CENTRO *UNIVERSITÁRIO* DE BARRA MANSA
PRÓ-REITORIA ACADÊMICA

CURSO DE ENGENHARIA DE COMPUTAÇÃO

LINGUAGEM DE PESQUISA INTEGRADA À LINGUAGEM DE PROGRAMAÇÃO

Por:

Leniel Braz de Oliveira Macaferi

Barra Mansa
Dezembro de 2007

CENTRO *UNIVERSITÁRIO* DE BARRA MANSA
PRÓ-REITORIA ACADÊMICA

CURSO DE ENGENHARIA DE COMPUTAÇÃO

LINGUAGEM DE PESQUISA INTEGRADA À LINGUAGEM DE PROGRAMAÇÃO

Por:

Leniel Braz de Oliveira Macaferi

Monografia apresentada ao curso de Engenharia de Computação do Centro Universitário de Barra Mansa, como requisito parcial para a obtenção do título de Engenheiro em Engenharia de Computação, sob a orientação do professor José Nilton Cantarino Gil.

Barra Mansa
Dezembro de 2007

LINGUAGEM DE PESQUISA INTEGRADA À LINGUAGEM DE PROGRAMAÇÃO

Por:

Leniel Braz de Oliveira Macaferi

Monografia apresentada ao curso de Engenharia de Computação do Centro Universitário de Barra Mansa, submetida à aprovação da banca examinadora composta pelos seguintes membros:

_____

José Nilton Cantarino Gil

_____

Laurentino Duodécimo Rosado Fernandes

_____

Marcelo Arantes de Oliveira

Barra Mansa
Dezembro de 2007

Ao Deus de Israel, ao seu filho amado Jesus Cristo e aos meus amados pais Ilsonina e Pedro.

Agradeço a todos que contribuíram para a realização deste trabalho.

“Tudo quanto te vier à mão para fazer, faça conforme as tuas forças, porque no além, para onde tu vais, não há obra, nem projetos, nem conhecimento, nem sabedoria alguma.”

Eclesiastes 9:10

RESUMO

Macaferi, Leniel Braz de Oliveira. Linguagem de pesquisa integrada à linguagem de programação. 2007. 96f. Monografia (bacharelado em Engenharia de Computação) - Centro Universitário de Barra Mansa, Barra Mansa, 2007. www.ubm.br

Dados formam a matéria prima da computação e são processados via software. Produtos de software são geralmente estruturados em camadas, tipicamente três: a camada de dados, a camada intermediária ou de lógica e a camada de apresentação ou do cliente. Cada uma destas camadas possui seu próprio modelo de dados. Estes diferentes paradigmas causam o problema da combinação mal sucedida entre estes três modelos completamente diferentes.

Ao invés de tentar unificar no nível do modelo de dados, uma melhor alternativa é unificar no nível das operações algébricas que podem ser definidas do mesmo modo sobre cada modelo de dados. Isto nos permite definir uma única linguagem de pesquisa que pode ser usada para pesquisar e transformar qualquer modelo de dados. Tudo o que os modelos de dados precisam implementar é um pequeno conjunto de operadores de pesquisa padrão, e cada modelo de dados pode fazer isto de uma maneira natural.

A indústria chegou a um ponto estável na evolução das tecnologias de programação orientada a objetos (OO). Desenvolvedores agora têm por certo as facilidades das linguagens de programação OO e seus ricos recursos iguais a classes, objetos, métodos e eventos. Tais linguagens suportam a criação e uso de bibliotecas de classe de estilo funcional de ordem maior. O suporte é o resultado das novas extensões de linguagem de programação que estão sendo desenvolvidas. Estas extensões permitem a criação de interfaces para programação de aplicativos (APIs) composicionais que possuem poderosas capacidades de pesquisa dentro da sintaxe da linguagem de programação. Isto torna viável a implementação dos operadores de pesquisa padrão. Os operadores de pesquisa padrão podem ser aplicados em todas as fontes de informação, não somente em domínios de bancos de dados relacionais ou XML.

Este trabalho visa apresentar e utilizar os aspectos mais importantes da linguagem integrada de pesquisa com foco na integração da linguagem de pesquisa SQL à linguagem de programação C#. Aspectos como a simplificação da maneira de escrever pesquisas, unificação da sintaxe para pesquisar qualquer fonte de dados, reforço da conexão entre dados relacionais e o mundo orientado a objetos e o menor tempo gasto no processo de desenvolvimento de software.

Palavras-chave: linguagem de pesquisa, linguagem de programação, modelos de dados, SQL, C#, LINQ

ABSTRACT

Macaferi, Leniel Braz de Oliveira. Query language integrated into the programming language. 2007. 96f. Monograph (bachelor's degree in Computer Engineering) - Barra Mansa University Center, Barra Mansa, 2007. www.ubm.br

Data is the raw material of computation and is processed via software. Software products are generally structured in tiers, typically three, the data tier, the middle or business tier and the presentation or client tier. Each of these tiers has its own data model. These different paradigms cause the impedance mismatch problem between these three disparate models.

Instead of trying to unify at the data model level, a better approach is to unify at the level of algebraic operations that can be defined the same way over each data model. This allows us to define a single query language that can be used to query and transform any data model. All the data model need to do is to implement a small set of standard query operators, and each data model can do so in a way natural to itself.

The industry has reached a stable point in the evolution of object-oriented (OO) programming technologies. Programmers now take for granted the facilities of oriented programming languages and their features like classes, objects, methods and events. Such languages support the creation and use of higher order, functional style class libraries. The support is the result of new language extensions being developed. These extensions enable the construction of compositional application program interfaces (APIs) that have equal expressive power of query languages inside the programming language syntax. This makes it possible to implement the standard query operators. The standard query operators can be then applied to all sources of data, not just relational or XML domains.

This work aims to present and use the most important aspects of the language integrated query with special focus on the integration of the SQL query language into the C# programming language. Aspects as simplification of the way of writing queries, unification of the syntax for querying any data source, reinforcement of the connection between relational data and the object oriented world and less time spent in the software development process.

Keywords: query language, programming language, data models, SQL, C#, LINQ

LISTA DE ABREVIATURAS

ADO	ActiveX Data Object
API	Application Program Interface
ASP	Active Server Page
BCL	Base Class Library
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Common Language Specification
CTS	Common Type System
DBMS	Database Management System
DLL	Dynamic Link Library
EXE	Executable file
FCL	Framework Class Library
GPA	Grade Point Average
GUI	Graphical User Interface
IDE	Integrated Development Environment
IL	Intermediate Language
ISO	International Standards Organization
JDBC	Java Database Connectivity
JIT	Just-in-Time Compiler
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LINQ	Language Integrated Query
MVC	Model-View-Controller
OO	Object Oriented
PL/SQL	Procedural Language/Structured Query Language
PLINQ	Parallel Language Integrated Query
ROX	Relations-Objects-XML
SP	Stored Procedure
SQL	Structured Query Language
SQLJ	Structured Query Language-Java
T-SQL	Transact-Structured Query Language
UDF	User Defined Function
VES	Virtual Execution System
XML	Extensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

SUMÁRIO

Pág.

1 INTRODUÇÃO

Aplicações distribuídas são geralmente estruturadas em camadas, tipicamente três: a camada de dados, a camada intermediária ou de lógica e a camada de apresentação ou do cliente [1]. Cada uma destas camadas possui seu próprio modelo de dados. A camada de dados usualmente lida com dados tabulados armazenados em um banco de dados relacional. A camada intermediária usualmente lida com classes definidas em alguma linguagem de programação orientada a objetos. Finalmente, a camada de apresentação usualmente lida com dados semi-estruturados como XML[a] ou JSON[b]. Daí o triângulo ROX[c] [2].

Antes mesmo de pensarmos em demais aspectos da aplicação, precisamos primeiro resolver o problema da combinação mal sucedida entre estes três modelos.

Uma maneira de pensar é escolher um modelo de dados como sendo o universal e mapear todos os outros modelos de dados com base neste. A opção mais popular é ver todos os dados como XML e então usar XQuery[d] como a linguagem de junção. Outros tentam implementar um novo modelo de dados que engloba todos os modelos de dados previamente conhecidos, porém, este modelo universal de dados não tem se mostrado como a solução correta a ser adotada. Ao invés de tentar unificar no nível do modelo de dados, uma melhor alternativa é unificar no nível das operações algébricas que podem ser definidas do mesmo modo sobre cada modelo de dados. Isto nos permite definir uma única linguagem de pesquisa que pode ser usada para pesquisar e transformar qualquer modelo de dados. Tudo o que os modelos de dados precisam implementar é um pequeno conjunto de operadores de pesquisa padrão, e cada modelo de dados pode fazer isto de uma maneira natural. Isto é o que a Language Integrated Query (LINQ) [3] ou Pesquisa Integrada à Linguagem se propõe a fazer.

Deste ponto em diante chamarei LINQ de linguagem de pesquisa integrada à linguagem de programação.

---

a XML é uma linguagem de marcação de propósito geral. É classificada como uma linguagem de extensão porque permite que os usuários definam suas próprias expressões. Seu objetivo principal é facilitar o compartilhamento de dados estruturados através de diferentes sistemas de informação, particularmente a Internet.

b JSON é um formato de intercâmbio de dados geralmente utilizado para a transmissão de dados estruturados através de uma conexão de rede em um processo chamado serialização. Sua aplicação principal é na programação de aplicativos web que utilizam a tecnologia Ajax (Asynchronous JavaScript e XML). Assim, serve como uma alternativa à tradicional XML.

c ROX é um termo usado para descrever o triângulo formado por relações, objetos e XML.

d XQuery é uma linguagem de pesquisa com recursos similares aos de uma linguagem de programação. É usada para pesquisar coleções de dados XML. Do ponto de vista semântico é similar à SQL.

1.1 Delimitação do tema

Este trabalho é uma pesquisa de natureza teórica e prática.

Serão apresentados os conceitos mais importantes sobre a integração da linguagem de pesquisa SQL à linguagem de programação C#.

Para fins de utilização e avaliação dos conceitos apresentados, uma aplicação cliente/servidor[e] será desenvolvida.

1.2 Problema

O problema é a combinação mal sucedida entre os diferentes modelos de dados.

Nos dias de hoje, os vários modelos de dados e suas linguagens de pesquisa são estritamente unidos.

Para tornar isto mais evidente, vejamos algumas pesquisas simples em cada modelo de dados. Cada pesquisa irá selecionar o nome, área e população de todos os países da Europa.

O Código 1 mostra a pesquisa em SQL:

```
select Name, Area, Population
from Countries
where Continent = "Europe"
```

Código 1 - Pesquisa em SQL

O Código 2 mostra a pesquisa em XQuery:

```
from $c in countries/country
where $c/continent == "Europe"
return <result>{$c/Name}{$c/Area}{$c/Population}</result>
```

Código 2 - Pesquisa em XQuery

O Código 3 mostra a pesquisa em uma linguagem de programação orientada a objetos:

```
class Result { string Name; double Area; int Population; }

var r = new List<Result>();

foreach(var c in Countries)
  if(c.Continent == "Europe")
    r.Add(new Result { c.Name, c.Area, c.Population });
```

Código 3 - Pesquisa em uma linguagem de programação orientada a objetos

Em cada caso, a pesquisa essencialmente itera sobre alguma forma de coleção, filtra os elementos da coleção que satisfazem uma dada condição, e finalmente cria uma coleção de resultado aplicando uma transformação no valor proveniente da coleção original.

[e] Cliente/servidor refere-se a uma arquitetura computacional que separa clientes (ex.: browser) e servidores (ex.: servidor de banco de dados), os quais são interligados entre si geralmente através de uma rede de computadores.

1.3 Enunciado das hipóteses

De forma otimista, dentro de 10 anos, as linguagens de programação simplesmente terão esta inovação de pesquisa integrada como um conceito inerente porque isto é altamente desejável e essencial.

O tempo despendido no desenvolvimento de produtos de software diminuirá drasticamente, o que permitirá um maior rendimento do trabalho dos desenvolvedores. Do outro lado do negócio, as empresas poderão gastar menos em virtude da rápida programação do software.

A performance das aplicações em geral tende a aumentar, pois novas extensões de linguagem que viabilizam a integração da linguagem de pesquisa à linguagem de programação estão sendo desenvolvidas. Técnicas de pesquisa que permitem explorar todo o potencial dos processadores multi-nucleares através do uso de processamento paralelo (veja Figura 1) também estão sendo desenvolvidas como é o caso da PLINQ[f] [4]. Com isso, a linguagem de pesquisa integrada à linguagem de programação poderá explorar todo este arsenal de tecnologias, as quais visam justamente diminuir o gargalo hoje existente nas aplicações que trabalham com grandes quantidades de dados em um ambiente completamente distribuído como é o caso da Internet. Mesmo aplicações que são executadas localmente em um único processador sentirão os efeitos da melhor performance.

