UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ – UTFPR
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

PAULO CESAR FERNANDEZ ALCARAZ

ENTITY FRAMEWORK, UMA INTRODUÇÃO AO ORM UTILIZANDO O CODE FIRST

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2016

PAULO CESAR FERNANDEZ ALCARAZ

ENTITY FRAMEWORK, UMA INTRODUÇÃO AO ORM UTILIZANDO O CODE FIRST

Trabalho de Diplomação apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – COADS – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Everton Coimbra de Araújo.

MEDIANEIRA

2016

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

TERMO DE APROVAÇÃO

Entity Framework, uma Introdução ao ORM utilizando o Code First

Por

Paulo Cesar Fernandez Alcaraz

Este Trabalho de Diplomação (TD) foi apresentado às 09:10h do dia 16 de novembro de 2016 como requisito parcial para a obtenção do título de Tecnólogo no Curso Superior de Tecnologia em Manutenção Industrial, da Universidade Tecnológica Federal do Paraná, *Campus* Medianeira. Os acadêmicos foram arguidos pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado com louvor e mérito.

Prof. Everton Coimbra de Araújo
UTFPR – *Campus* Medianeira
(Orientador)

Prof. Jorge Aikes Júnior
UTFPR – *Campus* Medianeira
(Responsável pelas atividades de TCC)

Prof. Paulo Lopes de Menezes
UTFPR – *Campus* Medianeira
(Convidado)

Prof. Jorge Aikes Júnior
UTFPR – *Campus* Medianeira
(Convidado)

A folha de aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

Para minimizar o problema de comunicação entre o banco de dados relacional e a programação orientada a objetos, utilizam-se ferramentas de Mapeamento Objeto Relacional, cujo o objetivo é possibilitar uma transparência na comunicação entre a aplicação e o seu banco de dados. O Entity Framework é o ORM (Object-Relational Mapping) da Microsoft, que faz essa comunicação entre esses dois modelos distintos, diminuindo o tempo de programação, focando no desenvolvimento e regras de negócio. Esse trabalho demonstra a utilização desse ORM, criando o banco de dados, seus relacionamentos, carregamentos de informações do banco de dados e a personalização das tabelas utilizando o Code First em C# com classes POCO (Plain OLD CLR Object).

Palavras chave: Entity Framework, ORM, Code First.

ABSTRACT

To minimize the problem of communication between the relational database and object-oriented programming, use Object-relational Mapping tools, whose goal is to enable transparent communication between the application and its data pack. The Entity Framework is Microsoft's ORM (Object-Relational Mapping) that makes that communication between these two different models, reducing programming time, focusing on the development and business rules. This work demonstrates the use of this ORM, creating the database, your relationships, database information shipments and customizing tables using the Code First in C # with POCO (Plain OLD CLR Object) classes.

Keys words: Entity Framework, ORM, Code First.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
ADO	*ActiveX Data Objects*
CRUD	*Create, Read, Update e Delete*
C#	*C sharp*
C++	*C plus plus*
DDL	*Data Definition Language*
DLL	*Dynamic-link library*
DML	*Data Manipulation Language*
EF	*Entity Framework*
GMT	*Greenwich Mean Time*
IBM	*International Business Machines*
ID	Identificador
IDE	*Integrated Development Environment*
LINQ	*Language INtegrated Query*
ORM	*Object-Relational Mapping*
POCO	*Plain OLD CLR Object*
POO	Programação Orientada a Objetos
SEQUEL	*Structured English QUEry Language*
SQL	*Structured Query Language*
TPC	*Table Per Concrete Type*
TPH	*Table Per Hierarchy*
TPT	*Table per Type*

SUMÁRIO

1 INTRODUÇÃO

O modelo relacional arquiteta todos os dados em linhas (sendo cada linha um novo registro) e colunas (campos que descrevem os dados de cada registro). Se os dados forem complexos para serem representados em uma grade bidimensional, são criadas tabelas adicionais para englobar as informações relacionadas. Cada tabela em um esquema relacional abrangerá alguns (não todos) os dados para uma grande quantidade de registros. Já o modelo de dados Orientado a Objeto não está limitado a manter as informações em linhas e colunas como no modelo relacional, o desenvolvedor cria uma definição (um modelo) que descreve completamente uma determinada classe. Cada registro (objeto) é uma instância específica daquela classe, sendo assim, cada objeto contém todos os itens de informação para um único registro. As definições de classe também podem incluir trechos de programação (métodos) que operam sobre os dados descritos pela classe. Segundo Elmasri e Navathe (2011, p. 304), essa diferença entre esses dois modelos é denominada de impedância.

Para reduzir essa impedância, foi criada uma técnica de desenvolvimento chamada de ORM (Object-Relational Mapping), que tem por objetivo mapear os dados da aplicação para o banco de dados relacional e realizar consultas SQL (Structured Query Language) nesse banco de dados para que sejam obtidas informações persistidas no modelo relacional. Tanto a persistência como a consulta, quem realiza é a ferramenta de Mapeamento Objeto Relacional, que visa facilitar o desenvolvimento de uma aplicação e focar nas negras de negócio.

O ORM Entity Framework é uma das principais ferramentas de persistência presentes na plataforma .NET, sendo uma parte integrante do pacote de tecnologias ADO.NET, possibilitando soluções para reduzir problemas na comunicação entre aplicação e banco de dados relacional - realizando a abstração em vários pontos. Também fornece recursos que possibilitam o aumento de produtividade no desenvolvimento da aplicação, buscando assim minimizar o tempo de produção e manutenção do código.

Essa pesquisa demonstra, em códigos, o uso do Entity Framework a partir da linguagem de programação C#, como realizar os carregamentos de dados, associações, heranças e migrações com classes POCO (Plain OLD CLR Object) em Code First, focando nos métodos e objetos dessas classes.

1.1 OBJETIVO GERAL

Programar Orientado a Objetos com Banco de Dados Relacional SQL Express da Microsoft, demonstrando como utilizar a técnica de ORM fazendo uso do Entity Framework, buscando compreender suas particularidades para a resolução da impedância objeto-relacional.

1.2 OBJETIVOS ESPECÍFICOS

Disponibilizar e compartilhar de forma livre e aberta as particularidades da ferramenta ORM Entity Framework, abordando seus principais tópicos:

- Apresentar o conceito de ORM (Object-Relational Mapping);
- Apresentar o conceito do Entity Framework;
- Apresentar o LINQ (Language-Integrated Query);
- Aplicar Entity Framework com classes POCO (Plain OLD CLR Object) e Code First;
- Apresentar e demonstrar tipos de associações;
- Apresentar e demonstrar o Migrations;
- Apresentar e demonstrar personalização utilizando Entity Framework.

1.3 JUSTIFICATIVA

Quando se pensa em programação, em muitos casos, terá um banco de dados em algum ponto dela. Banco de Dados Relacional e Programação Orientada a Objetos são dois paradigmas distintos, e para que haja uma comunicação necessitam de algumas técnicas. Entity Framework é uma delas para a plataforma .NET. Ele possibilita aos desenvolvedores trabalhar com dados na forma de propriedades e objetos específicos, sem ter que relacionar com as tabelas do Banco de Dados e as colunas onde os dados estão armazenados. O Entity Framework tem como

propósito focar na programação e abstrair o Banco de Dados, deixando o framework responsável pela inserção, alteração e consulta nessa aplicação, alcançando dois objetivos distintos: modelar entidades, relacionamentos e trabalhar com sistemas de armazenamento de dados para armazenar ou consultar informações.

1.4 ESTRUTURA DO TRABALHO

O capítulo 2 apresenta o referencial teórico abordado para o desenvolvimento desse trabalho, trazendo as referências dos autores citados e extraindo suas linhas de raciocínio para o embasamento desse documento.

No capítulo 3 são descritos os materiais e métodos utilizados para a aplicação do ORM Entity Framework, demonstrando qual a linguagem utilizada, SQL Express como banco de dados, a IDE (Integrated Development Environment) que foi desenvolvida o código, as estratégias existentes no Entity Framework e demonstrando como configurar a aplicação para utilizar esse ORM.

No capítulo 4 são demonstrados os resultados e discussões obtidos na aplicação do Entity Framework, exibindo suas particularidades e o código necessário para realizar uma aplicação com esse ORM, com comentários e dissertações cabíveis para cada ação.

Finalizando, o capítulo 5 apresenta as considerações finais, demonstrando a conclusão desse trabalho, sugestões e possíveis estudos futuros.

2 REFERENCIAL TEÓRICO

O conceito do modelo relacional surgiu baseado no conceito de relação matemática, parecido com uma tabela de valores como afirma Elmasri e Navathe (2011, p.38). Para Silberschataz (2006, p.25) "Um banco de dados relacional consiste em uma coleção de tabelas, cada uma das quais com um nome único. [...] Uma linha em uma tabela representa um relacionamento entre um conjunto de valores". Para cada tabela existe uma identificação única comumente chamada de chave primária, segundo Machado e Abreu (2002, p.184) chave primária (Primary Key) ”[...] designa o conceito de item de busca, ou seja, um dado que será empregado nas consultas à base de dados. É um conceito lógico da aplicação” e as chaves estrangeiras (Foreign Key) têm o objetivo de ligar uma tabela com outra como afirma Machado e Abreu (2002, p.184) “[...] constituem um conceito de vital importância no modelo relacional. São os elos de ligação entre as tabelas”. Dentro do esquema de banco de dados há duas diferenças entre: esquema de banco de dados e instância de banco de dados, onde o esquema de banco de dados é o projeto lógico do banco de dados e a instância de banco de dados é o instantâneo dos dados no banco de dados em um determinado tempo, Silberschatz (2006, p.27).

Para acessar os dados no banco de dados é utilizado uma linguagem de banco de dados, para Rob e Coronel (2011, p. 240), em tese, uma linguagem de banco de dados “[...] permite a criação de bancos e estruturas de tabelas para executar tarefas básicas de gerenciamento de dados (adicionar, excluir e modificar) e consultas complexas, transformando dados brutos em informações úteis”, essa linguagem também deve executar essas funções básicas sem que haja um esforço maior do usuário, tendo uma estrutura e sintaxe de comandos fáceis de aprender, também deve ser portátil, no sentido de adequar-se a um padrão básico para que os usuários não tenham que aprender novamente o básico quando passarem de um banco de dados para outro, a SQL (Structured Query Language) é a linguagem que melhor atende essas exigências. Segundo Elmasri e Nnavathe (2011, p.57) “O nome SQL hoje é expandido como Structure Query Language[...] SQL era chamado de SEQUEL (Structured English QUEry Language) e foi criada e implementada na IBM Research como a interface para um sistema de banco de dados relacional [...]”. Para Date (2004, p.71) “[...] a SQL é uma linguagem padrão para se lidar com o banco de dados relacionais, e é aceita por quase todos os produtos existentes no mercado”. Para Rob e Coronel (2011, p. 240) a SQL se adequa em duas categorias: Linguagem de Definição de Dados (DDL) – inclui comando para criar objetos de banco de dados como tabelas, índices e visualizações, também define direito de acesso; e Linguagem de Manipulação

de Dados (DML) – possui comando para inserir, atualizar, excluir e recuperar os dados em tabelas de banco de dados.

