenvolvimento ágil, uma vez que mantendo esse conceito para o mapeamento de herança, não se faz necessário configurar o mapeamento de entidades em arquivos de configuração, como é necessário no *NHibernate*, bastando apenas implementar as classes utilizando os métodos padrões de herança disponível na linguagem. Na figura 31 podemos visualizar a modelagem de classe e de tabelas ao se aplicar o conceito de mapeamento por classe concreta, essa modelagem se aplica tanto para a *YLibrary* como para o *NHibernate*.

Figura 31: Modelagem de Classe e de Tabelas aplicando o Conceito de Herança.



### 4.3 Problema de Identidade

A identidade de um registro no banco de dados relacional é expressa pela chave primaria (*primary key*), porém no paradigma orientado a objeto a identificação do objeto não é, naturalmente, definida por uma chave ou atributo, mas sim pela sua instância. Para manter o nível de integridade entre as informações e no banco de dados relacional é utilizado uma coluna na tabela para armazenar a valor referente a chave primaria do registro. No modelo orientado objeto a identidade de uma entidade é intrínseca, que se baseia em sua posição na

memória ou em convolução definida pelo usuário, normalmente utilizando método *equals*, que está disponível para todos os objetos na linguagem C# (KUATÉ, 2009).

A solução para esse problema no *NHibernate* é o acréscimo de um atributo a mais para todas as entidades persistentes, sendo esse atributo relacionado com a coluna que armazenará o valor de chave primaria na tabela. Conforme já visto anteriormente, esses atributos que não tem importância significativa na entidade são chamadas de informações ocultas (*Shadow Information*), sendo necessárias apenas para resolver problemas de impedância entre os dois modelos. Também é necessário especificar no arquivo XML de mapeamento de entidade a informação que o atributo adicionado é do tipo identidade, dessa forma o *NHibernate* saberá que o atributo adicionado na entidade é o identificador utilizado como chave primaria na tabela. Na figura 32 podemos visualizar o código da entidade usuário e seu mapeamento informando o atributo no qual será relacionado a chave primaria da tabela.

Figura 32: Entidade Usuário e seu Mapeamento.



Na *YLibrary* a solução para o problema de identidade é semelhante ao do *NHibernate*, porém não exige, explicitamente, que seja acrescentado em todas as entidades de negócio o atributo identificador para a solução de impedância. Um atributo identificador é utilizado para todas as entidades persistentes, assim como no *NHibernate*, porém esse atributo pertence à classe Entidade, localizado na referência *YLibrary*, como toda entidade persistente deve herdar desta classe, então todas as entidades terão seu atributo identidade, não sendo necessário implementar o atributo na entidade de negócio e não necessitando de arquivo de configuração para definir qual o atributo da entidade será o identificador do registro na tabela do banco de dados. Na figura 33 podemos visualizar o código da entidade usuário, diferentemente do *NHibernate*, não se faz necessário inserir o atributo identificador e nem

criar o arquivo mapeamento informando o atributo que será relacionado a chave primaria da tabela.

Figura 33: Entidade Usuário já Mapeado.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Entidades.Classes
{
    public class Usuario : YLibrary.Entidade
    {
        public String Login { get; set; }
        public String Senha { get; set; }
    }
}
```

#### 4.4 Problema de Associações

No modelo orientado a objeto, associações representam as relações entre os objetos existentes no projeto. Um exemplo de associação seria a modelagem de entidades em um processo em que todo usuário possui um endereço, sendo usuário e endereço entidades distintas. No mundo relacional uma associação é representada por uma chave estrangeira (*foreign key*) com cópia de valores de chaves de outras tabelas, sendo essa chave estrangeira suficiente para localizar um registro específico em outra tabela. Referencias de objetos são inerentemente direcionais, ou seja, é de um objeto para outro, se em um determinado projeto existir a necessidade de um relacionamento bidirecional entre duas entidades, deverá ser feita duas associações, sendo uma associação para cada entidade (CURE, 2010). Caracterizando-se como associação no modelo orientado a objeto a instâncias das entidades como atributos, e no modelo relacional a utilização de chave estrangeira, é possível identificar outro problema de impedância relacionado aos dois paradigmas.

O *NHibernate* soluciona o problema de associações aplicando os conceitos vistos no capítulo 2.3.2, desta forma tanto associações direcionais como bidirecionais serão atendidas. Sendo uma associação direcional ou bidirecional, se faz necessário especificar no arquivo XML de configuração qual o tipo de associação será aplicado na entidade, podendo ser um para um, um para muitos ou muitos para muitos. Na figura 34 podemos visualizar um

exemplo de mapeamento da entidade Contato com sua associação um para um com a entidade Telefone.

Figura 34: Mapeamento de Associação “um-para-um”.