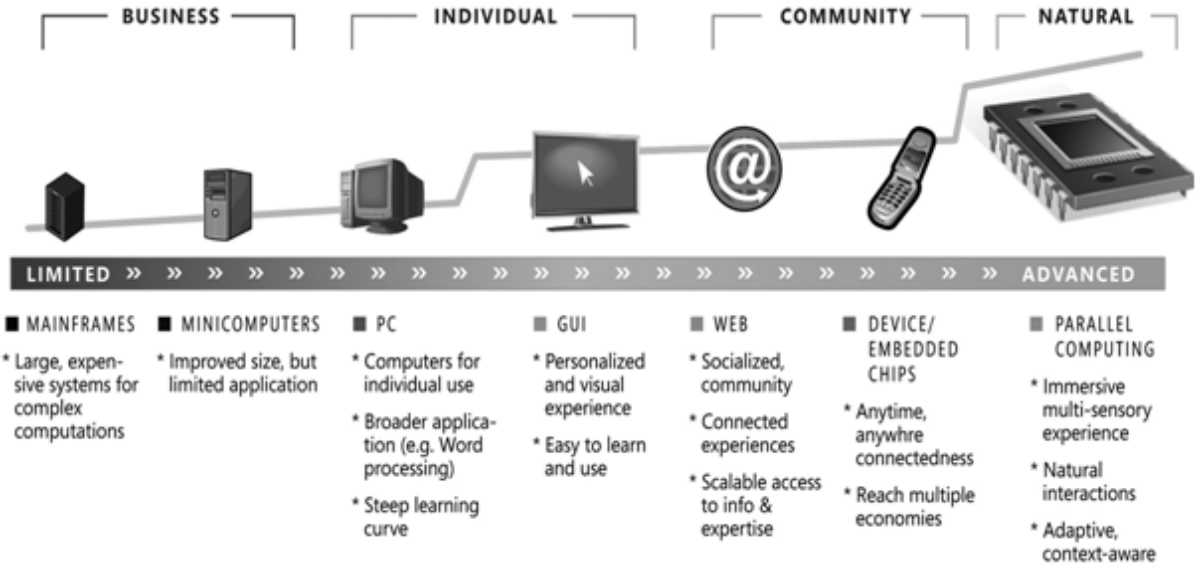

Figura 1 - Computação paralela

[f] PLINQ é um motor de execução de pesquisas concorrentes para a linguagem de pesquisa integrada à linguagem de programação. As pesquisas sobre objetos são executadas de maneira paralela. O principal objetivo de PLINQ é expor o paralelismo de dados através do uso de pesquisas. Qualquer computação feita com objetos, a qual tenha sido implementada como pesquisa pode ser executada paralelamente. Para tanto os objetos precisam implementar a interface IParallelEnumerable. PLINQ faz parte da biblioteca Parallel FX.

1.4 Objetivos específicos e geral

Os objetivos específicos englobam o melhor entendimento dos conceitos relacionados à integração da linguagem de pesquisa à linguagem de programação através de um estudo mais detalhado das extensões de linguagem que compõem este novo modelo unificado de programação. Após o estudo de tais extensões, a meta é construir uma aplicação cliente/servidor para colocar em prática os conceitos da linguagem de pesquisa integrada à linguagem de programação. Isto permitirá a utilização e avaliação desta nova maneira de programar.

De posse dos resultados, o objetivo geral será constatar as vantagens e possíveis desvantagens da integração da linguagem de pesquisa na linguagem de programação.

1.5 Justificativa do trabalho

A proposta da integração da linguagem de pesquisa na linguagem de programação é separar os modelos de dados de suas linguagens de pesquisa alavancando as similaridades fundamentais na semântica[g] operacional de cada linguagem de pesquisa. Deste modo, o problema da combinação mal sucedida é praticamente resolvido, o que contribui para simplificar o desenvolvimento de aplicações distribuídas que dependem intensamente de dados.

Os produtos de software requerem cada vez mais recursos ricos em multimídia, visualizações interativas, inteligência artificial, inteligência empresarial e o que mais possa enriquecê-las e tudo isso envolve a manipulação de modelos de dados.

Ao integrar a linguagem de pesquisa na linguagem de programação de maneira nativa, lidar diretamente com todos estes modelos através de uma única linguagem de pesquisa tende a ser mais fácil.

[g] Semântica é o estudo do significado dos símbolos e das palavras.

2 FUNDAMENTAÇÃO TEÓRICA

É fornecida a teoria necessária para que o leitor possa acompanhar o desenvolvimento do trabalho.

2.1 Linguagem de pesquisa

Uma linguagem de pesquisa é uma linguagem complexa de máquina com fortes regras sintáticas, semântica sofisticada e sem funcionalidade óbvia. Oferece várias opções que são usualmente propriedades de linguagens de programação (inserir, atualizar, deletar, etc) [5].

Linguagens de pesquisa são usadas para facilitar as tarefas de criação, pesquisa, organização e manipulação de dados. Dados estes geralmente armazenados em bancos de dados.

2.1.1 Pesquisa

Uma pesquisa é uma expressão na qual se especifica a informação que se deseja retornar da fonte de dados ou fontes de dados. Opcionalmente, uma pesquisa também especifica como esta informação deve ser classificada, agrupada e formatada antes de ser retornada.

A sintaxe[a] de pesquisa é usada para criar expressões de pesquisa. Uma expressão de pesquisa contém três cláusulas básicas: from, where e select.

Pesquisas são geralmente expressas em uma linguagem de pesquisa especializada. Diferentes linguagens de pesquisa foram desenvolvidas com o passar do tempo para os vários tipos de fontes de dados, por exemplo, a linguagem SQL para bancos de dados relacionais e XQuery para XML.

2.2 Linguagem de programação

Uma linguagem de programação é uma linguagem artificial que pode ser usada para controlar o comportamento de uma máquina, particularmente um computador [6]. Linguagens de programação, igualmente às linguagens naturais, são definidas por regras sintáticas e semânticas.

Linguagens de programação são usadas para facilitar as tarefas de comunicação,

[a] Sintaxe é o estudo das regras gramaticais que governam a estrutura de sentenças e expressões.

organização e manipulação de dados e para expressar algoritmos[b] precisamente. Alguns autores restringem o termo "linguagem de programação" para aquelas linguagens que podem expressar todos os algoritmos possíveis [7].

Muitas linguagens de programação já foram criadas e novas linguagens são criadas com freqüência para satisfazer as novas necessidades dos desenvolvedores. Atualmente existem mais de 2.500 linguagens de programação documentadas [8].

2.3 Combinação mal sucedida entre as linguagens de pesquisa e de programação

O conceito de linguagem de pesquisa que foi definido primeiramente no começo da década de 1970 não assumia o pragmatismo universal[c] [9]. Porém, em virtude de tal universalidade ser inevitável em aplicações reais, houve uma suposição de que uma linguagem de pesquisa deveria ser embutida (integrada) a uma linguagem de programação universal.

Uma linguagem de pesquisa deve ser uma “sublinguagem” que determina o acesso ao banco de dados somente. O resto de toda a aplicação deve ser programado em uma linguagem de programação típica. Esta suposição requer que a linguagem de pesquisa seja integrada na linguagem de programação de modo que:

- Pesquisas possam ser usadas dentro de aplicações;
- Pesquisas possam ser parametrizadas através de valores calculados dentro de aplicações;
- Resultados de pesquisas possam ser passados para aplicações.

A diferença entre os conceitos de linguagem causa dificuldades técnicas significantes para chegar neste objetivo de integração. Muitos desenvolvedores e profissionais da computação também ficaram desapontados pela degradação técnica e estética do ambiente de programação. Esta degradação é comumente denotada como impedance mismatch[d]. Este termo denota várias características desvantajosas que são implicadas pela mistura eclética de

[b] Algoritmo é uma lista definida de instruções concisas para completar uma tarefa; dado um estado inicial, percorre uma séria de estados sucessivos, eventualmente terminando em um estado final.

[c] Pragmatismo universal requer de uma linguagem de pesquisa ou de programação todas as facilidades que são atualmente esperadas por projetistas e desenvolvedores que usam estas linguagens. Tais linguagens devem permitir a execução eficiente de tarefas de programação em tempo, custo, força humana e qualidade adequada. Este conceito não se refere à completude relacional e à completude de Turing. Estes conceitos são totalmente irrelevantes quando é considerado o pragmatismo universal de linguagens de pesquisa e de programação.

[d] O termo impedance mismatch (combinação mal sucedida) tem sua raiz na eletrônica, onde denota a diferença entre a impedância de uma fonte e a impedância de um receptor, o que faz com que a força efetiva seja desperdiçada.

uma linguagem de pesquisa (em particular SQL) com uma linguagem de programação (tal como C, C++, C# ou Java[e]). A seguir estas características são listadas e comentadas.

- Sintaxe: em um mesmo código os desenvolvedores devem usar dois estilos de programação e devem seguir duas gramáticas diferentes. Conceitos similares são denotados diferentemente, por exemplo, strings em C# são escritas dentro de "..." (aspas duplas), em SQL '...' (aspas simples), e diferentes conceitos são denotados similarmente, por exemplo, em C# = (sinal de igual) denota uma atribuição, em SQL uma comparação.
- Tipificação: tipos e denotações de tipos assumidos por uma pesquisa e linguagens de programação diferem, como uma regra. Isto afeta tipos atômicos como inteiro, real, booleano, etc. Isto também afeta tipos complexos, tal como tabelas (um construtor de tipo básico em SQL, que não existe em linguagens de programação). Linguagens de programação populares introduzem checagem de tipo estática (no tempo de compilação), o que é impossível em SQL porque linguagens de pesquisa são baseadas no dinamismo ao invés da atribuição estática.
- Semântica e paradigmas de linguagem: o conceito de semântica de ambas as linguagens é totalmente diferente. Linguagens de pesquisa são baseadas no estilo declarativo (o que deve ser retornado ao invés de como retornar), enquanto que linguagens de programação são baseadas no estilo imperativo (uma seqüência de comandos para uma máquina, a qual executa o que fazer).
- Nível de abstração: linguagens de pesquisa livram os desenvolvedores da maioria dos detalhes inerentes à organização e implementação dos dados, por exemplo, organização de coleções, presença ou ausência de índices, etc. Em linguagens de programação estes detalhes são geralmente explícitos (embora possam ser tratados por algumas bibliotecas de código).
- Tempos de junção e mecanismos: linguagens de pesquisa são baseadas na junção tardia (no tempo de execução) de todos os nomes que ocorrem em pesquisas, enquanto que em linguagens de programação a junção é feita antecipadamente (no tempo de conexão e compilação). Logo, do ponto de vista de uma aplicação, pesquisas são simplesmente strings de caracteres.

[e] Java é uma linguagem de programação originalmente desenvolvida pela Sun Microsystems como uma componente chave da plataforma Java. A linguagem deriva muito de sua sintaxe das linguagens C e C++ mas tem um modelo de objetos mais simples e com menos facilidades de baixo-nível. Aplicações Java são tipicamente compiladas para bytecode e podem ser executadas em qualquer Java virtual machine (JVM), ou máquina virtual Java não importando a arquitetura do computador.

- Espaço de nomes e regras de escopo: pesquisas não percebem nomes que ocorrem em aplicações e vice-versa. Isto porque de maneira geral deve haver alguma interseção entre estes espaços de nomes, por exemplo, variáveis da aplicação devem parametrizar pesquisas. Extensões adicionais, com própria sintaxe, semântica e pragmatismo são necessários. Estas extensões são as cargas presentes no tamanho e legibilidade do código da aplicação. Além do mais, em linguagens de programação escopos de nomes são organizados hierarquicamente e processados por regras especiais baseadas em pilhas. Estas regras são ignoradas por uma linguagem de pesquisa. Isto ocasiona problemas como, por exemplo, chamadas recursivas a métodos.
- Coleções: bancos de dados armazenam coleções, por exemplo, tabelas, as quais são processadas por pesquisas. Em linguagens de programação coleções são ausentes ou severamente limitadas.
- Valores nulos: bancos de dados e suas linguagens de pesquisa possuem recursos especializados para armazenar e processar valores nulos. Tais recursos podem não estar presentes em linguagens de programação, o que impõe a introdução de alguns substitutos. Por exemplo, se algum valor em um banco de dados relacional pode ser nulo, mapear este valor para uma linguagem de programação requer duas variáveis: uma para armazenar a informação sobre a nulidade e outra para armazenar o valor.
- Facilidades de iteração: em linguagens de pesquisa iterações são obtidas com operadores macroscópicos de pesquisa, tais como seleção, projeção, junção, soma de coleções, etc. Em linguagens de programação, iterações são obtidas e codificadas explicitamente como loops (for, while, repeat, etc). Processar resultados de uma pesquisa em uma linguagem de programação requer mecanismos especiais, tais como cursores e iteradores.
- Persistência: linguagens de pesquisa endereçam dados persistentes (armazenados em um disco ou outra memória de vida longa), enquanto que linguagens de programação processam somente dados armazenados em uma memória volátil. Integrar pesquisa e linguagem de programação requer extensões especiais para parametrizar pesquisas através de variáveis voláteis e transmissão de dados persistentes para memória volátil e vice-versa.

- Pesquisas e expressões: existe um casamento inadequado entre pesquisas e expressões de linguagens de programação. Algumas pesquisas parecem ser expressões e vice-versa, mas existe uma subdivisão sintática forte nas mesmas, o que pode ser fracamente entendido por um desenvolvedor. Por exemplo, em algumas linguagens de pesquisa, 2 + 2 é uma pesquisa, mas é também uma expressão em uma linguagem de programação. Uma pesquisa não pode ser um parâmetro para um método, mas uma expressão pode. Existem outras restrições sintáticas, as quais podem causar algum caos em todo o ambiente de programação.
- Referências: se uma pesquisa é para ser usada para inserir, atualizar ou deletar dados, esta deve retornar referências para os dados armazenados, por exemplo, identificadores de dados ao invés de valores. De acordo com a semântica oficial do modelo relacional de bancos de dados, pesquisas retornam tabelas de valores, sem referências. Para construções de atualização de dados definidas em uma linguagem de programação tal semântica é inconsistente; na verdade, isto significa que pesquisas não podem ser usadas para atualizar dados.