No banco de dados há um conceito de *ACID* – Atomicidade, Coerência, Isolamento e Durabilidade que tem por objetivo gerenciar as transações no banco de dados. Para Khoshafian (1994, p.249) a ACID apresenta as seguintes definições: a atomicidade, significa que a transação ou é executada inteiramente ou não é executada; a coerência significa que as transações mapeiam um banco de dados persistente de um estado coerente para outro; o isolamento significa que as transações não leem resultados intermediários de outras transações não efetivadas e a Durabilidade trata que uma transação é efetivada e garante que seus efeitos serão suportados apesar das falhas.

A principal finalidade dos sistemas de banco de dados para Silberschatz (2006, p. 02) é manter as informações organizadas, reduzir as redundâncias e inconsistência de dados – ter as mesmas informações duplicadas em várias arquivos e várias cópias dos mesmos dados; retirar a dificuldade de acesso a dados – os dados necessários sejam recuperados de uma forma conveniente e eficaz; isolar os dados – retirar a dispersão de vários arquivos e esses arquivos podem estar em diferentes formatos; reduzir o problema de integridade – os valores de dados armazenados precisam satisfazer determinadas restrições de consistência; problema de automaticidade – os dados sejam restaurados ao estado consistente em que se encontravam antes de uma falha sistêmica; problema de segurança – permitir acesso somente as pessoas autorizadas para que acessem os dados.

O SQL Express é um banco de dados relacional, gratuito e potente, segundo seu site oficial da ferramenta:

> "Este pacote contém o banco de dados principal do SQL Server, bem como as ferramentas para gerenciar as instâncias do SQL Server, incluindo SQL Server Express, LocalDB e SQL Azure. Se você precisar de Reporting Services ou de uma pesquisa de texto completo, use o SQL Server Express com Advanced Services."

Atualmente utiliza-se o Banco de Dados Relacional junto com a Programação Orientada a Objetos com o objetivo de resolver problemas do cotidiano em aplicações comercias.

A Programação Orientada a Objetos (POO) possui uma estrutura geral distinta da estrutura de um programa equivalente em linguagem procedural. Em POO o foco é escrever código de fácil manutenção e que possam ser reutilizados, trazendo os objetos da vida real para

representar uma entidade no programa, os objetos são instâncias de uma classe conforme os autores Robinson, Allen, Cornes, Glynn, Greenvoss, Harvey, Nagel, Skinner, Watson (2004, p.1041). Essa classe é um modelo que representa uma entidade e possui membros como variáveis (atributos - características desse objeto e determinam o estado do mesmo) e funções (operações - ações que acontecem a um objeto). O autor Galuppo (200, p.131), define a orientação a objetos como:

> "A orientação a objetos é um paradigma fundamentado em trazer os objetos da vida real para representar uma entidade na aplicação por meio de um tipo bem definido. Os objetos propriamente ditos são instâncias de uma classe. A classe, principal símbolo da orientação a objetos, é um modelo que representa uma entidade, ou seja, a estrutura de um objeto é baseada em uma classe. Por exemplo, uma classe Pessoa é um modelo para um ou mais objetos, sendo que cada um desses tem sua área em memória. Uma classe é formada por membros, sendo os principais os atributos (variáveis) e as operações (funções). Os atributos determinam o estado do objeto, por exemplo, os campos Nome e Idade da classe Pessoa representam o nome e a idade da pessoa, respectivamente. Os atributos são as características que um objeto possa assumir. As operações são as funcionalidades que um objeto pode exercer: A função Falar da classe Pessoa, por exemplo, representa o ato de falar da pessoa. Desse modo, as operações são ações que acontecem a um objeto. Porém, esse é o conceito básico e introdução ao paradigma da orientação a objetos, sem se prender aos detalhes que a ela pertencem"

No conceito de Orientação a Objeto existe o objeto que é qualquer coisa que seja identificável em um único item material e a classe que é uma definição genérica de um objeto, de acordo com Robinson et al (2004, p. 1043). Na Orientação a Objetos existem dentro da classe métodos, que são blocos de funcionalidade do que a classe pode realizar e propriedades que é um método ou par de métodos que são expostas para o mundo externo de acordo com Robinson et al (2004, p. 1055).

Para Jandl (2007, p.76) a Orientação a Objetos é "[...] uma técnica de programação que se baseia na construção e utilização de objetos. Um objeto, ao combinar dados e operações específicas delimita um conjunto particular de funcionalidades ou responsabilidades." De acordo com Jandl (2007, p.76) classes, objetos e instanciação é um conceito fundamental quando se trata de Orientação a Objetos. Uma classe é um modelo para um novo tipo de objeto relacionando suas características e funcionalidades, pode representar uma entidade real ou

abstrata. Cada classe possuem seus campos, atributos ou variáveis-membros necessários que são destinadas a armazenar dados que caracterizam um objeto e suas partes. Dentro da classe possuem seus métodos, operações ou funções-membro que são sub-rotinas associadas aos objetos, os métodos são trechos de código que possibilitam realizar alguma ação ou transformar os valores dos atributos dessa classe. Também dentro de cada classe existe um construtor que são métodos cujo seu objetivo é de inicializar e preparar novos objetos durante sua instanciação, podem receber parâmetros para caracterizar esse objeto na sua criação, porém só podem ser acionados por um operador específico que são responsáveis pela criação de novos objetos.

A linguagem C# é uma linguagem de programação totalmente orientada a objetos e suporta todos os conceitos da orientação a objetos. C# permite desenvolver e criar uma gama de aplicativos robustos e seguros, como aplicativos de cliente do Windows, serviços Web XML, componentes distribuídos, aplicativos de cliente-servidor, aplicativos de banco de dados que são executadas no .NET Framework. Sua sintaxe é reconhecível para programadores familiarizados com as linguagens de C, C++ e Java. Segundo o site oficial da linguagem:

> "C# suporta os métodos e tipos genéricos, que fornecem maior segurança e desempenho de tipo, e iteradores, que permitem implementadores de classes de coleção para definir os comportamentos personalizados de iteração que são simples de usar por código do cliente. as expressões de LINQ (Consulta Integrada à Linguagem) tornam a consulta fortemente tipada em uma construção de linguagem de primeira classe."

O processo de compilação do C# é mais simples em comparação ao C, C++ e Java. Não contém nenhum arquivo de cabeçalho separado e nenhum requisito de que métodos e tipos sejam declarados em uma ordem específica. Um arquivo de origem C# pode definir qualquer número de classes, estruturas, interfaces e eventos, segundo o site oficial.

Um conceito existente na Orientação a Objetos é de heranças, de acordo com Lippman (2002, p. 105) a herança "[...] permite agrupar classes em famílias de tipos relacionados, possibilitando o compartilhamento de operações e dados comuns. [...] A herança define um relacionamento de pai/filho". O pai é a interface pública e implementação privada que são comuns a todos os seus filhos e cada filho adiciona ou sobrescreve o que herda, implementando seu próprio comportamento, como afirma Lippman (2002, p. 105). Para Galuppo (2004, p.131) a herança é definida como "[...] um relacionamento pai/filho entre tipos, o que origina a hierarquia das classes." O objetivo do conceito de herança é a reutilização do código que

permite definir uma funcionalidade comum na classe pai que possa ser usada e possivelmente alterada pelas classes filhas, conforme o autor Troelsen (2009, 167). Para Jandl (2007, p. 108) a herança é "[...] um mecanismo que possibilita a uma classe usar campos e métodos definidos em outra classe, que significa o compartilhamento de membro entre classes. A herança é baseado em um relacionamento hierárquico [...]" dessa forma, um pai, ou seja, a classe de referência empresta suas definições para as classes definidas como filhas.

Um dos grandes obstáculos entre o modelo de banco de dados e o modelo de linguagem orientada a objetos é a obtenção, inserção, alteração ou exclusão dos dados, visto que esses dois modelos possuem paradigmas distintos. O banco de dados trabalha com linhas (tuplas - cada tupla possui um valor) e colunas (identificações, atributos) e a orientação a objetos trabalha com classes, atributos e métodos. Essa diferença entre os dois modelos é denominada impedância. Segundo o Elmasri e Navathe (2011, p. 304) "A divergência de impedância é o termo usado para se referir aos problemas que ocorrem devido às diferenças entre modelo de banco de dados e o modelo da linguagem de programação".

O primeiro problema que pode acarretar é que os tipos de dados da linguagem de programação são diferentes dos tipos de dados de atributo que estão disponíveis no modelo relacional, sendo assim, é necessário ter um vínculo para cada linguagem de programação hospedeira que especifica, para cada tipo de atributo os tipos de linguagem de programação que são compatíveis. Um vínculo diferente é necessário para cada linguagem de programação, pois diferentes linguagens possuem diferentes tipos de dados. O segundo problema ocorre pois os resultados das maiorias das consultas são conjuntos de linhas - tuplas, e cada linha é formada por uma série de valores de atributos, é necessário que haja um vínculo para que seja mapeada a estrutura de dados do resultado da consulta, que é uma tabela, para que haja uma estrutura apropriada para a linguagem de programação orientada a objetos conseguir ler os dados. Para facilitar essa comunicação entre esses dois modelos foram criados técnicas de ORM (Object-Relational Mapping).

A proposta de arquitetar o ORM atribuiu a necessidade de sustentar as carências do modelo de desenvolvimento relacional, proporcionando uma interface Orientada a Objetos de forma transparente, para modelagem ser feita ao paradigma que melhor se adequar (Stonebraker, 1996 apud Soares 2006). Para Bauer (2005, 31) "[...]o mapeamento objeto/relacional é a persistência de objetos automatizados (e transparente) dentro de um aplicativo [...]". Um ORM dispõe diversos métodos básicos que irão realizar a interação entre a aplicação e o banco de dados, responsabilizando por algumas tarefas básicas, como por

exemplo um CRUD - Create, Read, Update e Delete. Também gerencia detalhes de mapeamento para um conjunto de objetos para um banco de dados relacional. Reduz a necessidades de escrever consultas SQL e códigos de conexão, possibilitando um código mais elegante e facilitando sua manutenção.

O Entity Framework (EF) é uma das principais ferramentas de persistência de dados presente na plataforma .NET, que pode ser utilizado com a linguagem de programação C# ou Visual Basic, dentre outras. É uma evolução do pacote de tecnologias ADO.NET. A Microsoft criou essa ferramenta ORM, Open Source, denominada Entity Framework, com o objetivo de suportar o desenvolvimento orientado a objetos e realizar essa comunicação entre a aplicação que está sendo desenvolvida e um banco de dados relacional. Segundo o site oficial da Microsoft, o Entity Framewok:

> "[...]é uma ferramenta de mapeamento objeto relacional (ORM – Object Relational Management), que permite aos desenvolvedores trabalhar com classes (entidades) que correspondem a tabelas em um banco de dados, tornando transparente o acesso a estes dados e principalmente, eliminando a necessidade de escrever código de banco de dados (SELECT, INSERT, UPDATE, DELETE) na aplicação."