```
<class name="Contato"
  table="Contato">
  <id name="Id">
    <generator class="native"/>
  </id>
  <property name="Nome" column="String"/>
  <one-to-one name="Telefone"
    unique="true"
    column="Id"/>
</class>
```

A *YLibrary* também utiliza os conceitos apresentados no capítulo 2.3.2, permitindo realizar todas as associações necessárias em um projeto. Apesar do *framework* utilizar reflexão para obter as informações das entidades, esse tipo de informação não pode ser interpretado pelo *framework*, necessitando que seja adicionado uma anotação no atributo da instancia da classe relacionada, qual será o tipo de associação realizada entre as entidades. Para os casos com associação muitos para muitos não precisará de anotação, mas precisará criar entidades auxiliares para gerir essas associações, refletindo essas entidades em tabelas no banco de dados. Na figura 35 podemos ver o exemplo de mapeamento um para um da entidade Contato com a entidade Telefone.

Figura 35: Mapeamento “um-para-um” na *YLibrary*.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using YLibrary.Annotations;

namespace Entidades.Classes
{
    public class Contato : YLibrary.Entidade
    {
        public String Nome { get; set; }
        [UmParaUm]
        public Telefone Telefone { get; set; }
    }
}
```

## 4.5 Conclusão

Desenvolver uma ferramenta para realizar o mapeamento objeto relacional e manter os modelos de desenvolvimento já aplicados na empresa em projetos antigos, requer bastante

conhecimento na linguagem no qual está se implementando, obtendo-se como resultado uma ferramenta adaptada aos modelos e metodologias utilizados na empresa e uma maior agilidade no processo de desenvolvimento dos projetos. Com a ferramenta implementada se tornou possível migrar, sem grande complexidade, os sistemas já desenvolvidos sem o *framework*, como também desenvolver novos sistemas sem sofrer com o problema da grande curva de aprendizagem para equipe de desenvolvimento na utilização de uma nova ferramenta, uma vez que a equipe possa trabalhar utilizando as mesmas metodologias.

Foi possível perceber que a nova ferramenta resolve os principais problemas de impedância existentes ao se utilizar os paradigmas orientado a objeto e relacional, como também fornece um modelo de desenvolvimento ágil, permitindo com que a equipe de desenvolvimento possa focar seus esforços na codificação de suas entidades, ficando a cargo da ferramenta o trabalho de persistir a estrutura e modelagem das classes no banco de dados. Utilizando como comparativo o *NHibernate*, que é uma ferramenta confiável e uma das mais utilizadas atualmente no mercado de trabalho, foi possível analisar as soluções adotadas pela nova ferramenta para os problemas de impedância existentes no desenvolvimento de projetos utilizando os paradigmas orientado a objeto e relacional. Esse comparativo realizado com uma ferramenta confiável no mercado de trabalho, permite que a nova ferramenta possa aderir a novos projetos, uma vez que a garantia da qualidade do projeto está resguardada pelas soluções de impedância apresentadas e pela aplicação dos conceitos de desenvolvimento ágil.

A *YLibrary* possui diversos recursos que permitem um desenvolvimento ágil de projetos na linguagem C#, um desses recursos é a utilização de reflexão para realizar o mapeamento das entidades. Com a reflexão o desenvolvedor não precisa criar um arquivo de mapeamento para identificar a classe e seus atributos para tabelas e colunas, respectivamente, diminuindo a quantidade de arquivos ou linhas de códigos necessárias para persistir as informações no banco de dados. Outra das vantagens da nova *ferramenta* é a utilização de transações, garantindo na camada de persistência a consistência nos dados gravados e sua estrutura modularizada do projeto, permitindo uma melhor adaptação a novos bancos de dados.

O *framework* desenvolvido foi utilizado pela equipe de desenvolvimento da empresa *Integrasoft* para construção de alguns sites, obtendo uma maior agilidade no desenvolvimento dos projetos. Com um percentual de 100% de aceitação pela equipe de desenvolvimento da empresa na utilização do novo *framework*, a *YLibrary* comprovou ser uma ferramenta de fácil aprendizagem, ágil e confiável para implementação de novos

projetos. Abaixo podemos visualizar alguns dos sites que foram desenvolvidos pela empresa utilizando o *framework*:

- [www.papodecamarim.net](http://www.papodecamarim.net): Loja virtual de moda feminina, desenvolvido em novembro de 2011.
- [www.msrenovaveis.com.br](http://www.msrenovaveis.com.br): Site da empresa Ms Renováveis referência na gestão da geração de energia, desenvolvido em janeiro de 2012.
- [www.cageprev.com.br](http://www.cageprev.com.br): Site da fundação cagece de previdência complementar, desenvolvido em outubro de 2012.