As conseqüências da combinação mal sucedida afetam não somente a estética e ergonomia[f] do ambiente de programação. A combinação mal sucedida implica em um nível adicional de programação, com sua própria sintaxe, semântica e pragmatismo. Este nível faz com que o aprendizado da linguagem leve mais tempo, a programação tenda a mais erros, etc. As aplicações se tornam desnecessariamente maiores e menos legíveis. Além disso, este nível adicional pode causar pior performance e manutenção das aplicações.

Para evitar a combinação mal sucedida, a linguagem de pesquisa deve ser integrada na linguagem de programação, de forma a permitir que pesquisas sejam suavemente conectadas com construções e abstrações da programação. Esta tendência é especialmente vista em produtos tais como PL/SQL, T-QL e padrões SQL-1999 e SQL 2003.

Recentemente foram lançadas algumas propostas conhecidas como pesquisas nativas que tornam possível a utilização de pesquisas dentro de linguagens de programação típicas como C# (LINQ, tema deste trabalho) ou Java (db4o). Este mecanismo é advogado como 100% seguro, 100% verificado no tempo de compilação e 100% refatorável. Até pouco tempo atrás, havia muitas limitações às pesquisas nativas, em particular:

---

f Ergonomia ou fatores humanos refere-se à relação entre o homem e o ambiente. É uma disciplina científica que lida com o entendimento das relações entre os humanos e outros sistemas. Aplica teoria, princípios, dados e métodos para projetar buscando a otimização do bem-estar humano e a completa performance do sistema.

- Pesquisas criadas na sintaxe de uma linguagem de programação eram muito menos legíveis mesmo em comparação com SQL (a qual já não é tão legível). Existia muita interferência de informações do ponto de vista da expressão do objetivo de uma pesquisa.
- A força expressiva das pesquisas estava abaixo do poder aceitável, por exemplo, em pesquisas nativas não existiam operadores de junção, agrupamento, agregação, ordenação, quantificação, conjunto, etc. Na verdade, tais pesquisas englobavam somente simples seleções.
- Era impossível ter abstrações de persistência do lado do servidor tal como views, stored procedures, functions, etc.
- O grande problema era a performance obtida para grandes coleções de objetos: não haviam métodos de otimização de pesquisas nativas.

O projeto da linguagem de pesquisa integrada à linguagem de programação (LINQ) está em uma melhor posição porque este inclui extensões na sintaxe da linguagem de programação C#. Esta proposta estende a sintaxe e semântica da linguagem de programação em direção às linguagens de pesquisa.

2.4 Programação orientada a objetos

Em virtude da complexidade crescente do hardware e do software, pesquisadores estudaram como a qualidade do software poderia ser mantida. Daí, a programação orientada a objetos foi desenvolvida como uma tentativa de endereçar este problema através da forte ênfase em modularidade (unidades discretas de lógica de programação) e reusabilidade de código [10].

Programação orientada a objetos pode ser vista como uma coleção de objetos cooperadores, em oposição ao tradicional modelo de programação, no qual uma aplicação pode ser vista como uma lista de instruções. Na programação orientada a objetos, cada objeto é capaz de receber mensagens, processar dados e enviar mensagens para outros objetos. Cada objeto pode ser visto como uma pequena máquina individual com distinto papel e responsabilidade [11].

Ao tratar os módulos de software através de objetos, a programação orientada a objetos promove grande flexibilidade e manutenção do código. Com ênfase na modularidade, o código orientado a objetos é mais simples para desenvolver e mais fácil de entender, o que propicia uma análise mais direta do código.

Uma pesquisa realizada por Armstrong [12] sobre quase 40 anos de literatura voltada à computação, identificou um número de conceitos fundamentais, encontrados na maioria das definições de programação orientada a objetos. Estes e outros conceitos adicionais segundo [13] são definidos a seguir.

2.4.1 Classe e objeto

Classe é o conceito fundamental da programação orientada a objetos. Uma classe é uma estrutura de dados que combina o estado (campos ou atributos) e ações (métodos e outros membros de função) em uma única unidade. Uma classe provê uma definição para instâncias da classe criadas dinamicamente, também chamadas de objetos.

2.4.2 Variável e tipo

Uma variável representa um local de armazenamento, e cada variável tem um tipo que determina quais valores podem ser armazenados na mesma.

2.4.3 Membro

Um membro de uma classe pode ser estático ou instanciado. Um membro estático pertence à classe e um membro instanciado pertence ao objeto (instância da classe).

A Tabela 1 provê uma visão geral dos tipos de membros que uma classe pode conter:

Membro	**Descrição**
Constantes	Valores constantes associados à classe
Campos	Variáveis da classe
Métodos	Computações e ações que podem ser executadas pela classe
Propriedades	Ações associadas com leitura e escrita de propriedades nomeadas da classe
Indexadores	Ações associadas com instâncias de indexação da classe como um array
Eventos	Notificações que podem ser geradas por uma classe
Operadores	Conversões e operadores de expressão suportados pela classe
Construtores	Ações requeridas para inicializar instâncias de uma classe ou a própria classe
Destrutores	Ações a serem executadas antes que instâncias da classe sejam permanentemente destruídas
Tipos	Tipos aninhados declarados pela classe

Tabela 1 - Tipos de membros que uma classe pode conter

2.4.4 Acessibilidade

Cada membro de uma classe tem uma acessibilidade associada, a qual controla as regiões do código da aplicação que são aptas a acessar o membro. Existem cinco formas possíveis de acessibilidade. As acessibilidades estão sumarizadas na Tabela 2:

Acessibilidade	**Descrição**
público	Acesso não limitado
protegido	Acesso limitado à classe ou às classes derivadas a partir da classe
interno	Acesso limitado à aplicação
protegido interno	Acesso limitado à aplicação ou às classes derivadas a partir da classe
privado	Acesso limitado à classe

Tabela 2 - Formas de acessibilidade de um membro

2.4.5 Método

Um método é um membro que implementa uma computação ou ação que pode ser executada por um objeto ou classe. Métodos estáticos são acessados através da classe. Métodos de instância são acessados através de instâncias da classe.

2.4.6 Parâmetro

Um parâmetro é usado para passar um valor ou uma variável para um método. Os parâmetros de um método recebem seus valores a partir de argumentos que são especificados quando o método é invocado.

2.4.7 Troca de mensagem

A troca de mensagem é o processo pelo qual um objeto envia dados para outro objeto ou solicita que outro objeto invoque um método. Este processo é conhecido também como interfaceamento.

2.4.8 Herança

Uma classe herda os membros de sua classe base direta. Herança significa que uma classe implicitamente contém todos os membros de sua classe base direta, exceto, por exemplo, construtores, destrutores e construtores estáticos da classe base.

2.4.9 Encapsulamento

Encapsulamento é a combinação de propriedades e métodos para criar um objeto cuja estrutura interna permanece oculta, evitando a sua manipulação, mas permite o acesso a seus serviços através de mensagens transmitidas por uma interface.

2.4.10 Abstração

Abstração é o processo de simplificação da complexidade da realidade através do modelamento de classes apropriadas para o problema. É trabalhar no nível mais apropriado de herança para um dado aspecto de um problema. Por exemplo, um objeto pode ser tratado como uma instância de uma determinada classe na maioria do tempo, uma instância de uma subclasse quando for necessário acessar os atributos e comportamentos específicos da subclasse ou uma instância de uma classe base.

2.4.11 Polimorfismo

Polimorfismo é um mecanismo através do qual uma classe derivada pode estender ou especializar classes base. Permite que membros da classe derivada sejam tratados da mesma forma que os membros da classe base. Pode ser considerado também como a habilidade que objetos pertencentes a diferentes tipos têm para responder a chamadas de métodos de mesmo nome, cada um de acordo com um comportamento específico do tipo.

2.4.12 Interface

Uma interface define um contrato. Uma classe ou estrutura que implementa uma interface deve aderir a este contrato. Uma interface pode herdar de múltiplas interfaces base, e uma classe ou estrutura pode implementar múltiplas interfaces.

Interfaces podem conter métodos, propriedades, eventos e indexadores. A própria interface não provê implementações para seus membros. A interface meramente especifica os membros que devem ser supridos por classes ou estruturas que implementam a interface.

2.4.13 Delegate

Um delegate representa referências para métodos com uma lista particular de parâmetros e um tipo de retorno. Um delegate torna possível tratar métodos como entidades que podem ser atribuídas a variáveis e passadas como parâmetros. Um delegate é similar ao

conceito de ponteiro de função encontrado em outras linguagens de programação, mas diferentemente de um ponteiro de função, um delegate é orientado a objetos e seguro.

2.5 Banco de dados relacional

Um banco de dados é uma coleção de dados computacionais inter-relacionados que serve à necessidade de múltiplos usuários dentre de uma ou mais organizações [14].

Um banco de dados relacional é uma estrutura de dados que protege os dados e que permite que tais dados sejam manipulados de uma maneira previsível e resistente a erros. O modelo relacional, o qual tem sua raiz em princípios matemáticos da teoria de conjuntos e da lógica de predicados, suporta o fácil retorno de dados, reforça a exatidão e consistência dos dados, e provê uma estrutura de banco de dados independente de aplicações que acessam os dados armazenados [15].

A seguir são definidos segundo [16] alguns dos componentes de um banco de dados relacional que ajudam a organizar e estruturar os dados fazendo com que o banco de dados fique em conformidade com um conjunto de requerimentos.

2.5.1 Relação ou tabela

A parte principal de um banco de dados relacional é a relação. Uma relação (tabela) é um conjunto de colunas e linhas que representa uma única entidade construída a partir de dados relacionados. Cada tabela abrange um ou mais atributos (colunas). Dados são armazenados em tuplas (linhas). Uma linha é constituída por um conjunto de dados, e cada dado constitui uma coluna definida na tabela. Cada linha representa um registro.

2.5.2 Restrição

Restrição é um meio de restringir os tipos de dados que podem ser armazenados nas tabelas. É geralmente definida na forma de expressões que resultam em um valor booleano que indica se a restrição é válida. Uma restrição permite a implementação de regras de negócio dentro do banco de dados.

2.5.3 Domínio de dado

Um domínio de dado ou simplesmente domínio, é o conjunto de valores que podem ser atribuídos a uma coluna de uma tabela. Haja vista que o domínio restringe os valores, o

mesmo pode ser considerado uma restrição, mas como colunas devem especificar um domínio, este pode ser considerado parte da definição da tabela.

2.5.4 Chave primária

Uma linha de uma tabela representa um registro e seus dados associados. A chave primária é um tipo de restrição que impede a duplicação de uma linha, ou informação crítica sobre a linha dentro de uma tabela. Uma chave primária pode ser criada a partir de um conjunto de colunas da tabela.

2.5.5 Chave estrangeira

Uma chave estrangeira é uma restrição referencial entre duas tabelas. Pode ser considerada uma referência para uma chave primária. Isto significa que uma linha da tabela que faz a referência tem os valores da chave primária da linha pertencente à tabela que está sendo referenciada. A restrição referencial cria um relacionamento entre as tabelas.

2.5.6 Stored procedure

Uma stored procedure (rotina armazenada) também chamada de sproc ou SP é uma sub-rotina[g] disponível para aplicações que acessam um banco de dados relacional. É geralmente usado para executar código relativo a operações comuns, tais como inserir uma linha em uma tabela ou disponibilizar informação estatística sobre padrões de uso do banco de dados. Pode também ser usada para validar dados ou para controlar mecanismos de acesso ao banco de dados. Além do mais, uma stored procedure pode ser usada para consolidar e centralizar a lógica de programação implementada em aplicações. Processamentos complexos que requerem a execução de várias instruções SQL podem ser implementados através de uma stored procedure.

2.5.7 View

Uma view (visão) é uma tabela virtual criada a partir do resultado de uma pesquisa. É computada ou combinada a partir de uma ou mais tabelas do banco de dados. É geralmente usada para restringir os dados contidos em uma tabela, unir e simplificar múltiplas tabelas em

[g] Uma sub-rotina (função, método, procedimento ou subprograma) é uma porção de código dentro de um programa maior. Realiza uma tarefa específica e pode ser relativamente independente do restante do código.

uma única tabela virtual, agir como uma tabela agregada, onde dados são agregados através de funções como soma e média e limitar a exposição de tabelas para as aplicações.

2.5.8 User defined function

Uma user defined function (função definida pelo usuário) ou UDF provê um mecanismo para estender a funcionalidade de um de banco de dados através da adição de uma função que pode ser usada em instruções SQL. Existem dois tipos de funções: funções que retornam um valor escalar e funções que retornam tabelas. Uma função escalar retorna somente um valor único ou nulo e uma função de tabela retorna uma tabela que consiste de zero ou mais linhas e cada linha pode ter uma ou mais colunas.