Esse ORM trabalha com convenções próprias, como nomenclatura de ID, tipos de dados dos campos da tabela e que podem ser alteradas de acordo com sua necessidade, conforme o site oficial do Entity Framework.

O EF permite aos desenvolvedores trabalhar com dados na forma de objetos e propriedades específicas de domínio, retirando o foco das tabelas no banco de dados subjacentes e colunas onde os dados são armazenados e focando na regra de negócio, permitindo desenvolver em um nível mais alto de abstração. O EF possibilita que o desenvolvedor mantenha sua aplicação com menos código do que com aplicações tradicionais, pois elimina código de acesso ao banco de dados, conforme o site oficial do Entity Framework. Para consultar os dados persistidos no banco de dados, utiliza-se o LINQ (Language-INtegrated Query) que já vem incorporado na plataforma .NET.

O Language-INtegrated Query (LINQ) é um conjunto de recursos introduzidos no Visual Studio 2008, que aumenta os recursos avançados de consulta à sintaxe das linguagens de programação C# e Visual Basic. Seu objetivo é simplificar a situação em que os desenvolvedores tem que aprender uma nova linguagem de consulta para cada tipo de fonte de

dados ou formato de dados (SQL, XQuery) que devem oferecer suporte, oferecendo um modelo mais fácil e consistente para trabalhar com dados em vários tipos de fontes de dados/formatos. Em uma consulta LINQ sempre se trabalha com objetos. Para Stellman (2008, p. 561) "O LINQ funciona com praticamente todos os tipos de fontes de dados utilizáveis no .NET. Seu código precisa de um using System.Linq; no início do arquivo[...] o IDE automaticamente coloca uma referência à LINQ no início de cada arquivo[...]" e para Troelsen (2009, p. 434) o LINQ é:

> "[...]é um conjunto de tecnologias relacionadas que tenta fornecer uma matéria única e simétrica de interagir com diversas formas de dados. [...] O LINQ pode interagir com qualquer tipo que implemente a interface IEnumerable<T>, incluindo arrays simples, bem como coleções de dados genéricas e não genéricas."

O LINQ consistem em três diferentes ações: obter a fonte de dados, criar consultas e executar consultas. No C# como a maioria das linguagens de programação uma variável deve ser declarada antes de ser usada. Em uma consulta LINQ, a primeira etapa é especificar a fonte de dados. Uma consulta especifica quais informações devem ser recuperadas da fonte de dados também especifica como essas informações devem ser classificadas agrupadas e moldadas antes de ser retornada. Uma consulta é armazenada em uma variável de consulta e inicializada com uma expressão de consulta. A expressão de consulta contém três cláusulas: *select* – cláusula que especifica os tipos dos elementos retornados, *from* – cláusula que especifica a fonte de dados e *where* – cláusula que aplica o filtro. Em LINQ, a variável de consulta não faz nada e não retorna nenhum dado, somente armazena as informações necessárias para produzir os resultados quando a consulta é executada em um momento posterior, de acordo com o site oficial da Microsoft.

O LINQ utiliza expressões *lambda* que é uma maneira conveniente de escrever um código que teria que ser gravados no formulário mais complexo, como um método anônimo, um delegado genérico ou uma árvore de expressão. No C# o operador lambda é representado pelo igual maior (=>), que é lido como "vai para", segundo o site oficial da Microsoft.

Com base no site oficial da Microsoft, para utilizar o EF existem três estratégias: Model First, Database First e Code First. Cada estratégia é uma alternativa para poder trabalhar com Entity Framework.A primeira estratégia, Model First, permite modelar a base de dados de forma visual por meio dos próprios recursos do Visual Studio, fazendo uso de um arquivo .EDMX e a geração de um script para criar ou atualizar a base de dados. Utilizando o Model First retira-

se a necessidade de conhecimentos específicos da estrutura do banco de dados e como o próprio EF funciona, porém, pode acarretar algumas dificuldades em relação a manutenção e alterações na regra de negócio da aplicação.

O Database First é uma alternativa de engenharia reversa do EF, que permite criar um Entity Data Model (classes, propriedades e DbContext) a partir de um banco de dados já existente. É ideal para desempenho e esquemas já existentes, a segurança a nível de banco de dados é melhor ajustada. Esse método é complexo para determinar mudanças incrementais tanto do banco de dados quanto da aplicação e também é totalmente dependente da tecnologia do banco de dados.

O Code First é uma abordagem para criar classes POCO (Plain Old CRL Objects - padrão em que classes são definidas com uma estrutura simples, codificadas de maneira a não possuir uma ligação direta com qualquer tipo de framework ORM) e a partir do código escrito é gerado o banco de dados. Com o CodeFirst é possível controlar todos os aspectos do mapeamento das classes com o banco de dados, desde o nome da tabela no banco, obrigatoriedade dos campos, tamanho, dentre outras.

O Entity Framework possui o recurso denominado *Logging*. Há todo momento o ORM envia comandos para o banco de dados, estes comandos podem ser interceptados pelo código da aplicação. Isto é mais comumente utilizado para o log do SQL, mas também pode ser utilizado para exibir as modificações que o ORM realizou ou exibir comandos abortados. Quem realiza as ações é a propriedade *DbContext.Database.Log* e pode ser definido como um delegado para qualquer método que aceita uma cadeia de caracteres - *string*. Quando essa propriedade é definida na aplicação, possibilita exibir o SQL para todos os diferentes tipos de comandos (inserções, atualizações e exclusões gerado como parte do método *SaveChanges()* do EF), parâmetros, comandos que são executados de forma assíncrona, *timestamp* indicando quando o comando começou a ser executado, comandos que foram concluídos com êxito ou gerou uma falha lançando uma exceção, a indicação do valor do resultado, a quantidade aproximada de tempo que levou para executar o comando, como afirma o site oficial da Microsoft.

O Entity Framework quando se utiliza a abordagem Code First, possibilita realizar o processo de atualização automática do banco de dados através do pacote denominado Migrations. Esse pacote gere atualizações gerenciáveis no banco de dados, possibilita que o próprio Migrations gerencie de forma automática as atualizações onde o Migrations que se encarrega de atualizar o banco (ao executar a aplicação, isto será feito de maneira automática)

ou delegue para o programador realizar as migrações, possibilitando aplicar as últimas modificações ou voltar em alguma migração que foi realizada. Dessa forma mantém o banco de dados sempre atualizado com as classes, como afirma o site oficial da Microsoft.

3 MATERIAL E MÉTODOS

Para a elaboração desse Trabalho de Diplomação foram utilizados, como material, a linguagem de programação orientada a objetos C#, para a criação e elaboração das classes propostas no projeto prático. As classes utilizadas foram: Pessoa, Aluno, Professor, Endereço, Turma e Disciplina. A criação de uma pasta no projeto para organizar o mesmo, pasta *poco*: contém as classes como Aluno, Professor, Endereço e demais classes que compõe esse projeto. A pasta *controlador*: cada classe possui um controlador para realizar suas operações no banco de dados. A pasta *contexto*: nessa pasta, onde encontra-se a classe *context* do projeto que o Entity Framework utiliza para criação do banco de dados, tabelas, atributos. Criação de um projeto na mesma *Solution*, denominada *UiConsole*, com o objetivo de realizar as inserções, alterações e exibições a partir de uma classe *main*.

A utilização de herança para que as classes Aluno e Professor herdem da super classe Pessoa os atributos como Nome e CPF, demonstrando o conceito de herança da Orientação a Objeto.

A utilização de três tipos de associações como um-para-um, um-para-muitos e muitos-para-muitos entre as classes, com o objetivo de demonstrar a utilização dessas associações na aplicação do projeto.

A utilização de processo de carga tardia para o carregamento das informações do banco de dados para a aplicação, com o objetivo de verificar se as informações estão sendo persistidas no banco de dados.

A utilização de Logging para que seja visualizada o que é feito no banco de dados, com o objetivo de demonstrar como utilizar esse recurso disponível no Entity Framework.

A utilização de Migrations para que demonstre a possibilidade de atualizar o banco de dados conforme são atualizadas as classes na aplicação.

A utilização do LINQ (Language-INtegrated Query) para a manipulação como inserir, alterar, excluir e realizar a listagem de dados obtidos por meio do Entity Framework nas classes realizadas em C#. A utilização do LINQ foi direcionada em inserir e buscar as inserções realizadas no banco de dados.

O conceito de POCO (Plain OLD CLR Object) para as criações das classes com a estratégia Code First (estratégia adotada para a criação do banco de dados utilizando linhas de código). A abordagem dessa estratégia foi com o intuito de demonstrar a possibilidade da

criação do código sem nenhum vínculo com algum ORM, as outras estratégias não foram abordadas ou testadas nesse trabalho.

A IDE (Integrated Development Environment) Visual Studio na versão 2013 foi instalada na máquina com a versão do Windows 8.1 para a realização do código proposto como a criação das classes, instalação do Entity Framework no projeto e exibição do banco de dados na própria IDE.

A utilização do banco de dados SQLExpress para a criação da base de dados do projeto. Esse banco é nativo da IDE, o intuito foi demonstrar como seria as alterações no banco de dados relacional.

A utilização do Astah Community para a criação do diagrama proposto, permitindo criar uma figura para a exemplificação das classes propostas. A utilização dessa ferramenta foi com o objetivo de utilizar uma ferramenta de já conhecimento do discente.

Para instalar o EF, há duas maneiras: via NuGet e Package Manager Console. Essas duas maneiras buscarão na rede uma referência dessa DLL (Dynamic-Link Library), baixar essa DLL no projeto e fazer uma referência no projeto a essa DLL. Para isso, a instalação dá-se da seguinte maneira: *Tools > NuGet Package Manager > Package Manager Console* ou *Tools > NuGet Package Manager > Manage NuGet for Solution*. A diferença dessas duas maneiras é que em uma é visual (Manage NuGet for Solution) e a outra é via comandos (Package Manager Console), que é *Install-Package EntityFramework*. Após a conclusão da instalação, é possível verificar a referência adicionada em *References,* no projeto.

Para habilitar o Migrations no projeto, deve-se abrir o *Package Manager Console* e digitar o comando *Enable-Migrations* ativando esse recurso no projeto. O EF cria no projeto uma pasta Migrations com duas classes: a *(HASH)_InitialCreate.cs* (possui as configurações que já estão no banco de dados) e a classe *Configuration.cs* (possui suas configurações e o método *Seed()* que serve para povoar o banco de dados). Para adicionar a migração deve-se utilizar o comando *Add-Migrations* seguido de um nome no *Package Manager Console*, e na pasta Migrations tem-se a migração criada. É necessário realizar essa atualização no banco de dados com o comando *Update-Database* no *Package Manager Console*.