Com a utilização da *YLibrary* é possível atender as atuais demandas da empresa, garantindo uma agilidade no desenvolvimento dos projetos e qualidade na entrega dos produtos, porém a ferramenta pode evoluir para que possa ser acrescentado novas funcionalidades. Uma melhoria que pode ser implementada na *YLibrary* seria a capacidade de herdar os componentes do próprio *framework* .Net da *Microsoft*, disponibilizando novos componentes adaptados para criação de formulários persistentes na camada de apresentação, esses componentes modificados possibilitaria a criação de páginas utilizando um modelo já predefinido nos projetos da empresa, aumentando ainda mais a agilidade no desenvolvimento.

Com a automatização existente na *YLibrary* para o mapeamento de entidades e criação da estrutura de tabelas e relacionamentos no banco de dados, seria também possível adaptar a ferramenta para gerar automaticamente os modelos entidade relacionamento. Com a criação automática do modelo entidade relacionamento, seria possível agilizar o processo de documentação do projeto no qual está sendo desenvolvido.



## REFERÊNCIAS

- AGUIAR, Gustavo Maia. **A Impedância, o Mapeamento Objeto Relacional e Implementações**. 2009. Disponível em <<http://gustavomaiaaguiar.wordpress.com/2009/11/10/>>. Acessado em 05 de outubro de 2013.
- AVANSINI, Régis da Silva. **Investigação da efetividade de um Framework Corporativo de persistência de dados com base no Framework *NHibernate* em um ambiente empresarial**. 2012. Disponível em: <<http://www.espweb.uem.br/wp/wp-content/uploads/2012/05/R%C3%A9gis-da-Silva-Avansini.pdf>> Acesso em 26 de setembro de 2013.
- CURE, Aaron B. ***NHibernate 2 Beginner's Guide***. Packt Publishing, p. 8-76. 2010. **Definição de arquitetura do *NHibernate***. <<http://nhforge.org/wikis/reference2-0en/architecture.aspx>>. Acessado em 06 de outubro de 2013.
- CUSHMAN, Pauline; TOLEDO, Ramon. **Schaum's Outline of Fundamentals of Relational Databases**. Paperback, p. 34-36. 2000.
- DENTLER, Jason. ***NHibernate 3.0 Cookbook***. Packt Publishing, p. 48-62. 2010.
- ELMASRI, Ramez. NAVATHE, Shamkant. **Fundamentals of Database System**. Addison-Wesley, p.176-182. 2010.
- FARINELLI, Fernanda. **Tutorial de Programação Orientada a Objetos**. 2007. Disponível em <[http://sistemas.riopomba.ifsudestemg.edu.br/dcc/materiais/1662272077\\_POO.pdf](http://sistemas.riopomba.ifsudestemg.edu.br/dcc/materiais/1662272077_POO.pdf)>. Acessado em 22 de novembro de 2013.
- GRIFFITHS, Ian. **Programming C# 5.0**. O'Reilly Media, p.487-508. 2012.
- JOHNSON. **Database: models, languages, design**. Oxford University, p. 30-44. 1997.
- KLINE, Kevin, KLINE Daniel. **SQL in a Nutshell: A Desktop Quick Reference**. O'Reilly Media, p. 1-26, 2009.
- KUATÉ, Pierre Henr; HARRIS, Tobin; BAUER, Christian; KING, Gavin. ***NHibernate in Action***. Manning Publications, p. 3-99, 2009.
- LIBERTY, Jesse. **Programando C#**. Alta Books, p. 285-299. 2009.
- LIMA, Edwin. **C# e .NET – Guia do Desenvolvedor**. Editora Campus, p. 196. 2002.
- MACDONALD, Matthew; HAMILTON, Bill. **ADO.NET in a Nutshell**. Paperback, p.9-13. 2003.
- SENDIN, Rodrigo. **DDD, *NHibernate* e ASP.NET**. .NET Magazine, Edição N° 72, p. 38-58, março de 2010.

STAJANO, Frank. **A Gentle Introduction to Relational and Object Oriented Databases.** Olivetti & Oracle Research Laboratory, 1998.

ZÜLLIGHOVEN, Heinz. **Object-Oriented Construction Handbook: Developing Application-Oriented Software with the Tools & Materials Approach.** Morgan Kaufmann, p. 35-37, 2004.