2.6 .NET Framework

O .NET Framework é um componente integral do Windows que suporta a construção e execução da próxima geração de aplicações e serviços web [17]. O .NET Framework é projetado para satisfazer os seguintes objetivos:

- Prover um ambiente consistente de programação orientada a objetos não importando se o código objeto é armazenado e executado localmente, executado localmente, mas distribuído na Internet, ou executado remotamente.
- Prover um ambiente consistente para a execução de código que minimiza os conflitos de distribuição e versão de software.
- Prover um ambiente de execução de código que promove a execução segura de código, incluindo código criado por uma terceira parte desconhecida ou semiconfiável.
- Prover um ambiente de execução de código que elimina os problemas de performance de ambientes escritos ou interpretados.
- Tornar a experiência do desenvolvedor consistente através de vários tipos de aplicações, tais como aplicações baseadas no Windows e aplicações baseadas na web.
- Construir toda a comunicação com os padrões da indústria para garantir que o código baseado no .NET Framework possa ser integrado com outros código.

O .NET Framework tem dois componentes principais: o common language runtime (CLR) ou tempo de execução de linguagem comum e a base class library (BCL) ou biblioteca de classes base. O tempo de execução de linguagem comum é o fundamento do .NET

Framework. Pode-se pensar no CLR como um agente que gerencia o código no tempo de execução, provendo serviços centrais tais como gerenciamento de memória, gerenciamento de processos, etc., enquanto também reforça a segurança de tipos e outras formas de exatidão de código que promove segurança e robustez. De fato, o conceito de gerenciamento de código é um princípio fundamental do CLR. Código que tem como alvo o CLR é conhecido como código gerenciável, enquanto que código que não tem como alvo o CLR é conhecido como código não gerenciável. A biblioteca de classes base, o outro componente principal do .NET Framework, é uma coleção compreensiva de tipos orientados a objetos que são reusáveis e que podem ser empregados no desenvolvimento de aplicações que variam de tradicionais linhas de comando (console applications), interfaces de usuário ou graphical user interfaces (GUI) a até mesmo aplicações baseadas nas últimas inovações providas pela tecnologia ASP.NET, tais como formulários web e serviços web baseados em XML.

A Figura 2 ilustra a relação entre o tempo de execução de linguagem comum e a biblioteca de classes base para as aplicações e para o sistema em geral. Também é mostrado como o código gerenciado opera dentro de uma ampla arquitetura.

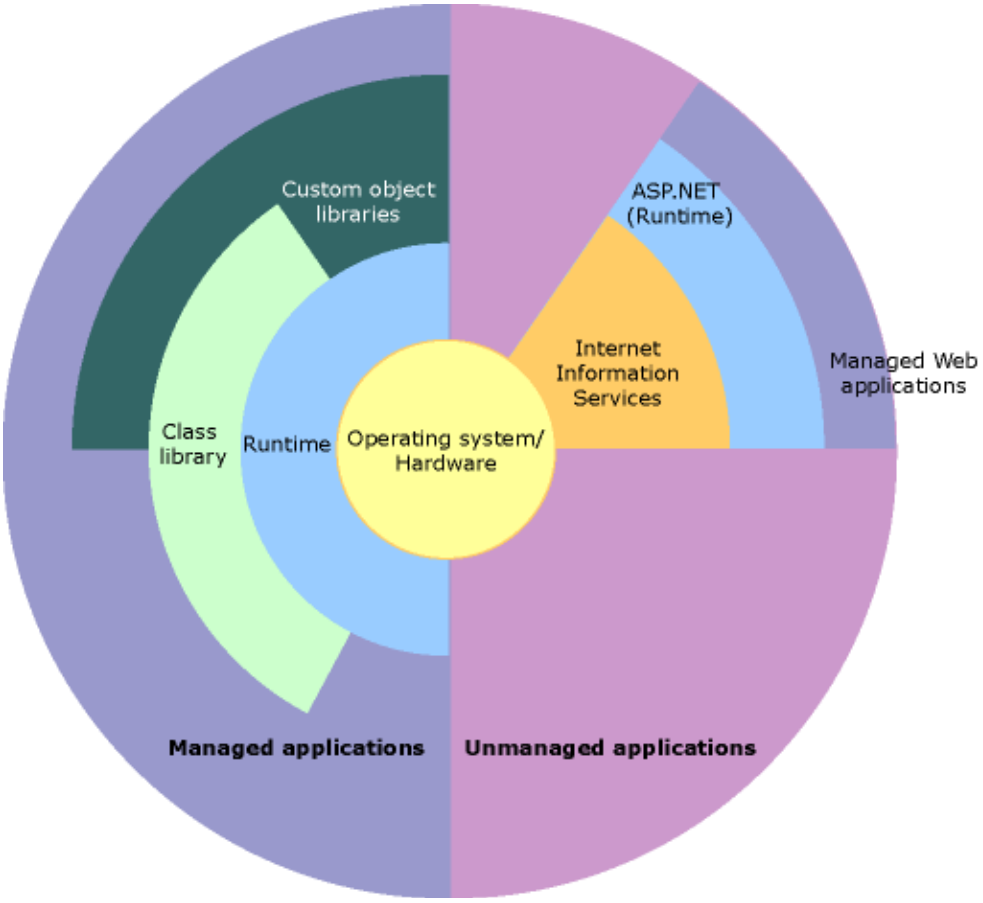

Figura 2 - .NET Framework em contexto

Um objetivo principal do projeto .NET Framework é prover suporte à independência de plataforma. Isto é, uma aplicação escrita para usar o framework deveria executar sem mudança em qualquer tipo de computador para o qual o framework fosse implementado, porém, o .NET Framework só foi implementado por completo no sistema operacional Windows. Outros implementaram porções do framework em sistemas não Windows, mas ainda hoje estas implementações não estão completas e nem são amplamente utilizadas. Dessa forma, o suporte multi-plataforma ainda não foi alcançado.

2.6.1 Principais recursos

Os principais recursos do .NET Framework são:

- Interoperabilidade: haja vista que a interação entre novas e antigas aplicações é comumente requerida, o .NET Framework provê meios para acessar funcionalidades que estão implementadas em programas que executam fora do ambiente .NET.
- Mecanismo comum do tempo de execução: linguagens de programação no .NET Framework são compiladas em uma linguagem intermediária conhecida como common intermediate language (CIL) ou linguagem intermediária comum. Na implementação original, esta linguagem intermediária não é interpretada, ao invés disso, é compilada de um modo conhecido como compilação instantânea ou just in-time compilation (JIT) para código nativo. A combinação destes conceitos é chamada de common language infrastructure (CLI) ou infra-estrutura de linguagem comum; A implementação da CLI é conhecida como tempo de execução de linguagem comum ou common language runtime (CLR).
- Independência de linguagem: o .NET Framework introduz um sistema comum de tipo ou common type system (CTS). A especificação do CTS define todos os possíveis tipos de dados e construções de programação suportadas pela CLR e como estas podem ou não interagir umas com as outras. Em virtude desse recurso, o .NET Framework suporta o desenvolvimento de aplicações em múltiplas linguagens de programação.
- Biblioteca de classes base: a biblioteca de classes base ou base class library (BCL), algumas vezes também mencionada como biblioteca de classes da estrutura ou framework class library (FCL), é uma biblioteca de tipos disponíveis para todas as linguagens que usam o .NET Framework. A BCL provê classes que encapsulam

um vasto número de funções comuns, incluindo leitura e escrita de arquivos, desenho de imagens, interação com banco de dados e XML, manipulação de documentos, etc.

- Distribuição simplificada: a instalação de programas de computador deve ser cuidadosamente gerenciada para garantir a não interferência com software previamente instalado, e a concordância com os severos requerimentos de segurança que têm aumentado progressivamente. O .NET Framework inclui recursos de projeto e ferramentas que ajudam a endereçar estes requerimentos.
- Segurança: o .NET Framework permite que o código seja executado em diferentes níveis de segurança sem o uso de um mecanismo de segurança separado.

2.6.2 Arquitetura

A arquitetura do .NET Framework pode ser dividida em: infra-estrutura de linguagem comum, bibliotecas de código parcialmente compiladas ou assemblies, metadados e biblioteca de classes base.

2.6.3 Infra-estrutura de linguagem comum

O componente mais importante do .NET Framework está situado dentro da infra-estrutura de linguagem comum ou common language infrastructure (CLI). O propósito da CLI é prover uma plataforma agnóstica[h] de linguagem para o desenvolvimento e execução de aplicações, incluindo, mas não limitada a componentes para o tratamento de exceções, coleta de lixo, segurança e interoperabilidade. A implementação da CLI é chamada de common language runtime (CLR) ou tempo de execução de linguagem comum. A CLR é composta de quatro partes primárias:

- Common Type System (CTS) ou Sistema Comum de Tipos;
- Common Language Specification (CLS) ou Especificação de Linguagem Comum;
- Just-in-Time Compiler (JIT) ou Compilador[i] Instantâneo;
- Virtual Execution System (VES) ou Sistema de Execução Virtual.

A Figura 3 ilustra a infra-estrutura de linguagem comum:

[h] Agnóstica neste caso significa que as linguagens não dependem de uma plataforma de computação particular.
[i] Um compilador é um programa de computador (ou conjunto de programas) que traduz texto escrito em uma linguagem de programação (a linguagem fonte) para outra linguagem de computador (linguagem alvo).

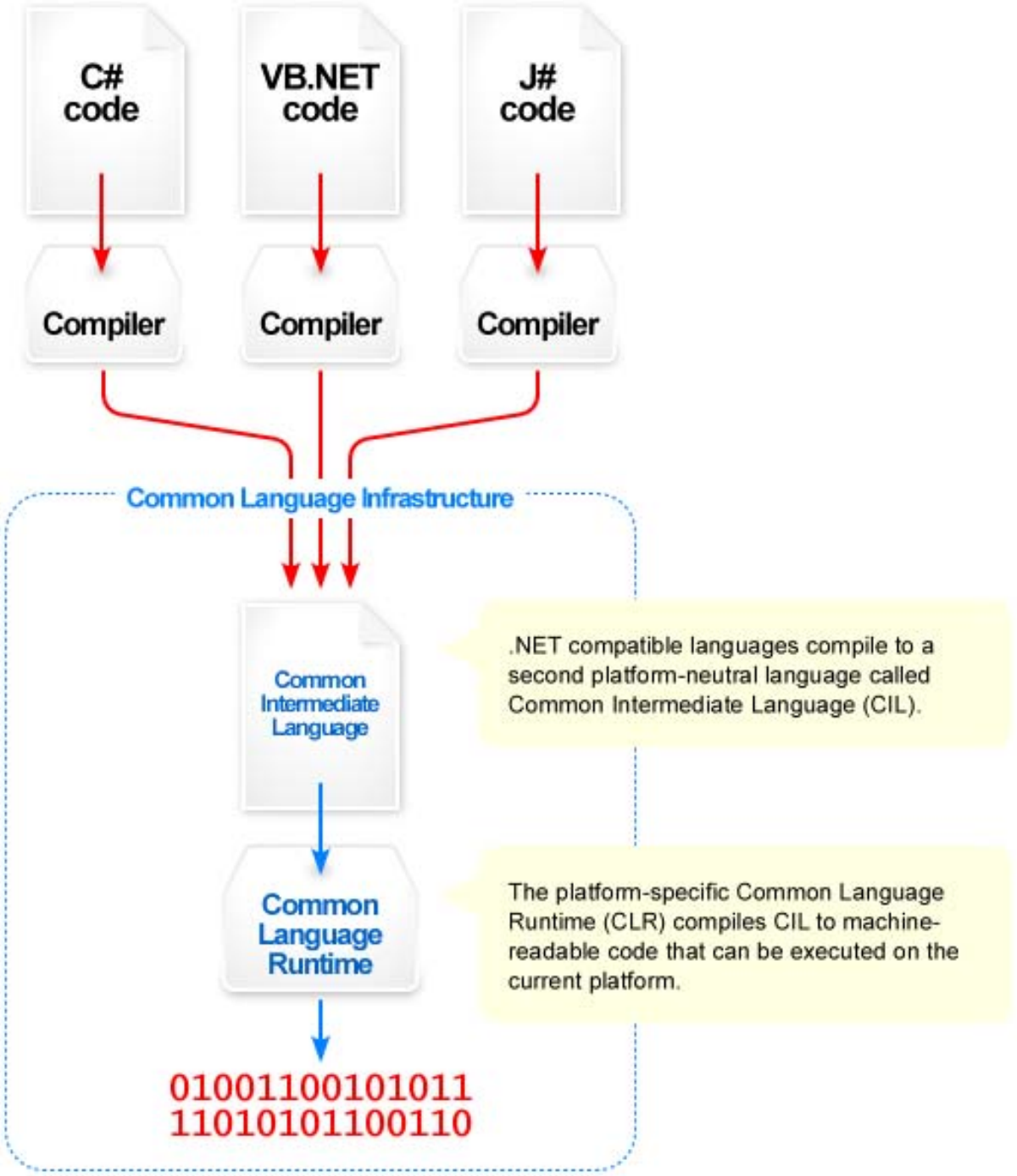

Figura 3 - Visão geral da infra-estrutura de linguagem comum

2.6.4 Assemblies

A linguagem intermediária comum ou common intermediate language (CIL) é hospedada em bibliotecas de código parcialmente compiladas, assemblies do .NET, as quais para a implementação do Windows significam um executável portátil ou portable executable (PE), um arquivo (EXE ou DLL). Estes formam as unidades de distribuição, versão e segurança do .NET.