Pode-se usar o Migrations para gerar um script para rodar no ambiente do cliente. No Package Manager Console digite o comando: *Update-Database -Script -SourceMigration:$InitialCreate -TargetMigration:"NomeDaMigração"* e enter para gerar um script com as mudanças desde a criação (InitialCreate) até a última alteração realizada (NomeDaMigração).

4 RESULTADO E DISCUSSÕES

Nesse capítulo é apresentado o conteúdo prático realizado, seguindo o modelo da Figura 1:

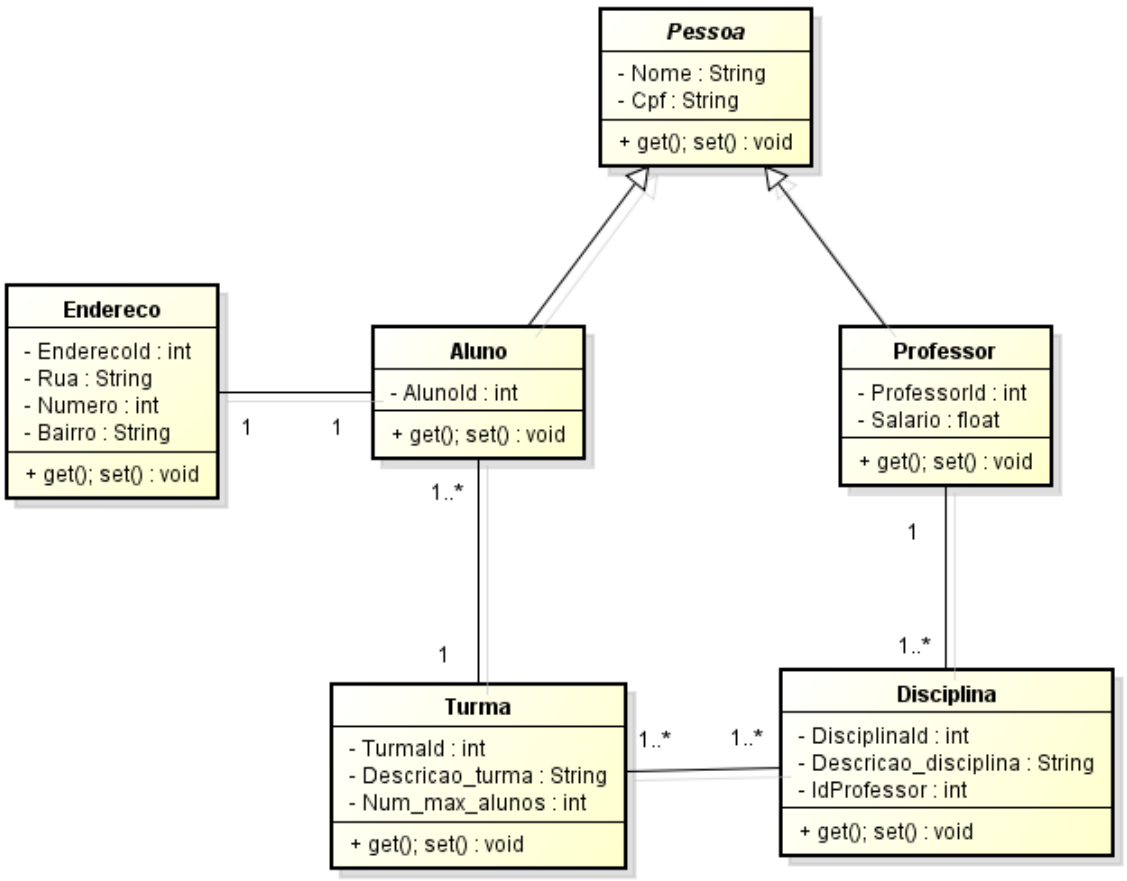

Figura 1 – Diagrama de classes do modelo de negócio
Fonte: do Autor

Para realizar o mapeamento das classes propostas, é necessário implementar uma classe de contexto com a base de dados, que estende de *DbContext* (System.Data.Entity). Essa classe é responsável por gerenciar e acompanhar alterações em instâncias de classes no referido modelo. Ela também é encarregada de gerenciar a conexão, configurações e o contexto de transação das operações com o banco de dados. Veja a Figura 2:

```
using SistemaBasicoInstituicao.poco;
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SistemaBasicoInstituicao.contexto
{
    13 references
    public class TccContext : DbContext
    {
        6 references
        public TccContext() : base("bancoaplicacao") { }
        2 references
        public DbSet<Endereco> Enderecos { get; set; }
        4 references
        public DbSet<Aluno> Alunos { get; set; }
        7 references
        public DbSet<Professor> Professores { get; set; }
        10 references
        public DbSet<Turma> Turmas { get; set; }
        7 references
        public DbSet<Disciplina> Disciplinas { get; set; }
        1 reference
```

Figura 2 - Classe de Contexto
Fonte: do Autor

Note, na figura anterior, que para nomear o banco de dados sem que seja gerado automaticamente pelo EF, no construtor da classe contexto, utiliza-se a instrução *base("NomeDoBanco")*, que se encarrega de nomear o banco. Caso não tenha no construtor essa instrução inserida o banco de dados fica com o mesmo nome do projeto. A propriedade *DbSet<>* tem a função de mapear as classes para o banco de dados e vincular a um objeto, ela gera uma tabela ou utiliza no modelo relacional, mapeada da classe utilizada em sua declaração em *DbSet*.

4.1 PERSONALIZAÇÕES NO ENTITY FRAMEWORK

Todas as personalizações tais como mudar o nome do ID das tabelas, indicar tamanho máximo do campo, indicar se o campo é obrigatório, entre outras, devem ser realizadas sobrescrevendo o método *OnModelCreating*. O EF cria os campos na tabela de forma automática e já atribui os tipos de cada campo baseado nas classes. Para modificar essa convenção realiza-se configurações nesse método, por exemplo, um atributo que é *string* no modelo orientado a objetos, no banco dados ele é criado sendo *NVarchar(max)* ao invés de

varchar. Para personalizar cada campo deve-se indicar o tipo dessa propriedade e a mudança que deseja realizar na criação do banco, conforme Figura 3:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder
       .Properties<string>()
       .Configure(c => c.HasColumnType("varchar"));

    modelBuilder
        .Properties<int>().Configure(c => c.HasColumnType("int"));
```

Figura 3 - Classe Contexto – Trecho do método OnModelCreating()
Fonte: do Autor

Para implementar esse mapeamento para *varchar* utiliza-se o objeto *modelBuilder*, o método *Properties<tipo>()* e, por meio do método *Configure()*, realiza-se o mapeamento para *varchar*. Esse processo tem que ser realizado para todos os tipos de propriedade que se necessite alterar.

O Entity Framework possui uma convenção em que os IDs das classes devem ser identificados a partir da palavra ID. Dessa forma se não for realizada alguma configuração informando ao ORM da alteração de nome de ID, ele não reconhece nenhum campo como sendo o ID da classe. No modelo apresentado na Figura 1 todos os IDs estão nomeados com o nome da classe e em seguida a palavra Id. Para realizar essa configuração na nomenclatura seguindo o modelo proposto no diagrama, deve-se usar o objeto *modelBuilder*, invocar o método *Properties()*, realizar uma consulta onde recupera-se o nome da classe e adiciona o nome dessa tabela e em seguida insere a palavra Id, por último chamar o método *Configure()* para identificar como chave primária esse novo campo nomeado como apresentado na Figura 4. Essa mudança somente é necessária uma única vez para que cada tabela tenha seu nome de ID personalizado.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder
        .Properties().Where(p => p.Name == p.ReflectedType.Name + "Id")
        .Configure(p => p.IsKey());
```

Figura 4 - Classe Contexto – Fragmento do método OnModelCreating() – ID
Fonte: do Autor

Para todo a propriedade que deseja ser refletido no banco de dados seguindo a forma que foi proposto na regra de negócio apresentada nesse modelo, por exemplo, dizendo se é obrigatório, atribuindo um tamanho máximo, tem que ser realizada da seguinte forma: chamar o *modelBuilder*, em seguida invocar o *Entity<NomeClasse>*, com o *Property* selecionar o atributo dessa classe e em seguida informar se deseja colocar campo não nulo, máximo de tamanho para ser refletido no modelo relacional, conforme na Figura 5.

```
modelBuilder
    .Entity<Professor>()
    .Property(p => p.Nome)
    .IsRequired().HasMaxLength(200);

modelBuilder
    .Entity<Professor>()
    .Property(p => p.Cpf)
    .IsRequired()
    .HasMaxLength(11);

modelBuilder
    .Entity<Professor>()
    .Property(p => p.Salario)
    .IsRequired();
```

Figura 5 – Classe Contexto, fragmento do método OnModelCreating - Alterações no mapeamento nas Propriedades
Fonte: do Autor

Essa configuração tem que ser realizada para todas as propriedades que são requeridas no modelo proposto, para que o EF seja informado e realize essas particularidades em cada campo. Veja a Figura 6, que apresenta duas versões para a tabela mapeada.

	Name	Data Type	Allow Nulls
🔑	ProfessorId	int	☐
	Salario	real	☑
	Nome	nvarchar(MAX)	☑
	Cpf	nvarchar(MAX)	☑

	Name	Data Type	Allow Nulls
🔑	ProfessorId	int	☐
	Salario	real	☐
	Nome	varchar(200)	☐
	Cpf	varchar(11)	☐

Figura 6 - Alterações Realizadas nas Propriedades mapeadas para campos da tabela
Fonte: do Autor

O lado esquerdo da Figura 6 apresenta a tabela com os campos sem nenhum tipo de configuração feita na classe contexto do projeto, nesse caso, todos os atributos que no modelo orientado a objetos são *varchar* ocupam o máximo de tamanho no banco de dados (*NVarchar(MAX))* e podem ser nulos. O lado direito apresenta os campos que foram realizadas as configurações na classe contexto, sendo o campo Nome do tipo *varchar*, limitado em 200 caracteres e não pode ser nulo, o campo CPF também é do tipo *varchar* porém limitado a 11 caracteres e não pode ser nulo e o campo Salario não pode ser nulo.

4.2 OS TRÊS TIPOS DE HERANÇA

Há três abordagens para aplicar o conceito de herança, são elas: Tabela por Hierarquia (TPH - *Table Per Hierarchy*), Tabela por Tipo (*TPT - Table Per Type*) e Tabela por Classes Concretas (TPC - *Table Per Concrete Class*). No modelo proposto a classe Aluno e Professor são subclasses da classe Pessoa e cada abordagem de herança gera no banco um tipo de estrutura para as tabelas mapeadas, conforme na Figura 7.