2.6.5 Metadados

Toda linguagem intermediária comum (CIL) é autodescrita através de metadados. O CLR checa os metadados para garantir que o correto método seja chamado. Metadados são geralmente gerados pelos compiladores das linguagens e também descrevem toda a informação sobre o assembly.

2.6.6 Biblioteca de classes base

A biblioteca de classes base ou base class library (BCL) é uma biblioteca de classes disponível para todas as linguagens que usam o .NET Framework. A BCL provê o encapsulamento das mais diversas funções utilizadas pelos programas desenvolvidos com o suporte do .NET Framework.

2.7 SQL

SQL (Structured Query Language) ou Linguagem de Pesquisa Estruturada é uma linguagem de pesquisa que foi originalmente criada nos moldes de uma linguagem declarativa[j]. É diferente das linguagens de programação, tais como C, C# ou Java, as quais foram criadas nos moldes de uma linguagem procedural[k] [15].

SQL foi criada para pesquisar dados, criar, modificar e gerenciar esquemas e controlar o acesso aos objetos de um banco de dados relacional.

SQL é tratada como uma sub-linguagem porque esta é geralmente usada em associação com linguagens de programação, as quais não foram projetadas nativamente para manipular dados armazenados em um banco de dados.

2.8 C#

C# (pronunciada “C Sharp”) é uma linguagem de programação simples, moderna, orientada a objetos e segura. C# tem sua raiz na família de linguagens C e é imediatamente familiar para desenvolvedores que programam em C, C++ e Java [13]. Foi influenciada por outras linguagens de programação, mais notadamente Java. Sua ênfase é a simplificação.

---

[j] O paradigma da programação declarativa ou funcional explicitamente especifica o objetivo de uma computação, o que a computação deve fazer, deixando a implementação do algoritmo para atingir o objetivo por conta da linguagem de programação.

[k] O paradigma da programação procedural ou imperativa descreve uma computação como instruções que mudam o estado de um programa. Desta forma, é expresso como a computação deve acontecer, ou seja, explicitamente especifica-se um algoritmo para atingir um objetivo.

C# adicionalmente inclui suporte à programação orientada a componentes. A construção contemporânea de software depende cada vez mais de componentes de software na forma de pacotes de funcionalidade independentes e autodescritivos. O principal de tais componentes é que eles apresentam um modelo de programação com propriedades, métodos e eventos, possuem atributos que provêem informação declarativa sobre os componentes e incorporam sua própria documentação. C# provê construções de linguagem para suportar diretamente estes conceitos, o que faz de C# uma linguagem muito natural para criar e usar componentes de software.

Muitos dos recursos de C# ajudam na construção de aplicações robustas e duráveis: o coletor de lixo automaticamente recupera a memória ocupada por objetos não usados; o tratamento de exceções provê uma aproximação estruturada e extensível para detectar e tratar erros; e a arquitetura segura da linguagem torna impossível a leitura de variáveis não inicializadas, a indexação de arrays fora de seus limites, ou a execução de conversões não controláveis.

C# possui um sistema unificado de tipos. Todos os tipos de C# incluindo tipos primitivos tais como int e double herdam de um único tipo objeto. Assim, todos os tipos compartilham um conjunto de operações comuns e valores de qualquer tipo podem ser armazenados, transportados e operados através de uma maneira consistente.

3 METODOLOGIA

São apresentadas as extensões de linguagem que formam a base da linguagem de pesquisa integrada à linguagem de programação.

3.1 Extensões de linguagem

Extensões de linguagem foram adicionadas nas versões 2.0 e 3.0 da linguagem C#. Estas extensões formam a base da linguagem de pesquisa integrada à linguagem de programação. Veja a seguir uma introdução a tais extensões.

3.1.1 Genéricos

Genéricos introduzem o conceito de parâmetros de tipo, os quais tornam possível o desenho de classes, estruturas, interfaces, delegates, e métodos que adiam a especificação de um ou mais tipos até o momento em que os componentes sejam declarados e instanciados pelo código cliente. Por exemplo, usando um parâmetro de tipo genérico T, é possível escrever uma única classe que outro código cliente pode usar sem incorrer o custo ou risco de operações de conversão e empacotamento (boxing) durante o tempo de execução.

O Código 4 mostra o uso de generics:

```
class Generic<T> { }

class Foo { }

Generic<int> generic1 = new Generic<int>();
Generic<Foo> generic2 = new Generic<Foo>();
```

Código 4 - Genéricos

Genéricos maximizam a reusabilidade do código, aumentam a segurança do tipo de dado e a performance; o uso mais comum de genéricos é na criação de classes de coleção.

A biblioteca de classes do .NET Framework contém várias coleções genéricas. Estas devem ser usadas sempre que possível ao invés de classes como ArrayList.

É possível criar classes, estruturas, interfaces, delegates, métodos e eventos genéricos;

Classes genéricas podem restringir o acesso a métodos para determinados tipos de dados.

Informações dos tipos usados em um tipo de dado genérico podem ser obtidas no tempo de execução através de reflexão (reflection).

3.1.2 Métodos anônimos

Métodos anônimos permitem que um bloco de código seja passado como um parâmetro de um delegate. Ajudam a reduzir a despesa com código na instanciação de delegates eliminando a necessidade de criar um método separado.

Especificar um bloco de código no lugar de um delegate pode ser útil em uma situação em que criar um método parece uma despesa desnecessária.

Um bom exemplo do uso de um método anônimo pode ser visto ao iniciar um novo processo (thread).

O Código 5 mostra o uso de métodos anônimos:

```
Thread thread = new Thread(delegate() { Console.Write("Hello World!"); });
int n = 0; Del d = delegate() { Console.Write("Copy #: {0}", n++); };
```

Código 5 - Métodos anônimos

No código acima, a thread também contém o código que será executado, sem a necessidade de criar um método adicional para o delegate.

O escopo dos parâmetros de um método anônimo é o bloco de código do método anônimo.

É um erro ter uma instrução goto, break, ou continue dentro do bloco de um método anônimo cujo alvo está fora do bloco. É também um erro ter uma instrução de salto, tal como goto, break ou continue dentro do bloco de um método anônimo cujo alvo está dentro do bloco.

As variáveis locais e parâmetros cujo escopo contém uma declaração de um método anônimo são chamados de exteriores ou variáveis capturadas do método anônimo. Diferentemente de variáveis locais, o tempo de vida de uma variável externa se estende até que o delegate que referência o método anônimo seja elegível à coleta de lixo (garbage collection). Uma referência a n é capturada no momento em que o delegate é criado.

Uma coleta de lixo não pode acessar os parâmetros ref ou out de um escopo externo.

Nenhum código não seguro pode ser acessado dentro do bloco de uma coleta de lixo.

3.1.3 Iteradores

Um Iterador é um método, um assessor get ou um operador que permite o suporte à iteração foreach em uma classe ou estrutura sem a necessidade de implementar a interface IEnumerable por completo. Ao invés disso, é preciso prover apenas um iterador, o qual simplesmente atravessa as estruturas de dados da classe. Quando o compilador detecta o

iterador, este irá automaticamente gerar os métodos Current, MoveNext e Dispose da interface IEnumerable ou IEnumerable<T>.

O Código 6 mostra o uso de iteradores:

```
public class Weekdays : IEnumerable
{
  string[] weekdays = { "Mon.", "Tue.", "Wed.", "Thur.", "Fri." };

  public IEnumerator GetEnumerator()
  {
    for(int i = 0; i < weekdays.Length; i++)
      yield return weekdays[i];
  }
}

Weekdays weekdays = new Weekdays();

foreach(string day in weekdays)
  Console.WriteLine(day);
```

Código 6 - Iteradores

Um iterador é uma seção de código que retorna uma seqüência ordenada de valores do mesmo tipo.

Um iterador pode ser usado como o corpo de um método, um operador, ou um assessor get.

O código do iterador usa a instrução yield return para retornar cada elemento um após o outro. yield break termina a iteração.

Múltiplos iteradores podem ser implementados em uma classe. Cada iterador dever ter um único nome da mesma forma que qualquer membro de classe, e pode ser invocado pelo código cliente em uma instrução foreach.

O tipo de retorno de um iterator dever ser IEnumerable, IEnumerator, IEnumerable<T> ou IEnumerator<T>.

A keyword[a] (palavra-chave) yield é usada para especificar o valor, ou valores retornados. Quando a instrução yield return é alcançada, a localização corrente é armazenada. A execução é reiniciada a partir deste local na próxima vez que o iterador for chamado.

Iteradores são especialmente úteis em classes de coleção, pois provêem uma maneira fácil de iterar estruturas de dados não triviais tais como árvores binárias.

[a] Uma keyword é uma palavra ou identificador que tem um significado particular para a linguagem de programação. O significado de uma keyword pode variar de linguagem para linguagem.

3.1.4 Tipos parciais

Tipos parciais permitem que as definições de classes, estruturas e interfaces sejam divididas em múltiplas partes, as quais podem ser armazenadas em diferentes arquivos de código para melhor desenvolvimento e manutenção da aplicação.

Permite também que vários desenvolvedores trabalhem em paralelo em diferentes partes da classe, estruturas ou interfaces de modo que diferentes aspectos de uma classe, estrutura ou interface complexa sejam mantidos em arquivos separados.

A divisão de uma classe, estrutura ou interface em vários arquivos pode ser útil quando se está trabalhando com projetos grandes.

Tipos parciais permitem a separação de tipos gerados pela máquina e tipos escritos pelo usuário, o que torna fácil a complementação do código gerado por uma ferramenta.

O Código 7 mostra o uso de tipos parciais:

```
partial class Developer
{
  public void DoWork()
  {
    Console.WriteLine("Developer is working.");
  }
}

// This partial class could be in a separate code file
partial class Developer
{
  public void GoLunch()
  {
    Console.WriteLine("Developer is lunching.");
  }
}

Developer developer = new Developer();

developer.DoWork();
developer.GoLunch();
```

Código 7 - Tipos parciais

A keyword partial indica que outras partes da classe podem ser definidas dentro do namespace[b].

Todas as partes devem usar a keyword partial.

Todas as partes devem estar disponíveis no tempo de compilação para formar o tipo final.

Todas as partes devem ter a mesma acessibilidade, tal como público, privado, etc.

[b] Um namespace é um contêiner ou ambiente abstrato criado para armazenar um agrupamento lógico de identificadores únicos, por exemplo, nomes.

Se qualquer uma das partes for declarada como abstrata, então todo o tipo é considerado abstrato. Se qualquer uma das partes for declarada selada, então todo o tipo é considerado selado. Se qualquer uma das partes declara um tipo base, então todo o tipo herda o tipo base.

Todas as partes que especificam uma classe base devem concordar entre si, mas partes que omitem uma classe base ainda herdam o tipo base. Partes podem especificar diferentes interfaces base, e o tipo final implementa todas as interfaces listadas por todas as declarações parciais.

Qualquer membro de classe, estrutura, ou interface declarado em uma definição parcial está disponível para todas as outras partes. O tipo final é uma combinação de todas as partes no tempo de compilação.

3.1.5 Tipos anuláveis

Tipos anuláveis são instâncias da estrutura System.Nullable<T>. Um tipo anulável pode representar a correta variação de valores para o tipo de valor base, mais um valor adicional nulo. Por exemplo, a um Nullable<Int32> pronunciado “anulável do tipo Int32” pode ser atribuído qualquer valor a partir de -2147483648 até 2147483647, ou o valor nulo. Um Nullable<bool> pode receber os valores verdadeiro, falso, ou nulo. A habilidade para atribuir um valor nulo para os tipos numéricos e booleanos é especialmente útil quando se está lidando com bancos de dados e outros tipos de dados que contêm elementos que podem não ter um valor atribuído. Por exemplo, um campo booleano em um banco de dados pode armazenar os valores verdadeiro ou falso, ou este pode ser indefinido.

O Código 8 mostra o uso de tipos anuláveis:

```
int? num = null;

int? i = 10; double? d = 17.17;

// 0 is assigned to y
int y = num.GetValueOrDefault();

if(num.HasValue == true)
  Console.WriteLine("num = {0}", num.Value);
else
  Console.WriteLine("num = null");

// num.Value throws an exception if num.HasValue is false
try
{
  y = num.Value;
}
catch(InvalidOperationException e)
{
  Console.WriteLine(e.Message);
}

int l = num ?? -1;

Nullable<Nullable<int>> n; // error: nested nullable types
```

Código 8 - Tipos anuláveis

Tipos anuláveis representam variáveis do tipo de valor, às quais pode ser atribuído o valor nulo. Não é possível criar um tipo anulável baseado em um tipo de referência. Tipos de referência já suportam o valor nulo.

A sintaxe T? é a abreviação de Nullable<T>, onde T é um tipo de valor. As duas formas podem ser trocadas.

Um valor para um tipo anulável deve ser atribuído da mesma forma que para um tipo comum de valor.

A função Nullable<T>.GetValueOrDefault deve ser usada para retornar o valor atribuído ou o valor padrão do tipo de valor se o valor for nulo.

As propriedades read-only (somente-leitura) HasValue e Value devem ser usadas para testar para um nulo e retornar o valor. A propriedade HasValue retorna verdadeiro se a variável contém um valor, ou falso se este for nulo. A propriedade Value retorna um valor se um valor for atribuído à variável, caso contrário, uma exceção é lançada.

O valor padrão para uma variável de tipo anulável atribui falso em HasValue. O valor é indefinido.

O operador ?? deve ser usado para atribuir um valor padrão que será aplicado quando um tipo anulável cujo valor corrente é nulo for atribuído para um valor não anulável.

Tipos anuláveis aninhados não são permitidos.

3.1.6 Expressões de pesquisa

Para um desenvolvedor, a parte mais visível da linguagem de pesquisa integrada à linguagem de programação é a expressão de pesquisa. Expressões de pesquisa são escritas em uma sintaxe de pesquisa declarativa. Usando a sintaxe de pesquisa, é possível realizar operações de filtragem, ordenação e agrupamento em fontes de dados com o mínimo de código. O mesmo molde de uma expressão de pesquisa básica pode ser usado para pesquisar e transformar dados em bancos de dados SQL, DataSets, documentos XML, streams e coleções.

O Código 9 mostra o uso de expressões de pesquisa:

```
// Specifies the data source
int[] scores = new int[] { 6, 10, 83, 23, 4, 49, 4, 12, 29 };

// Defines the query expression
IEnumerable<int> query = from s in scores
                         where s > 17
                         select s;

// Executes the query
foreach(int score in query)
  Console.WriteLine(score);
```

Código 9 - Expressões de pesquisa

Note no código acima a operação de pesquisa completa, incluindo a criação da fonte de dados, a definição da expressão de pesquisa, e a execução da pesquisa em uma instrução foreach.

Expressões de pesquisa podem ser usadas para pesquisar como também para transformar dados de qualquer fonte de dados compatível com a linguagem de pesquisa integrada à linguagem de programação. Por exemplo, uma única pesquisa pode retornar dados de um banco de dados, juntá-los com algum dado em uma coleção presente na memória e produzir um stream de saída XML.

Expressões de pesquisa são fáceis de aprender porque elas usam muitas construções da linguagem de programação, as quais já são familiares.

As variáveis em uma expressão de pesquisa são todas fortemente tipificadas, embora em muitos casos não exista a necessidade de explicitamente prover o tipo porque o compilador pode inferir o mesmo.

Uma pesquisa não é executada até que ocorra uma iteração sobre a variável de pesquisa em uma instrução foreach.

No tempo de compilação, expressões de pesquisa são convertidas para chamadas aos métodos dos operadores de pesquisa padrão de acordo com as regras estabelecidas na especificação da linguagem de programação. Qualquer pesquisa que possa ser expressa

através de uma sintaxe de pesquisa pode também ser expressa usando a sintaxe de método. Todavia, na maioria dos casos a sintaxe de pesquisa é mais legível e concisa.

Como regra geral, ao escrever pesquisas, recomenda-se que seja usada a sintaxe de pesquisa sempre que possível e a sintaxe de método sempre que necessário. Não há diferença semântica ou de performance entre as duas formas. Expressões de pesquisa são geralmente mais legíveis do que as expressões equivalentes escritas em sintaxe de método.

Algumas operações de pesquisa, tais como Count ou Max não possuem cláusulas de expressões de pesquisa equivalentes e devem então ser expressas como uma chamada de método. A sintaxe de método pode ser combinada com a sintaxe de pesquisa através do uso de parênteses.

Expressões de pesquisa podem ser compiladas para árvores de expressão ou para delegates, dependendo do tipo no qual a pesquisa é aplicada. Pesquisas do tipo IEnumerable<T> são compiladas para delegates. Pesquisas dos tipos IQueryable e IQueryable<T> são compiladas para árvores de expressão.

3.1.7 Propriedades automaticamente implementadas

Propriedades automaticamente implementadas ou propriedades auto-implementadas tornam as declarações de propriedades mais concisas em casos onde nenhuma lógica adicional é necessária nos assessores das propriedades. Quando uma propriedade é declarada, o compilador cria um campo de suporte anônimo e privado que não é acessível, exceto através dos assessores get e set da propriedade.

O Código 10 mostra o uso de propriedades auto-implementadas:

```
public int DeveloperID { get; private set; }
public string Name { get; set; }
```

Código 10 - Propriedades automaticamente implementadas

Propriedades auto-implementadas devem declarar ambos os assessores get e set.

Para criar uma propriedade auto-implementada somente-leitura, marca-se o assessor set como privado.

Atributos não são permitidos em propriedades auto-implementadas. Caso seja necessário usar um atributo no campo de suporte de uma propriedade, deve-se criar uma propriedade regular para tanto.

3.1.8 Variáveis locais implicitamente tipificadas

Variáveis locais podem ter um “tipo” inferido var ao invés de um tipo explícito. A keyword var instrui o compilador a inferir o tipo da variável a partir da expressão do lado direito da instrução de inicialização. O tipo inferido pode ser um tipo interno, anônimo, definido pelo usuário, definido na biblioteca de classes do .NET Framework, ou qualquer expressão.

O Código 11 mostra o uso de variáveis locais implicitamente tipificadas:

```
var i = 7; // i is compiled as an int
var s = "Hello World!"; // s is compiled as a string
var list = new List<int>(); // list is compiled as List<int>
for(var v = 1; v < 10; v++) {...}
foreach(var item in list){...}
using(var file = new StreamReader("C:\\file.txt")) {...}
var x = x++; // compile time error
```

Código 11 - Variáveis locais implicitamente tipificadas

É importante notar que a keyword var não significa “variant” (variante) e não indica que a variável é fracamente tipificada. O único significado é que o compilador determina e atribui o tipo mais apropriado.

A keyword var pode ser usada nos seguintes contextos: variáveis locais (variáveis declaradas no escopo de métodos), inicialização de uma instrução for, inicialização de uma instrução foreach e em uma instrução using.

A keyword var somente pode ser usada quando uma variável local é declarada e inicializada na mesma instrução; a variável não pode ser inicializada com null.

A keyword var não pode ser usada em campos no escopo da classe.

Variáveis declaradas usando var não podem ser usadas na expressão de inicialização.

Múltiplas variáveis implicitamente tipificadas não podem ser inicializadas na mesma instrução.

Se um tipo nomeado var está no escopo, então ocorrerá um erro no tempo de compilação se houver a tentativa de inicializar uma variável local com a keyword var.

O único cenário no qual uma variável local dever ser implicitamente tipificada com var é na inicialização de tipos anônimos.

Pode ser útil usar var em expressões de pesquisa, pois o exato tipo construído da variável de pesquisa é difícil de ser determinado.

A keyword var pode também ser útil quando um tipo específico da variável é tedioso de ser digitado, ou é óbvio, ou não ajuda muito na leitura do código. Um exemplo onde var é útil é em tipos genéricos aninhados.

O uso de var também tem o potencial de tornar o código mais difícil de entender para outros desenvolvedores. Por esta razão, a documentação original da linguagem C# usa var somente quando necessário.

3.1.9 Métodos de extensão

Métodos de extensão permitem a adição de métodos em tipos já existentes sem a criação de um novo tipo derivado, recompilação ou modificação do tipo original. Métodos de extensão se enquadram em um tipo especial de método estático, mas eles são chamados como se fossem métodos de instância no tipo estendido. Para o código cliente, não existe diferença aparente entre chamar um método de extensão e os métodos que já estão definidos em um tipo.

Os métodos de extensão mais comuns são os dos operadores de pesquisa padrão que adicionam funcionalidade de pesquisa para os tipos existentes IEnumerable e IEnumerable<T>.

O primeiro parâmetro de um método de extensão especifica em que tipo o método opera, e o parâmetro é precedido pelo modificador this. Métodos de extensão só estão no escopo quando são explicitamente importados para dentro do código fonte com a diretiva using.

O Código 12 mostra o uso de métodos de extensão:

```
public static int WordCount(this string str)
{
  return str.Split(null).Length;
}

string s = "Hello World!"; int i = s.WordCount();
```

Código 12 - Métodos de extensão

O método de extensão é invocado no código com a sintaxe de método de instância. Todavia, a linguagem intermediária gerada pelo compilador traduz o código em uma chamada de método estático. Assim, o princípio de encapsulamento não está sendo realmente violado. De fato, métodos de extensão não podem acessar variáveis privadas no tipo em que eles estão estendendo.

Em geral, chamar métodos de extensão será mais comum do que implementar novos

métodos de extensão. Haja vista que métodos de extensão são chamados usando a sintaxe de métodos de instância, nenhum conhecimento especial é requerido para usá-los no código cliente.

Em geral, é recomendado que métodos de extensão sejam usados de forma reduzida e somente quando forem necessários. Sempre que possível, o código cliente que deve estender um tipo existente deve criar um novo tipo derivado do tipo existente. Ao usar um método de extensão para estender um tipo existente, o qual o código fonte não pode ser mudado, corre-se o risco de falha no método de extensão em virtude de uma mudança na implementação do tipo.

Ao implementar métodos de extensão para um dado tipo, é importante lembrar que um método de extensão nunca será chamado se este tiver a mesma assinatura de um método definido no tipo. Métodos de extensão são trazidos para dentro do escopo no nível do namespace. Por exemplo, se existem múltiplas classes estáticas que contêm métodos de extensão em um mesmo namespace chamado Extensions, todas serão trazidas para o escopo através do uso da diretiva using Extensions.

3.1.10 Métodos parciais

Métodos parciais são usados por ferramentas geradoras de código. Permitem que os desenvolvedores modifiquem o código automaticamente gerado sem que ocorra a perda das modificações caso o código seja recriado pela ferramenta [18].

Quando implementados em conjunto com classes parciais, o nome dos métodos parciais podem ser reservados de forma a permitir que tais métodos sejam implementados posteriormente pelo consumidor da classe. Isto oferece ao desenvolvedor um lugar onde será possível adicionar código que nunca será reescrito mesmo que a ferramenta que gerou o código seja executada novamente.

O Código 13 mostra o uso de métodos parciais:

```
// Autogenerated class
public partial class PartialClass
{
  partial void PartialMethod();

  public void CallPartialMethod()
  {
    PartialMethod();
  }
}

// This class is written by hand
public partial class PartialClass
{
  partial void PartialMethod()
  {
    Console.WriteLine("Second half");
  }
}

class Program
{
  static void Main(string[] args)
  {
    PartialClass p = new PartialClass();

    p.CallPartialMethod();
  }
}
```

Código 13 - Métodos parciais

No código acima, a primeira instância da classe PartialClass foi automaticamente gerada por uma ferramenta. Note que a classe é declarada parcial da mesma forma que o método PartialMethod. Note também que o método PartialMethod contém somente um cabeçalho sem a implementação do método. Isto pode ser considerado um convite para que o desenvolvedor implemente o corpo deste método.

Caso o desenvolvedor aceite o convite e crie a segunda metade da classe parcial PartialClass implementando o método PartialMethod, este será chamado no tempo de execução pelo código presente na primeira parte da classe parcial. Caso o convite não seja aceito pelo desenvolvedor, então o código para chamar o método PartialMethod será otimizado pelo compilador, fazendo com que todas as referências do método PartialMethod sejam removidas inteiramente no tempo de execução.

Isto permite que o desenvolvedor trabalhe com uma pequena implementação da classe PartialClass que contém somente alguns métodos, eliminando assim o trabalho com complexas e longas listas de código que podem ser encontradas em código gerado pelas ferramentas do desenvolvedor.

As regras que governam os métodos parciais definem que eles devem começar com a

palavra chave partial, não podem ter modificadores de escopo tais como público, privado ou interno, não podem ser marcados como externos, podem ser marcados como estáticos ou não-seguros, podem ser genéricos, podem ter parâmetros ref, mas nenhum parâmetro out e não podem ser referenciados como delegate.

Métodos parciais são implicitamente marcados como privados. Isto significa que eles não podem ser chamados fora da classe parcial e não podem ser declarados como virtuais.

3.1.11 Expressões lambda

Uma expressão lambda é uma expressão em linha ou um bloco de instruções com uma sintaxe concisa que pode ser usada sempre que um tipo delegate é esperado. Por exemplo, é possível utilizar uma expressão lambda como um argumento para a chamada de um método, para atribuir um valor para um delegate ou para definir um event handler (tratador de evento). Expressões lambda podem ser virtualmente usadas em qualquer lugar onde um método anônimo pode ser usado.

Expressões lambda são usadas extensivamente nas pesquisas como um meio conveniente para criar os delegates que serão invocados, ou as árvores de expressão que serão analisadas, quando a pesquisa for executada posteriormente. Como um desenvolvedor de aplicações escrevendo pesquisas, tipicamente surge a necessidade de escrever expressões lambda diretamente usando a sintaxe de método no momento da chamada de certos métodos. Todas as expressões lambda usam o operador lambda =>, o qual é lido como “vai para”. O lado esquerdo de um operador lambda especifica o parâmetro de entrada (se algum) e o lado direito contém a expressão ou bloco de instruções. A expressão lambda x => x + 1 é lida “x vai para x mais 1.” Esta expressão pode ser atribuída a um tipo delegate.