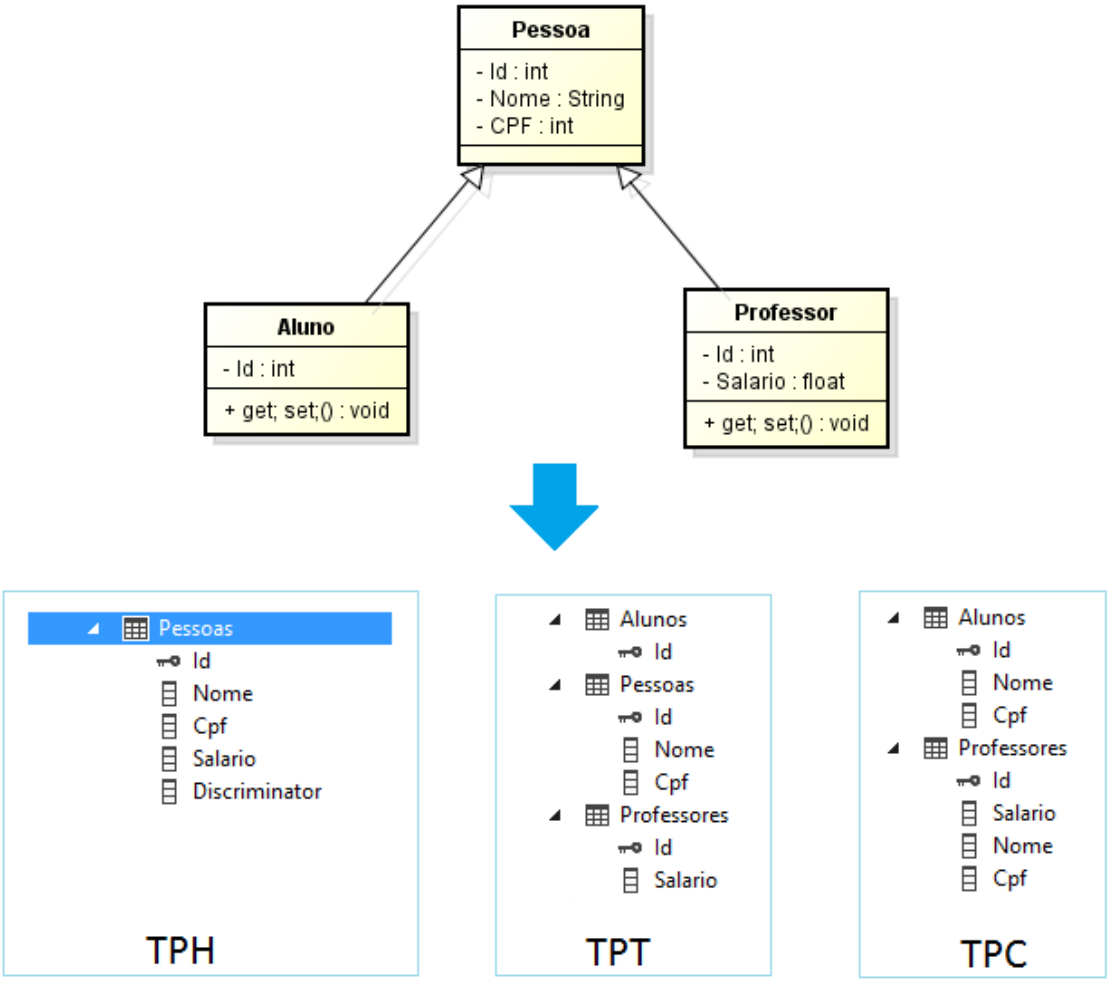

Figura 7 – As abordagens de mapeamento para Heranças
Fonte: do Autor

A abordagem TPH, também é conhecida como *Single Table Inheritance*, referencia a classe base na hierarquia, ou seja a classe Pai, e usa as classes filhas normalmente no projeto. O banco terá uma única tabela e os atributos de cada classe. O EF cria um campo chamado *Discriminator* para distinguir um tipo de objeto do outro, o que não for aplicável para algum *Discriminator* terá o seu valor nulo. Para aplicar o TPH, na classe contexto do projeto adiciona uma propriedade *DbSet*<> da classe Pai, conforme na Figura 8.

```
public class TccContext : DbContext
{
    2 references
    public DbSet<Pessoa> Pessoas { get; set; }

}
```

Figura 8 - Implementação da Herança com a abordagem TPH
Fonte: do Autor

A segunda abordagem, TPT, realiza um mapeamento em que há uma tabela por tipo (uma tabela para cada classe criada), essa herança é indicada para manter o banco de dados normalizado e para utiliza-la deve-se inserir as propriedades *DbSet<>* das classes filhas, utilizar o objeto *modelBuilder*, invocar o método *Entity<>()* com o nome da tabela e em seguida o *ToTable()* no método *OnModelCreating()* do contexto do projeto, como apresentado na Figura 9.

```
public class TccContext : DbContext
{
    5 references
    public DbSet<Aluno> Alunos { get; set; }
    5 references
    public DbSet<Professor> Professores { get; set; }

    0 references
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {

        modelBuilder.Entity<Aluno>().ToTable("Alunos");
        modelBuilder.Entity<Professor>().ToTable("Professores");
    }
}
```

Figura 9 - Implementação da Herança com a abordagem TPT
Fonte: do Autor

A última abordagem, TPC, tem por objetivo construir tabelas com base somente nas classes concretas (subclasses) mantendo na tabela da subclasse os dados da superclasse. Na classe contexto, no método *OnModelCreating()*, utiliza-se o objeto *modelBuilder*, o método *Entity()* com o nome da tabela e o *Map()* para indicar essa herança, como apresentado na Figura 10.

```
public class TccContext : DbContext
{
    5 references
    public DbSet<Aluno> Alunos { get; set; }
    5 references
    public DbSet<Professor> Professores { get; set; }

    0 references
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Aluno>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("Alunos");
        });
        modelBuilder.Entity<Professor>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("Professores");
        });
    }
}
```

Figura 10 - Implementação da Herança com a abordagem TPC
Fonte: do Autor

4.3 TIPOS DE ASSOCIAÇÕES ENTRE AS CLASSES

Uma associação ocorre quando uma classe tem um tipo de relacionamento com outra e, nesse cenário, existem três tipos de associações: um-para-um (*one-to-one*), um-para-muitos (*one-to-many*) e muitos-para-muitos (*many-to-many*), sendo que no um-para-muitos existe o desdobramento para o muitos-para-um. No modelo proposto tem-se uma associação um-para-um entre Aluno e Endereço. Para informar o ORM que há essa relação, deve-se, na classe Aluno, implementar uma propriedade pública e virtual do tipo Endereço. O virtual é utilizado para adicionar uma lógica de consulta dos dados relacionados fazendo um *join* e carregando as propriedades relacionadas, e na classe Endereço não se realiza nenhuma alteração, apresentado na Figura 11.

```
public class Endereco
{
    3 references
    public int EnderecoId { get; set; }
    5 references
    public string Rua { get; set; }
    5 references
    public int Numero { get; set; }
    5 references
    public string Bairro { get; set; }
}
```

```
public class Aluno : Pessoa
{
    2 references
    public int AlunoId { get; set; }
    //one-to-one
    5 references
    public virtual Endereco Endereco { get; set; }

}
```

Figura 11 - Associação um-para-um entre as Classes Endereco e Aluno
Fonte: do Autor

Como pode ser verificado, na Figura 11, a simples implementação de uma propriedade do tipo Endereço já mapeia, nas tabelas o relacionamento entre elas, conforme na Figura 12.

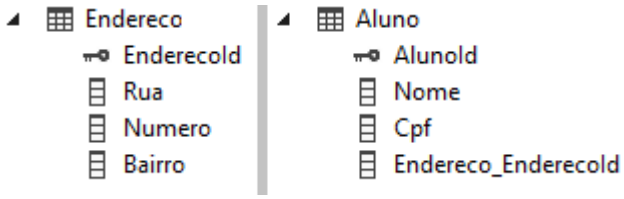

Figura 12 – Relacionamento um-para-um entre as tabelas Endereço e Aluno
Fonte: do Autor

A tabela Aluno possui um atributo do tipo *int*, com o nome de *Endereco_EnderecoId* indicando esse relacionamento um-para-um entre essas tabelas.

Em uma associação um-para-muitos, a classe Professor pode se associar com uma ou mais classes Disciplina. Nesse caso, na classe Professor há um atributo público e virtual do tipo *IEnumerable*<> de Disciplina e na classe Disciplina há um atributo público do tipo Professor, como na Figura 13.

```
public class Professor : Pessoa
{
    12 references
    public int ProfessorId { get; set; }
    6 references
    public float Salario { get; set; }

    //one-to-many
    0 references
    public virtual IEnumerable<Disciplina>
            Disciplinas { get; set; }
}
```

```
public class Disciplina
{
    8 references
    public int DisciplinaId { get; set; }
    4 references
    public string Descricao_disciplina { get; set; }

    //many-to-one
    9 references
    public Professor Professor { get; set; }
}
```

Figura 13 - Associação um-para-muitos entre as Classes Professor e Disciplina
Fonte: do Autor

Como pode ser observado na Figura 13, a implementação dessas propriedades entre as classes Professor e Disciplina mapeia no banco de dados esse relacionamento entre elas, demonstrado na Figura 14.

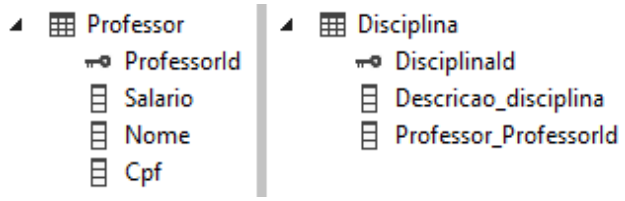

Figura 14 - Relacionamento um-para-muitos entre as tabelas Professor e Disciplina
Fonte: do Autor

A tabela Disciplina possui um atributo do tipo *int* com o nome de *Professor_ProfessorId* indicando esse relacionamento um-para-muitos entre as tabelas Professor e Disciplina.

A classe Turma pode se associar com uma ou mais classes Aluno, ocasionando outra associação um-para-muitos. Na classe Turma há uma propriedade pública e virtual do tipo *IEnumerable<>* de Aluno e na classe Aluno possui uma propriedade pública do tipo Turma, demonstrado na Figura 15.

```
public class Turma
{
    14 references
    public int TurmaId { get; set; }
    4 references
    public string Descricao_turma { get; set; }
    7 references
    public int num_max_alunos { get; set; }

    //one-to-many
    0 references
    public virtual IEnumerable<Aluno>
                Alunos { get; set; }
}
```

```
public class Aluno : Pessoa
{
    2 references
    public int AlunoId { get; set; }

    //many-to-one
    3 references
    public Turma Turma { get; set; }

}
```

Figura 15 - Associação um-para-muitos entre as Classes Turma e Aluno
Fonte: do Autor

À vista disso, o ORM sabe que a classe Turma e Aluno possuem uma associação entre elas, dessa forma, mapeia esse relacionamento um-para-muitos no banco de dados, apresentado na Figura 16.

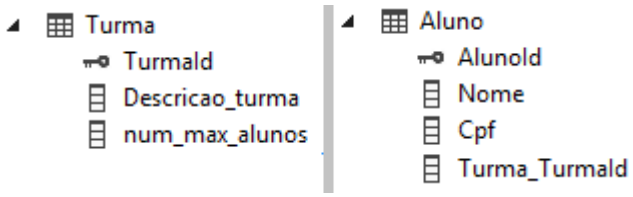

Figura 16 - Relacionamento um-para-muitos entre as tabelas Turma e Professor
Fonte: do Autor

A tabela Aluno possui um atributo do tipo *int* chamado de *Turma_TurmaId* que indica esse relacionamento um-para-muitos entre as tabelas Turma e Aluno.

Quando uma classe A está associada a qualquer número de ocorrências de uma classe B e a classe B está associada a qualquer número de ocorrência da classe A, tem-se uma associação muitos-para-muitos, nesse caso o EF cria uma nova tabela no modelo relacional com os IDs das tabelas que estão se relacionando. Para aplicar essa associação, na classe Disciplina há uma propriedade pública e virtual do tipo *ICollection*<> de Turma e na classe Turma há uma propriedade pública e virtual do tipo *ICollection*<> de Disciplina, como na Figura 17.

```
public class Disciplina
{
    8 references
    public int DisciplinaId { get; set; }
    4 references
    public string Descricao_disciplina { get; set; }

    //many-to-many
    5 references
    public virtual ICollection<Turma>
            Turmas { get; set; }
}
```

```
public class Turma
{
    14 references
    public int TurmaId { get; set; }
    4 references
    public string Descricao_turma { get; set; }
    7 references
    public int num_max_alunos { get; set; }

    //many-to-many
    4 references
    public virtual ICollection<Disciplina>
            Disciplinas { get; set; }
}
```

Figura 17 - Associação muitos-para-muitos entre as Classes Disciplina e Turma
Fonte: do Autor

Como pode ser verificado na Figura anterior, a implementação dessas propriedades nas classes Disciplina e Turma já mapeia nas tabelas o relacionamento entre elas, gerando uma nova tabela, conforme na Figura 18.

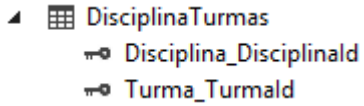

Figura 18 - Relacionamento muito-para-muitos entre as tabelas Ddisciplina e Turma
Fonte: do Autor

Nessa tabela gerada pelo ORM, há somente as chaves primárias das tabelas que estão nesse relacionamento muitos-para-muitos.

4.4 UTILIZANDO LINQ NA APLICAÇÃO

Quando é necessário realizar uma busca ou associação entre as classes é utilizado LINQ, como por exemplo, entre o Aluno e Turma, onde é preciso informar ao Entity Framwork que há alguma relação entre eles. Ao utilizar o LINQ, deve-se primeiro obter a fonte de dados, nesse caso a fonte de dados é o banco de dados que a aplicação já gerou pela classe contexto chamada *TccContext*. Após obter a conexão com o banco de dados, deve-se realizar uma consulta para que o EF saiba em qual tabela será persistido o objeto que foi criado, e por último, deve-se executar essa consulta, nesse ponto pode-se juntar comandos do EF e LINQ para que possam ser realizadas as ações no banco de dados. Por exemplo, para adicionar um novo objeto Aluno, popula esse objeto com as informações que deseja ser refletida no banco dados, conforme a Figura 19.

```
private static void AddAluno()
{
    Endereco e = new Endereco
    {
        Rua = "Ari Barroso",
        Numero = 123,
        Bairro = "Boa Vista"
    };

    AlunoControlador dbAluno = new AlunoControlador();
    TurmaControlador dbTurma = new TurmaControlador();

    Aluno a = new Aluno()
    {
        Nome = "Gustavo Neto",
        Cpf = "77788877778",
        Endereco = e,
        Turma = dbTurma.Listar().Where(t => t.TurmaId == 1).FirstOrDefault()
    };

    dbAluno.Salvar(a);
}
```

Figura 19 - Populando o objeto Aluno
Fonte: do Autor

Na Figura 19, cria-se um objeto Endereço para que seja vinculado ao objeto Aluno, obtém uma conexão para o Aluno e uma conexão para a Turma, atribui valores para os campos como Nome, CPF e adiciona o objeto Endereco a propriedade endereço do Aluno, por último, faz-se uma consulta LINQ onde o ID da turma, que é igual a 1, seja o ID da turma que o Aluno está sendo inserido. O método *FirstOrDefault()* retorna o primeiro valor ou o valor padrão que aquela consulta pode encontrar e concede ao atributo turma do Aluno, por fim, invoca o método *Salvar()* para realizar a persistência desse objeto no banco de dados, conforme a Figura 20.

```
public void Salvar(Aluno aluno)
{
    aluno.Turma = banco.Turmas.ToList()
        .Where(a => a.TurmaId == aluno.Turma.TurmaId)
        .FirstOrDefault();
    banco.Alunos.Add(aluno);
    banco.SaveChanges();
}
```

Figura 20 - Utilizando LINQ para salvar um objeto Aluno
Fonte: do Autor

Na figura 20, o objeto Aluno possui uma propriedade turma e nessa propriedade é realizada uma consulta LINQ onde é verificado que o ID da turma do objeto aluno seja igual o ID do objeto Turma que possui o valor 1, retornando uma lista com o primeiro objeto ou o objeto padrão dessa consulta. Após informar ao EF do vínculo que há entre Aluno e Turma, é invocado os métodos do Entity Framework para adicionar e salvar as alterações no banco de dados. É necessário realizar essa consulta antes da persistência do objeto Aluno para que o Entity Framework saiba que há uma associação entre os objetos Aluno e Turma e não crie uma nova linha na tabela Turma e sim vincule o ID da tabela Turma no campo ID da turma da tabela Aluno.

Toda alteração que é necessária realizar em uma linha de alguma tabela no banco de dados, é preciso realizar uma consulta onde retorne essa linha que irá ser modificada o valor no campo dessa tabela. Por exemplo, para alterar o salário de um Professor, cria um objeto Professor, atribui na propriedade ID do objeto um valor de chave primária já existente no banco de dados e atribui um novo valor a propriedade salário, após atribuir o novo valor, invoca o método *Alterar()* para realizar a alteração proposta, conforme a Figura 21.

```
public void Alterar(Professor professor)
{
    Professor professorSalvar = banco.Professores
        .Where(p => p.ProfessorId == professor.ProfessorId).First();
    professorSalvar.Salario = professor.Salario;
    banco.SaveChanges();
}
```

Figura 21 - Alterando o salário do Professor
Fonte: do Autor

Na Figura 21, é criado um objeto Professor denominado de *professorSalvar*, que tem por objetivo trazer do banco de dados a linha correspondente do ID existente para a modificação. Nesse novo objeto criado, é realizada uma consulta onde retorna uma lista com o primeiro valor onde a linha possua o ID que necessita ser alterado o campo salário. Após isso, é atribuído o novo valor e salvo no banco de dados a alteração. Com o LINQ é possível realizar consultas que retornem os dados que estão persistidos na base de dados sem utilizar a função do Entity Framework. Por exemplo, para realizar uma consulta no banco de dados que retorne todas as linhas da tabela Professor, deve-se realizar o procedimento da Figura 22.

```
ProfessorControlador dbProfessor = new ProfessorControlador();

var consultaTodosProfessores = (from prof in dbProfessor.banco.Professores
                                select prof).ToList();

foreach (var professores in consultaTodosProfessores)
{
    Console.WriteLine("ID {0} - NOME {1} - SALARIO {2}"
        , professores.ProfessorId, professores.Nome, professores.Salario);
}
```

```
ID 1 - NOME Marcelo Calabreze - SALARIO 4000
ID 2 - NOME Davi Ricardo - SALARIO 5000
ID 4 - NOME Livia Gonçalves - SALARIO 4000
```

Figura 22 - Consulta LINQ de todos os Professores
Fonte: do Autor

Na Figura 22, primeiramente deve-se obter a conexão com o banco, depois criar uma variável e realizar uma consulta onde busca na tabela Professor todos os professores que existem nessa tabela. A variável *consultaTodosProfessores* recebe uma lista com todos as linhas do banco de dados. Para realizar a exibição é necessário fazer um laço de repetição para exibir todos os dados que estão persistidos na tabela Professor.

O LINQ possui alguns operadores para realizar suas consultas – filtrar, ordenar. Para filtrar, utiliza-se a cláusula *where* onde busca um campo com o valor procurado. Por exemplo, para realizar uma busca onde retorne todos os professores que estão ganhando o salário de R$ 4.000,00 reais, deve-se escrever as linhas de código da Figura 23.

```
var consultaComFiltro = (from p in dbProfessor.banco.Professores
                         where p.Salario == 4000
                         select p);
foreach (var professores in consultaComFiltro)
{
    Console.WriteLine("ID {0} - NOME {1} - SALARIO {2}"
        , professores.ProfessorId, professores.Nome, professores.Salario);
}
```

```
ID 1 - NOME Marcelo Calabreze - SALARIO 4000
ID 4 - NOME Livia Gonçalves - SALARIO 4000
```

Figura 23 - Consulta de Professor utilizando filtragem
Fonte: do Autor

Na Figura 23, a variável *consultaComFiltro* recebe um ou mais valores que atende ao critério que está na cláusula *where*, onde o salário tem que ser igual a R$ 4.000,00 reais.

Para realizar uma busca que seja exibida ordenada, deve-se utilizar a cláusula *orderby*, essa cláusula que se encarrega de realizar uma ordenação por algum critério que seja estipulado. Por exemplo, para realizar uma consulta onde retorne todas as linhas que estão inseridas na tabela Aluno, e esse resultado seja ordenado pelo nome, deve-se realizar o código da Figura 24.

```
AlunoControlador dbAluno = new AlunoControlador();

var consultaOrdenadaAlunos = (from a in dbAluno.banco.Alunos
                              orderby a.Nome
                              select a).ToList();

foreach (var alunos in consultaOrdenadaAlunos)
{
    Console.WriteLine("ID {0} - NOME {1} "
        , alunos.AlunoId, alunos.Nome);
}
```

```
ID 3 - NOME Gustavo Neto
ID 1 - NOME Paulo Alcaraz
ID 2 - NOME Ricardo dos Reis
```

Figura 24 - Utilizando o ordenador na tabela Aluno
Fonte: do Autor

Na Figura 24, primeiramente deve-se obter a conexão com o banco de dados, depois criar uma variável que recebe a consulta onde busca na tabela Aluno todos as linhas que estão nessa tabela e retorna uma lista ordenada pelo nome do aluno.