O Código 14 mostra o uso de expressões lambda:

```
delegate int D(int i);
D myDelegate = x => x + 1;
int j = myDelegate(6); // j = 7
```

Código 14 - Expressões lambda

No código acima, a assinatura do delegate tem um parâmetro de entrada do tipo int, e retorna um int. A expressão lambda pode ser convertida para um delegate deste tipo porque esta também tem um parâmetro de entrada (x) e um valor de retorno que o compilador pode implicitamente converter para o tipo int. Quando um delegate é invocado com um parâmetro de entrada igual a 6, este retorna um resultado igual a 7.

Todas as restrições que se aplicam aos métodos anônimos também se aplicam às expressões lambda.

3.1.12 Inicializadores de objeto

Inicializadores de objeto provêem um meio para atribuir valores para qualquer campo ou propriedade acessível de um objeto no tempo de criação sem a necessidade de explicitamente invocar um construtor.

O Código 15 mostra o uso de inicializadores de objeto:

```
class Foo
{
  public int Integer { get; set; }
  public string String { get; set; }
  public double Double { get; set; }
  public bool Bool { get; set; }
}

Foo foo = new Foo { Integer = 17, String = "Hello World!", Double =
6.101983, Bool = true };
```

Código 15 - Inicializadores de objeto

3.1.13 Inicializadores de coleção

Inicializadores de coleção provêem um meio para especificar um ou mais inicializadores de objeto quando uma classe de coleção que implementa IEnumerable é inicializada.

O Código 16 mostra o uso de inicializadores de coleção:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

List<Foo> list = new List<Foo>
{
  new Foo { Integer = 2003, String = "Hello", Double = 1.0, Bool = false },
  new Foo { Integer = 2007, String = "World!", Double = 2.0, Bool = true }
};
```

Código 16 - Inicializadores de coleção

Usando um inicializador de coleção não há a necessidade de especificar múltiplas chamadas ao método Add da classe no código fonte; o compilador adiciona as chamadas.

O código acima mostra um inicializador de coleção que usa inicializadores de objeto para inicializar objetos da classe Foo. Note que os inicializadores de objeto individuais estão cercados por chaves e separados por vírgulas.

3.1.14 Tipos anônimos

Tipos anônimos provêem um meio conveniente para encapsular um conjunto de propriedades read-only (somente leitura) em um único objeto sem ter que primeiro explicitamente definir um tipo. O nome do tipo é gerado pelo compilador e não está disponível no nível do código fonte. O tipo das propriedades é deduzido pelo compilador.

O Código 17 mostra o uso de tipos anônimos:

```
var v = new { Amount = 777, Message = "Hello World!" };

var query = from d in db.Developers
            select new { d.Name, d.DeveloperLanguages };

foreach(var developer in query)
{
  Console.WriteLine("Name = {0}", developer.Name);

  foreach(var developerLanguage in developer.DeveloperLanguages)
    Console.WriteLine("Language = {0}", developerLanguage.Language.Name);
}
```

Código 17 - Tipos anônimos

Tipos anônimos são tipicamente usados na cláusula select de uma expressão de pesquisa para retornar um subconjunto de propriedades para cada objeto da seqüência fonte.

Tipos anônimos são criados usando o operador new com um inicializador de objeto.

Tipos anônimos são tipos de classe que consistem de uma ou mais propriedades públicas read-only. Nenhum outro tipo de membros de classe tais como métodos ou eventos são permitidos.

O cenário mais comum é inicializar um tipo anônimo com poucas propriedades a partir de um outro tipo.

No código acima, uma classe chamada Developer tem as propriedades Name e DeveloperLanguages juntamente com várias outras propriedades que podem não ter importância no código. A variável db.Developers é uma coleção de objetos do tipo Developer. A declaração do tipo anônimo começa com a keyword new. Esta keyword inicializa um novo tipo que tem somente duas propriedades de Developer. Isto faz com que somente poucos dados sejam retornados da pesquisa.

Se nomes de membros não forem especificados no tipo anônimo, o compilador dá aos membros do tipo anônimo o mesmo nome das propriedades que estão sendo usadas para inicializá-los. Deve-se prover um nome para uma propriedade que está sendo inicializada através de uma expressão.

Quando um tipo anônimo é atribuído a uma variável, esta variável dever ser

inicializada com a construção var. Isto se deve ao fato de que somente o compilador tem acesso ao nome base do tipo anônimo.

Tipos anônimos são tipos de referência que derivam diretamente da classe Object. O compilador dá um nome a eles, porém a aplicação não tem acesso ao nome. Da perspectiva do tempo de execução de linguagem comum (CLR), um tipo anônimo não é diferente de nenhum outro tipo.

Se dois ou mais tipos anônimos possuem o mesmo número e tipo de propriedades na mesma ordem, o compilador os trata como o mesmo tipo e eles compartilham a mesma informação gerada pelo compilador.

Um tipo anônimo possui escopo de método. Para passar um tipo anônimo, ou uma coleção que contém tipos anônimos, para fora do limite de um método, deve-se primeiramente converter o tipo para objeto. De qualquer modo, isto anula a tipificação forte do tipo anônimo. Se houver a necessidade de armazenar os resultados da pesquisa ou passar os resultados para fora do limite do método, é melhor considerar o uso de uma estrutura nomeada padrão ou classe que seja inicializada com um inicializador de objeto, ao invés de um tipo anônimo na cláusula select.

3.1.15 Arrays implicitamente tipificados

Um array implicitamente tipificado pode ser criado de forma que o tipo da instância do array é inferido a partir dos elementos especificados no inicializador do array. As regras para qualquer variável implicitamente tipificada também são aplicadas para arrays implicitamente tipificados.

Arrays implicitamente tipificados são geralmente usados em expressões de pesquisa juntamente com tipos anônimos e inicializadores de objeto e coleção.

O Código 18 mostra o uso de arrays implicitamente tipificados:

```
var a = new[] { "Hello", "World!" }; // string[]

var b = new[] // single-dimension jagged array
{
  new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 },
};

var c = new[] // jagged array of strings
{
  new[] { "First", "Second", "Third" },
};
```

Código 18 - Arrays implicitamente tipificados

É importante notar que nos arrays implicitamente tipificados os colchetes não são

usados do lado esquerdo da instrução de inicialização. É importante notar também que os jagged arrays (arrays de arrays) são inicializados usando new[] da mesma forma que um array de uma dimensão. Arrays implicitamente tipificados multidimensionais não são suportados.

3.1.16 Árvores de expressão

Árvores de expressão representam código do nível da linguagem de programação na forma de dados. Os dados são armazenados em uma estrutura que possui o formato de árvore. Cada nó da árvore de expressão representa uma expressão, por exemplo, uma chamada de um método ou uma operação binária tal como x < y.

O Código 19 mostra o uso de árvores de expressão:

```
// Creates an expression tree
Expression<Func<int, bool>> exprTree = num => num < 6;