4.5 PROCESSOS DE CARGA EXISTENTES NO EF

Quando é necessário realizar alguma consulta no banco de dados, retornando algum dado da base de dados, como por exemplo, obter as informações de todos os registros de uma tabela ou realizar uma consulta onde retorna uma linha específica, usa-se os processos de carga. Para o EF existem três tipos: Carga Tardia - Lazy Load, Carga Forçada - Eager Load e Carga Explícita - Explicit Load. O primeiro tipo de processo de carga é configurado por padrão, não carregando as informações das associações sem mencionar explicitamente as que devem ser carregadas. Por exemplo, para carregar uma lista de Disciplina e nessa lista conter Professor, é preciso, na consulta, fazer uso do método *Include()* e dentro desse método a classe que deseja carregar, conforme a Figura 25.

```
public IEnumerable<Disciplina> Listar()
{
    return banco.Disciplinas.Include(d => d.Professor).ToList();
}
```

Figura 25 - Processo de carga - Lazy Load
Fonte: do Autor

Como pode ser verificado, na Figura 25, o método *Listar()* retorna uma lista de *IEnumerable*<> de Disciplinas incluindo os Professores nessa lista.

O segundo processo, Eager Load, carrega todos os dados na memória de uma só vez, fazendo uso de sentenças *joins*. Para habilitar este tipo de carga, no contexto do projeto dentro do método *OnModelCreating()*, deve-se usar o *Configuration()* e atribuir o Lazy Loading como falso (Figura 26).

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    Configuration.LazyLoadingEnabled = false;
```

Figura 26 - Classe Contexto, fragmento do método OnModelCreating() - Habilitando Eager Load
Fonte: do Autor

Com este tipo de configuração, não é mais necessário utilizar o método *Include()* nas consultas, porém, qualquer multiplicidade N acarretará em problemas de desempenho. O Eager Load é mais utilizado para casos onde há muitas multiplicidades entre classes com associações um-para-um.

O último processo de carga, Explicit Load, é usado mesmo quando o Lazy Load está desativado, possibilitando o carregamento tardio das classe que estão associadas, para isso deve-se usar o método *Load()* ao invés de *Include()*.

4.6 LOGGING

Quando é necessário obter informações do que exatamente está sendo executado no banco de dados, por exemplo, quando realiza uma consulta em alguma tabela, insere uma nova linha na tabela, exclui-se uma linha, pode-se usar o *Logging*. O *Logging* tem a função de informar o que está sendo executado no banco de dados, possibilitando obter as informações

exatas do que está sendo feito. Para utilizar este recurso é necessário obter a propriedade do tipo do contexto, invocar a propriedade *Database* e atribuir à subpropriedade *Log* como na Figura 27.

```
public void Salvar(Aluno aluno)
{
    //LOGGING
    banco.Database.Log = Console.Write;

    banco.Alunos.Add(aluno);
    banco.SaveChanges();
}
```

Figura 27 - Método Salvar() - Utilizando Logging
Fonte: do Autor

Como pode ser verificado, na Figura 27, toda vez que for necessário salvar um Aluno utilizando o método *Salvar()*, primeiramente irá habilitar o *Logging* e exibir no console todas as alterações que o ORM faz para salvar um objeto no banco de dados. Com esse recurso habilitado, quando se realiza uma manutenção no estado de objetos pertencentes ao contexto, tem-se de saída, algo semelhante à Figura 28.

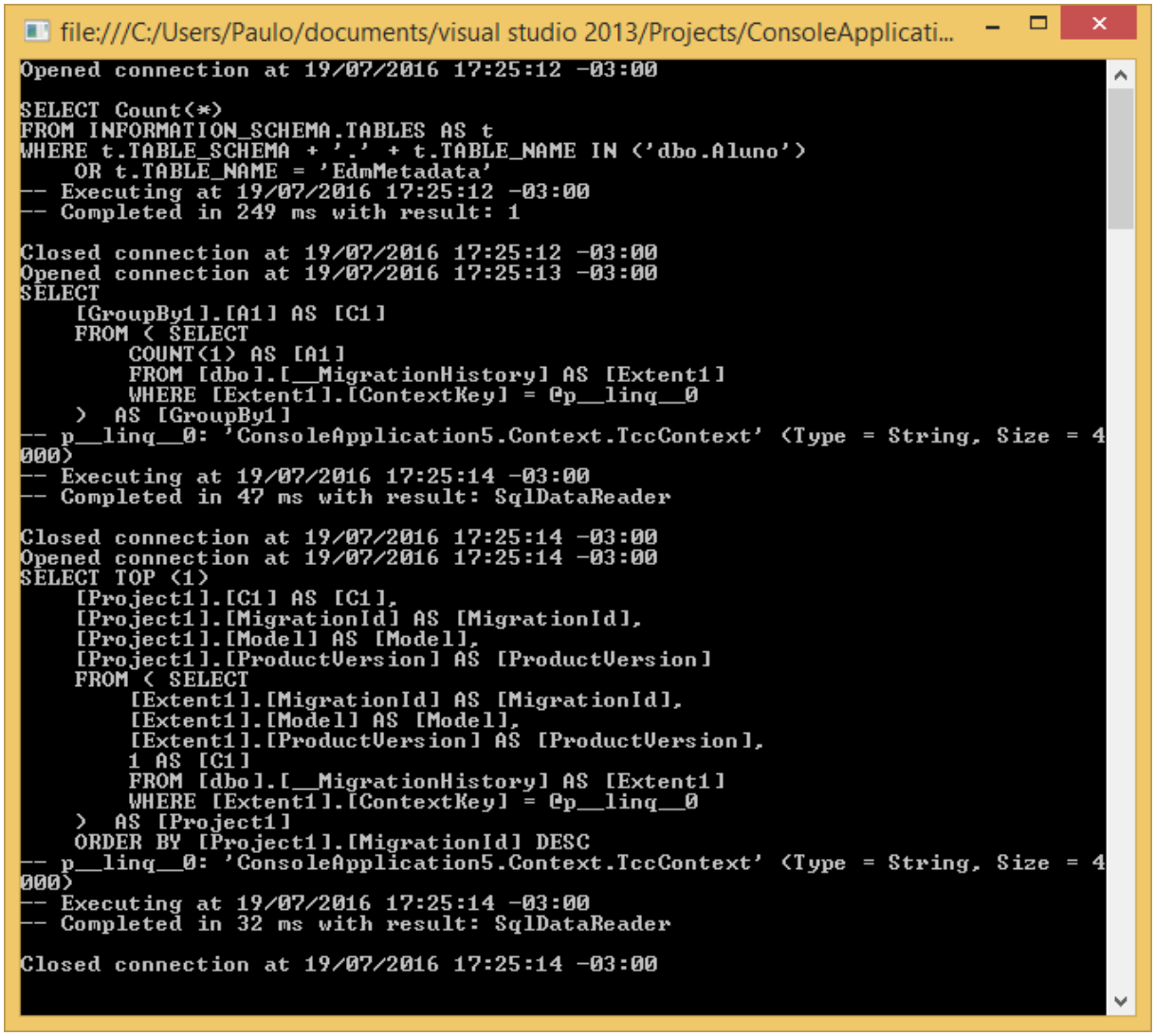

Figura 28 - Saída no console - Resultado Logging
Fonte: do Autor

Como pode ser observado na Figura 28, obtém-se informações de data, hora e GMT (Greenwich Mean Time) de que foi aberta a conexão, a consulta que foi realizada na tabela com a hora que foi executada e quanto tempo em segundos que demorou para ser completada, e todas as ações realizadas no banco. Desse modo é possível verificar como são realizadas as consultas que o ORM faz no banco de dados, demonstrando todas as suas ações.

O Logging também possibilita exibir as informações de uma forma personalizada. Por exemplo, é possível que a exibição do logging seja modificada, onde criando uma classe chamada *LogMod.cs* possua o código da Figura 29.

```
public class LogMod
{
    1 reference
    public void Log(string componente, string msg)
    {
        Console.WriteLine("COMPONENTE: {0} MSG: {1} ", componente, msg);
    }
```

Figura 29 - Modificando a Exibição do Logging
Fonte: do Autor

Na Figura 29, a classe possui um método público que não retorna nenhum valor, onde recebe como parâmetro duas *strings*: um componente para especificar a ação e uma informação de uma ação do Entity Framework. Com isso possibilita que a exibição do log seja diferente. Por exemplo, utilizando o exemplo de uma consulta LINQ, tem-se a Figura 30.

```
ProfessorControlador dbProfessor = new ProfessorControlador();

var meuLog = new LogMod();
dbProfessor.banco.Database.Log = s => meuLog.Log("EFapp", s);

var consultaTodosProfessores = (from prof in dbProfessor.banco.Professores
                                select prof).ToList();

foreach (var professores in consultaTodosProfessores)
{
    Console.WriteLine("ID {0} - NOME {1} - SALARIO {2}"
        , professores.ProfessorId, professores.Nome, professores.Salario);
}
```

Figura 30 - Exemplo da Utilização do Logging
Fonte: do Autor

Na Figura 30, primeiramente obtém a conexão com o banco de dados, depois é criada uma variável chamada de *meuLog* que recebe a classe *LogMod* e na seguinte linha tem-se a utilização da propriedade subpropriedade *Log* do Entity Framework, e invoca o método *Log* da classe *LogMod* criada, passando como parâmetro uma *string* de identificação denominada *EFapp* e a ação que a subpropriedade *Log* realiza no banco de dados, com isso, tem-se como saída no console a Figura 31.

```
COMPONENTE: EFapp MSG: Opened connection at 29/11/2016 01:29:01 -02:00

COMPONENTE: EFapp MSG:
SELECT Count(*)
FROM INFORMATION_SCHEMA.TABLES AS t
WHERE t.TABLE_SCHEMA + '.' + t.TABLE_NAME IN ('dbo.Aluno','dbo.Enderecoes','dbo.
Turmas','dbo.Disciplinas','dbo.Professor','dbo.DisciplinaTurmas')
    OR t.TABLE_NAME = 'EdmMetadata'
COMPONENTE: EFapp MSG:

COMPONENTE: EFapp MSG: -- Executing at 29/11/2016 01:29:01 -02:00

COMPONENTE: EFapp MSG: -- Completed in 42 ms with result: 6

COMPONENTE: EFapp MSG:

COMPONENTE: EFapp MSG: Closed connection at 29/11/2016 01:29:01 -02:00

COMPONENTE: EFapp MSG: Opened connection at 29/11/2016 01:29:02 -02:00

COMPONENTE: EFapp MSG: SELECT
    [GroupBy1].[A1] AS [C1]
    FROM ( SELECT
        COUNT(1) AS [A1]
        FROM [dbo].[__MigrationHistory] AS [Extent1]
        WHERE [Extent1].[ContextKey] = @p__linq__0
    )  AS [GroupBy1]
COMPONENTE: EFapp MSG:

COMPONENTE: EFapp MSG: -- p__linq__0: 'SistemaBasicoInstituicao.contexto.TccCont
ext' (Type = String, Size = 4000)

COMPONENTE: EFapp MSG: -- Executing at 29/11/2016 01:29:02 -02:00

COMPONENTE: EFapp MSG: -- Completed in 21 ms with result: SqlDataReader

COMPONENTE: EFapp MSG:

COMPONENTE: EFapp MSG: Closed connection at 29/11/2016 01:29:02 -02:00

COMPONENTE: EFapp MSG: Opened connection at 29/11/2016 01:29:02 -02:00

COMPONENTE: EFapp MSG: SELECT TOP (1)
    [Project1].[C1] AS [C1],
    [Project1].[MigrationId] AS [MigrationId],
    [Project1].[Model] AS [Model],
    [Project1].[ProductVersion] AS [ProductVersion]
    FROM ( SELECT
        [Extent1].[MigrationId] AS [MigrationId],
        [Extent1].[Model] AS [Model],
        [Extent1].[ProductVersion] AS [ProductVersion],
        1 AS [C1]
        FROM [dbo].[__MigrationHistory] AS [Extent1]
        WHERE [Extent1].[ContextKey] = @p__linq__0
    )  AS [Project1]
    ORDER BY [Project1].[MigrationId] DESC
COMPONENTE: EFapp MSG:
```

Figura 31 - Exibição no Console do Logging Modificado
Fonte: do Autor

Na Figura 31, exibe o log da execução da consulta LINQ realizada na Figura 30 como foi implementado no método *Log()* da classe *LogMod*. Dessa forma permite deixar mais visível todas as ações que o Entity Framework realiza na aplicação e abrir um leque para futuras personalizações com relação a essa função do ORM.

4.7 MIGRATIONS

Quando há necessidade de realizar alguma alteração no modelo orientado a objeto, é preciso refletir essa modificação no banco de dados, para isso deve-se habilitar essa função Migrations do Entity Framework para que possa realizar a atualização no banco de dados. A função do Migrations é manter a base de dados atualizada com as classes. Por exemplo, na classe Aluno é adicionado um novo atributo telefone, chamado de *TelAluno* do tipo *string*. Para que esse novo atributo seja refletido na tabela Aluno, primeiramente deve-se abrir o *Package Manager Console* e com o comando *Enable-Migrations* deve-se habilitar no projeto as migrações, depois é necessário adicionar a nova migração com o comando *Add-Migrations* seguido de um nome para essa migração. Após realizar a adição dessa migração, deve-se atualizar o banco de dados com o comando *Update-Database*, com esses procedimentos realizados tem-se a conclusão da atualização no banco de dados, conforme demonstra a Figura 32.

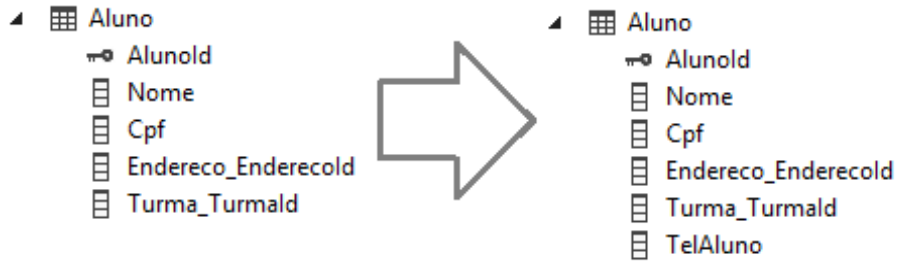

Figura 32 - Migrations - Adicionando um novo campo na classe Aluno
Fonte: do Autor

Na Figura 32, o lado esquerdo está a tabela Aluno no banco de dados sem a migração do novo campo de telefone do aluno, do lado direito é o resultado que se têm com essa migração com o campo TelAluno representando o telefone do Aluno. É dever do programador povoar ou não as novas alterações (usando o método *Seed()* da classe *Configuration.cs*, por exemplo) para os novos campos adicionado, caso não realize algum procedimento esses novos atributos permanecem com valor nulo no banco de dados.

Com o Migrations também existe a possibilidade de realizar migrações sem que seja necessário realizar o procedimento mencionado anteriormente, delegando ao Entity Framework a responsabilidade de realizar as atualizações no banco dados. Essas configuração tem que ser realizada no construtor da classe *Configuration.cs*, conforme a Figura 33.

```
public Configuration()
{
    AutomaticMigrationsEnabled = true;
```

Figura 33 - Habilitando migrações automáticas
Fonte: do Autor

Na figura 33, é informado ao Entity Framework para habilitar as migrações de forma automática, possibilitando que o ORM gerencie todas as alterações que é necessária ser refletida na base de dados e automatizando o processo de migração.

O Migrations possibilita gerar um script para rodar no ambiente do cliente. Para que isso seja realizado, no *Package Manager Console* deve-se usar o comando *Update-Database -Script -SourceMigration:$InitialCreate -TargetMigration:"NomeDaMigração"*, esse comando gera um script com as mudanças desde a criação do banco de dados (InitialCreate) até a última migração que necessita gerar o script.

5 CONSIDERAÇÕES FINAIS

O Entity Framework é um ORM da Microsoft utilizado para diminuir o tempo de desenvolvimento da aplicação, permitindo escolher uma das três estratégias (Model First, Database First ou Code First) para a utilização desse ORM.

Quando se opta pelo Code First, tem-se uma autonomia a mais no projeto, possibilitando realizar configurações a partir do modelo Orientado a Objetos da forma que mais atenda às necessidades do programador, auxiliando o desenvolvedor na questão da comunicação entre a aplicação e o banco de dados, automatizando e deixando muitos processos transparente para o programador.

As maiores dificuldades em se trabalhar com algum ORM é partir do princípio de abstração que cada ORM proporciona, visando suas qualidades e limitações, o EF pode realizar muitos procedimentos, porém se não tiver um conhecimento do que está sendo pedido para que o ORM processe, pode gerar complicações na hora de realizar a persistência de um objeto ou recuperar esse dado do banco de dados.

O principal objetivo de realizar essa documentação foi de demonstrar que existe ferramentas no mercado que podem ajudar no desenvolvimento, tornando público esse conteúdo, facilitando o aprendizado de algum outro programador, fornecendo uma base para que possa realizar o seu código utilizando ferramentas da Microsoft. Por isso, foi abordado os principais temas como, as associações entre cada classe, utilização de herança, utilização do processo de carga Lazy Load que já vem habilitado por padrão no Entity Framework e uma sutil demonstração de como usar o recurso de Logging e Migrations do EF que vem nativo na ferramenta para auxiliar e melhorar o tempo de produção.

Todas essas melhorias andam lado a lado em quesito de programação e banco de dados e a cada versão lançada pela Microsoft visa trazer melhorias e ferramentas que possam facilitar o uso desse ORM.

5.1 TRABALHOS FUTUROS/CONTINUAÇÃO DO TRABALHO

Poderá ser feito um projeto inteiro utilizando as outras abordagens do Entity Framework (model first e database first) para realizar um comparativo e trazer informações e particularidades de cada tipo de abordagem. Poderá também ser feito um comparativo de ORM’s existentes atualmente para que explore a particularidade de cada ORM, demonstrando seus prós e contras.

REFERÊNCIAS BIBLIOGRÁFICAS

BAUER, Christian; KING, Gavin. **Hibernate Em Ação**. Rio de Janeiro: Editora Ciência Moderna, **2005.**

DATE, Chris J. **Introdução a Sistemas de Bancos de Dados**. Rio de Janeiro: Elsevier, 2004.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados** 6º Edição. São Paulo: Pearson, 2011.

GALUPPO, Fabio; MATHEUS, Vanclei; SANTOS, Wallace. **Desenvolvendo com C#**. São Paulo: Bookman, 2004.

JANDL, Peter.**JAVA** Guia do programador Atualizado para JAVA 6. São Paulo: Novatec, 2007.

KHOSHAFIAN, Setrag. **BANCO DE DADOS ORIENTADO A OBJETOS**. Rio de Janeiro: Infobook, 1994.

LIPPMAN, Stanley B. **C#: um guia prático**. Porto Alegre: Bookman, 2003.

MACHADO, Felipe; ABREU, Mauricio. **PROJETO DE BANCO DE DADOS** Uma visão Prática. 9º Edição. São Paulo: Editora Érica, 2002.

MICROSOFT, **Entity Framework Logging and Intercepting Database Operations (EF6 Onwards)**. Disponível em: <https://msdn.microsoft.com/en-us/library/dn469464(v=vs.113).aspx>. Acessado em, 25 de novembro de 2016.

MICROSOFT, **Entity Framework**. Disponível em: <https://msdn.microsoft.com/en-us/data/aa937723>. Acessado em, 29 de fevereiro de 2016.

MICROSOFT, **Express Edition**. Disponível em: < https://www.microsoft.com/pt-BR/server-cloud/Products/sql-server-editions/sql-server-express.aspx >. Acessado em, 13, de abril de 2016.

MICROSOFT, **Fundamentos do Entity Framework 4**. Disponível em: <https://msdn.microsoft.com/pt-br/library/jj128157.aspx>. Acessado em, 03 de abril de 2016.

MICROSOFT, **Introdução a consultas LINQ (C#)**. Disponível em: <https://msdn.microsoft.com/pt-br/library/bb397906.aspx>. Acessado em, 25 de novembro de 2016.

MICROSOFT, **Introduction to the C# Language and the .NET Framework**. Disponível em: <https://msdn.microsoft.com/pt-br/library/z1zx9t92.aspx >. Acessado em, 23 de julho de 2016.

MICROSOFT, **Sintaxe de consulta e sintaxe de método em LINQ (C#).** Disponível em: <https://msdn.microsoft.com/pt-br/library/bb397947.aspx>. Acessado em, 25 de novembro de 2016.

ROB, Peter; CORONEL, Carlos. **Sistema de Banco de Dados:** Projeto, Implementação e Gerenciamento. Tradução da 8º edição norte-americana. São Paulo: Cengage Learning, 2011.

ROBINSON, Simon; ALLEN, K. Scott; CORNES, Ollie; GLYNN, Jay; GREENVOSS, Zach; HARVEY, Burton; NAGEL, Christian; SKINNER, Morgan; WATSON, Karli. **Profissional C# Programando**. São Paulo: Makron Books, 2004.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUDARSHAN, S. **Sistema de Banco de Dados**. Rio de Janeiro: Editora Campus, 2006.

SOARES Odilon. Herculano. S. **Utilização do Framework Hibernate pra Mapear Objeto Relacional na Construção de Um Sistema de Informação**. Disponível em, <https://dsc.inf.furb.br/arquivos/tcc/monografias/2006-1odilonherculanosoaresfilhovf.pdf>. Acessado em, 29 de agosto de 2016.

STELLMAN, Andrew; GREENE, Jennifer. **Use a Cabeça! C#**. Rio de Janeiro: Alta Books, 2008.

TROELSEN, Andrew. **Profissional C# e a Plataforma .NET 3.5 – Curso completo**. Rio de Janeiro: Alta Book, 2009.