// Decomposes the expression tree
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}", param.Name,
left.Name, operation.NodeType, right.Value);
```

Código 19 - Árvores de expressão

O código acima mostra como a árvore de expressão que representa a expressão lambda num => num < 6 pode ser decomposta em partes.

4 DESENVOLVIMENTO

É desenvolvida uma aplicação cliente/servidor, a qual faz uso de um modelo de objetos da linguagem de pesquisa integrada à linguagem de programação. O modelo de objetos é criado a partir de um banco de dados relacional.

4.1 Linguagem de pesquisa integrada à linguagem de programação

Integrar a linguagem de pesquisa à linguagem de programação é viável através de um conjunto de recursos que concede poderosas capacidades de pesquisa dentro da sintaxe das linguagens de programação.

Novos modelos para a manipulação de dados são introduzidos. Tais modelos podem ser estendidos para suportar potencialmente qualquer tipo de fonte de dados.

Existem bibliotecas de código que permitem o uso da linguagem de pesquisa integrada à linguagem de programação com coleções, bancos de dados SQL Server de forma nativa, DataSets e documentos XML.

A Figura 4 ilustra a arquitetura da linguagem de pesquisa integrada à linguagem de programação:

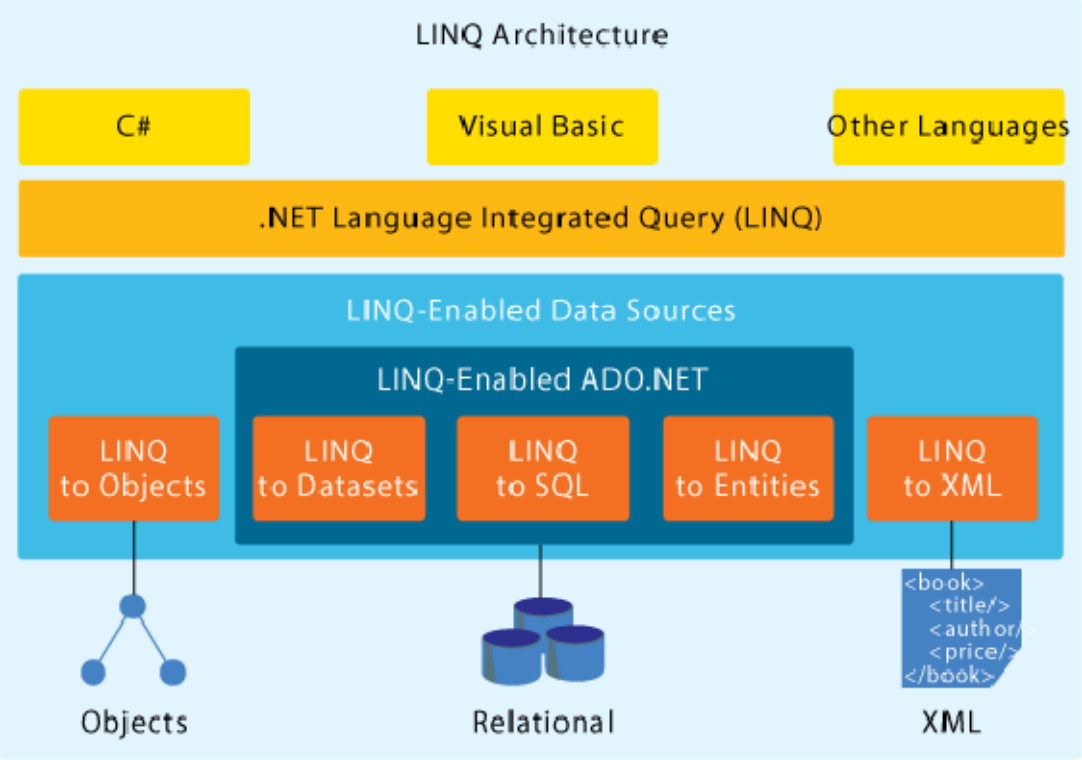

Figura 4 - Arquitetura da linguagem de pesquisa integrada à linguagem de programação

A tecnologia da linguagem de pesquisa integrada à linguagem de programação está presente no .NET Framework em sua versão 3.5. Esta tecnologia promete facilitar a maneira com a qual os desenvolvedores trabalham com dados. É introduzido o conceito de pesquisa como uma construção de primeira classe das linguagens de programação. Pesquisar ou atualizar dados se transforma em uma atividade simples. Com a linguagem de pesquisa integrada à linguagem de programação é possível modelar um banco de dados através de classes e então pesquisar o banco de dados, como também inserir, atualizar ou deletar dados. A linguagem de pesquisa integrada à linguagem de programação suporta transações, views, e stored procedures. Também prove uma maneira fácil de integrar validação dos dados e regras da lógica de negócios.

Ao integrarmos a linguagem de pesquisa à linguagem de programação obtemos as seguintes vantagens:

- Simplificação da maneira de escrever código. Se o desenvolvedor já conhece programação e conceitos de linguagem de pesquisa, tal como SQL, então o mesmo já sabe a maior parte do que é necessário para começar a utilizar a nova tecnologia.
- Unificação da sintaxe para pesquisar qualquer fonte de dados. É possível pesquisar um documento XML quase da mesma maneira que um banco de dados, um DataSet, uma coleção armazenada na memória, ou qualquer outra fonte de dados remota ou local que suporte a tecnologia.
- Reforço na conexão entre dados relacionais e o mundo orientado a objetos. Isto porque a pesquisa se torna uma construção de primeira classe da linguagem de programação. A pesquisa pode ser fortemente tipificada e as ferramentas do desenvolvedor podem ser usadas para criar diagramas relacionais dos objetos.
- Menor tempo de desenvolvimento através da captura de muitos erros no tempo de compilação. Há também suporte ao recurso IntelliSense[c] e à depuração de código.

A linguagem de pesquisa integrada à linguagem de programação representa uma inovação na programação de dados ligando as lacunas entre os dados relacionais e XML de um lado, e linguagens de programação orientadas a objetos do outro.

Historicamente, o mundo dos dados e o mundo dos objetos não se integraram bem. Desenvolvedores trabalham com as linguagens de programação orientadas a objetos e também com as linguagens de pesquisa SQL ou Xquery. De um lado estão conceitos como

[c] IntelliSense é um recurso que permite autocompletar o código. Serve também como documentação e clarificação dos nomes de variáveis e métodos. Usa reflexão de metadados.

classes, objetos, campos, herança, e APIs diversas. Do outro lado estão tabelas, colunas, linhas, nós, e linguagens separadas para lidar com estas estruturas de dados. Tipos de dados geralmente requerem tradução entre os dois mundos; existem diferentes funções padrão. Em virtude do mundo objeto não ter noção de uma pesquisa, uma pesquisa pode somente ser representada como uma string sem checagem de tipo no tempo de compilação ou suporte ao recurso IntelliSense na IDE. Transferir dados de tabelas SQL ou árvores XML para objetos em memória é geralmente tedioso e tende a erros.

Ao integrar a linguagem de pesquisa à linguagem de programação estes problemas são resolvidos, pois um modelo unificado de programação é provido para o trabalho com dados relacionais, arquivos XML, coleções em memória, e qualquer outro tipo de fonte de dados incluindo Active Directory[d] ou até mesmo arquivos .pst do Outlook.

Com a linguagem de pesquisa integrada à linguagem de programação, operações de pesquisa e transformações de dados se tornam conceitos iguais a classes e objetos. Toda a força da SQL ou XQuery está incluída dentro da própria linguagem de programação.

As extensões de linguagem permitem que o código escrito na linguagem de programação use a nova API dos operadores de pesquisa padrão, a qual define uma linguagem de pesquisa racionalmente completa para qualquer coleção que implemente a interface IEnumerable<T>. Isto significa que qualquer coleção ou array pode ser pesquisada através do uso dos operadores de pesquisa padrão; pesquisar objetos na memória deste jeito é chamado linguagem de pesquisa integrada à linguagem de programação para objetos. Provedores de bibliotecas que quiserem habilitar a linguagem de pesquisa integrada à linguagem de programação para suas classes de coleção simplesmente precisam implementar a interface IEnumerable<T> para tais classes. Usando a linguagem de pesquisa integrada à linguagem de programação para SQL, para XML ou para DataSet, o provedor que é implementado por cada uma dessas tecnologias transparentemente converte os dados da fonte para coleções baseadas em IEnumerable. A visão dos dados por parte do desenvolvedor, seja quando pesquisando ou atualizando, é sempre de uma coleção genérica IEnumerable<T> ou IQueryable<T>.

[d] Um Active Directory é um diretório de serviços usado para armazenar informações sobre os recursos da rede pertencentes a um domínio.

4.1.1 Operadores de pesquisa padrão

Os operadores de pesquisa padrão compõem um conjunto de métodos que formam o modelo da linguagem de pesquisa integrada à linguagem de programação. A maioria destes métodos opera em seqüências, onde uma seqüência é um objeto cujo tipo implementa a interface IEnumerable<T> ou a interface IQueryable<T>.

Os operadores de pesquisa padrão provêem habilidades de pesquisa incluindo filtragem, projeção, agregação, classificação, etc.

Existem dois conjuntos de operadores de pesquisa padrão, um que opera em objetos do tipo IEnumerable<T> e um outro que opera em objetos do tipo IQueryable<T>. Os métodos que compõem cada conjunto são membros estáticos das classes Enumerable e Queryable, respectivamente. Os operadores de pesquisa padrão são definidos como métodos de extensão do tipo em que eles operam. Isto significa que eles podem ser chamados usando a sintaxe de um método estático ou a sintaxe de um método de instância.

Os operadores de pesquisa padrão diferem no tempo de suas execuções, dependendo se eles retornam um valor singleton (único) ou uma seqüência de valores. Os métodos que retornam um valor singleton (por exemplo, Average e Sum) executam imediatamente. Os métodos que retornam uma seqüência adiam a execução da pesquisa e retornam um objeto enumerável.

No caso dos métodos que operam em coleções presentes na memória, isto é, os métodos que estendem IEnumerable<T>, o objeto enumerável retornado captura os argumentos que foram passados para o método. Quando um objeto é enumerado, a lógica do operador de pesquisa é empregada e os resultados da pesquisa são retornados. Em contraste, os métodos que estendem IQueryable<T> não implementam nenhum comportamento de pesquisa, mas constroem uma árvore de expressão que representa a pesquisa a ser realizada. O processamento da pesquisa é tratado pelo objeto fonte IQueryable<T>.

A Tabela 3 mostra os operadores de pesquisa padrão e suas respectivas funcionalidades:

Tipo de operador	**Operador**	**Descrição**
Agregação	Aggregate	Aplica uma função sobre uma seqüência.
	Average	Calcula a média de uma seqüência de valores numéricos.
	Count	Conta o número de elementos em uma seqüência com um tipo de retorno int.
	LongCount	Conta o número de elementos em uma seqüência com um tipo de retorno long.

	Max	Busca o máximo valor de uma seqüência de valores numéricos.
	Min	Busca o mínimo valor de uma seqüência de valores numéricos.
	Sum	Calcula a soma de uma seqüência de valores numéricos.
Agrupamento	GroupBy	Agrupa os elementos de uma seqüência.
Concatenação	Concat	Concatena duas seqüências
Conjunto	Distinct	Elimina elementos duplicados de uma seqüência.
	Except	Produz o conjunto diferença entre duas seqüências.
	Intersect	Produz o conjunto interseção de duas seqüências.
	Union	Produz o conjunto união de duas seqüências.
Conversão	AsEnumerable	Retorna a entrada tipificada como IEnumerable<T>.
	AsQueryable	Converte uma IEnumerable genérica para uma IQueryable genérica.
	Cast	Converte os elementos de uma seqüência para um dado tipo.
	OfType	Filtra os elementos de uma seqüência com base em um dado tipo.
	ToArray	Cria um array a partir de uma seqüência.
	ToDictionary	Cria um dicionário a partir de uma seqüência.
	ToList	Cria uma lista a partir de uma seqüência.
	ToLookup	Cria uma busca a partir de uma seqüência.
Elementar	ElementAt	Retorna um elemento localizado em dado índice de uma seqüência.
	ElementAtOrDefault	Retorna um elemento localizado em dado índice de uma seqüência ou um valor padrão se o índice estive fora do limite permitido.
	First	Retorna o primeiro elemento de uma seqüência.
	FirstOrDefault	Retorna o primeiro elemento de uma seqüência ou um valor padrão se nenhum elemento for encontrado.
	Last	Retorna o último elemento de uma seqüência.
	LastOrDefault	Retorna o último elemento de uma seqüência ou um valor padrão se nenhum elemento for encontrado.
	Single	Retorna o único elemento de uma seqüência.
	SingleOrDefault	Retorna o único elemento de uma seqüência ou um valor padrão se nenhum elemento for encontrado.
Geração	DefaultIfEmpty	Gera um elemento padrão para uma seqüência vazia.
	Empty	Retorna uma seqüência vazia.
	Range	Gera uma seqüência de números inteiros.
	Repeat	Gera uma seqüência repetindo um valor um dado número de vezes.
Igualdade	SequenceEqual	Verifica se duas seqüências são iguais.

Junção	GroupJoin	Executa uma junção agrupada de duas seqüências com base em chaves coincidentes extraídas dos elementos.
	Join	Executa uma junção interna de duas seqüências com base em chaves coincidentes extraídas dos elementos.
Ordenação	OrderBy	Ordena uma seqüência de acordo com uma ou mais chaves em ordem crescente.
	OrderByDescending	Ordena uma seqüência de acordo com uma ou mais chaves em ordem decrescente.
	Reverse	Reverte os elementos de uma seqüência.
	ThenBy	Ordena uma sequencia já ordenada de acordo com uma ou mais chaves em ordem crescente.
	ThenByDescending	Ordena uma sequencia já ordenada de acordo com uma ou mais chaves em ordem decrescente.
Particionamento	Skip	Omite um determinado número de elementos de uma seqüência e então retorna o restante da seqüência.
	SkipWhile	Omite os elementos de uma seqüência enquanto um teste for verdadeiro e então retorna o restante da seqüência.
	Take	Retorna um determinado número de elementos de uma seqüência e então omite o restante da seqüência.
	TakeWhile	Retorna elementos de uma seqüência enquanto um teste for verdadeiro e então omite o restante da seqüência.
Quantificação	All	Verifica se todos os elementos de uma seqüência satisfazem uma condição.
	Any	Verifica se algum elemento de uma seqüência satisfaz uma condição.
	Contains	Verifica se uma seqüência contém um dado elemento.
Restrição	Where	Filtra uma seqüência com base em um predicado.
Seleção	Select	Executa uma projeção sobre uma seqüência.
	SelectMany	Executa uma projeção elementar um para muitos sobre uma seqüência.

Tabela 3 - Operadores de pesquisa padrão

Chamadas para métodos de pesquisa podem ser encadeadas em uma pesquisa, o que permite que pesquisas se tornem arbitrariamente complexas.

O Código 20 mostra como os operadores de pesquisa padrão podem ser usados para obter informação sobre uma seqüência:

```
string sentence = "the quick brown fox jumps over the lazy dog";

// Splits the string into individual words to create a collection.
string[] words = sentence.Split(' ');

var query = words.GroupBy(w => w.Length, w => w.ToUpper()).
                  Select(g => new { Length = g.Key, Words = g }).
                  OrderBy(g => g.Length);

foreach(var g in query)
{
  Console.WriteLine("Words of length {0}:", g.Length);

  foreach(string word in g.Words)
    Console.WriteLine(word);
}

/* This code example produces the following output:

   Words of length 3:
   THE
   FOX
   THE
   DOG
   Words of length 4:
   OVER
   LAZY
   Words of length 5:
   QUICK
   BROWN
   JUMPS
*/
```

Código 20 - Operadores de pesquisa padrão

No código acima, três operadores de pesquisa padrão são usados: GroupBy, Select e OrderBy. Note que as chamadas para os operadores de pesquisa padrão são encadeadas na pesquisa.

A variável sentence armazena uma frase que é dividida em palavras, as quais são armazenadas na variável words que é um array de strings. A variável query é implicitamente tipificada, ou seja, seu tipo é deduzido pelo compilador a partir da expressão do lado direito da instrução de inicialização.

O operador de pesquisa padrão GroupBy é aplicado sobre a variável words. GroupBy tem a função de agrupar cada palavra contida em words através do número de caracteres de cada palavra. A expressão lambda w => w.Length cria o delegate que será invocado para selecionar a chave do grupo quando a pesquisa for executada. A outra expressão lambda w => w.ToUpper() cria o delegate que fará a seleção dos elementos pertencentes a cada grupo. Assim, cada é acumulada de acordo com seu comprimento no grupo apropriado.

O operador de pesquisa padrão Select faz uma projeção sobre cada grupo através do uso de uma expressão lambda que cria um tipo anônimo com um inicializador de objeto. O

comprimento de cada palavra (chave de cada grupo g.Key) é atribuído à variável Length e as palavras pertencentes a cada grupo (g) são atribuídas à variável Words.

Por último, o operador de pesquisa padrão OrderBy ordena cada grupo de acordo com o comprimento de suas palavras. Isto é feito através de uma expressão lambda que usa a propriedade Length pertencente ao tipo anônimo criado anteriormente.

4.1.2 Fonte de dados

A primeira coisa a se notar sobre a fonte de dados é que, se esta implicitamente suporta a interface IEnumerable<T>, então a fonte pode ser usada em uma pesquisa. Tipos IEnumerable<T> podem ser enumerados usando a construção foreach, e é assim que a pesquisa é executada. Tipos que suportam IEnumerable<T>, ou uma interface derivada tal como IQueryable<T>, são chamados tipos pesquisáveis.

Se uma fonte de dados é implicitamente um tipo pesquisável, esta não requer nenhuma modificação adicional ou tratamento especial para servir como uma fonte de dados para a linguagem de pesquisa integrada à linguagem de programação. O mesmo é verdade para qualquer tipo de coleção que suporte IEnumerable<T>, tal como as classes genéricas List, Dictionary, e outras da biblioteca de classes do .NET Framework. Se os dados da fonte ainda não estão na memória como um tipo pesquisável, o provedor da linguagem de pesquisa integrada à linguagem de programação deve representá-los como tal. Por exemplo, no Código 21, a linguagem de pesquisa integrada à linguagem de programação para XML carrega um documento XML em um tipo de elemento pesquisável XElement:

```
XElement gradeBook =
XElement.Load(@"C:\LINQ\Project\LINQ\XML\Gradebook.xml");
```

Código 21 - Fonte de dados proveniente de um documento XML

Com a linguagem de pesquisa integrada à linguagem de programação para SQL, primeiramente cria-se um mapeamento entre o modelo relacional do banco de dados e modelo de objetos da linguagem de programação. As pesquisas são escritas com base nos objetos, e no tempo de execução a linguagem de pesquisa integrada à linguagem de programação para SQL trata a comunicação com o banco de dados. No Código 22, Developers representa uma tabela do banco de dados, e Table<Developer> suporta a interface genérica IQueryable<T>.

```
DataContext db = new DataContext(@"C:\LINQ\Project\Data\Startup.mdf");

Table<Developer> Developers = db.GetTable<Developer>();
```

Código 22 - Fonte de dados proveniente de uma tabela do banco de dados

A regra básica é muito simples: uma fonte de dados válida para a linguagem de pesquisa integrada à linguagem de programação é um objeto que suporta a interface genérica IEnumerable<T>, ou uma interface que herda desta.

4.1.3 Operação de pesquisa

Todas as operações de pesquisa consistem de três ações essenciais:

- Obter a fontes de dados;
- Criar a pesquisa;
- Executar a pesquisa.

A Figura 5 mostra as três partes de uma operação de pesquisa. Como pode ser visto, a execução da pesquisa é distinta da pesquisa; em outras palavras, nenhum dado é retornado simplesmente por uma pesquisa ter sido criada.

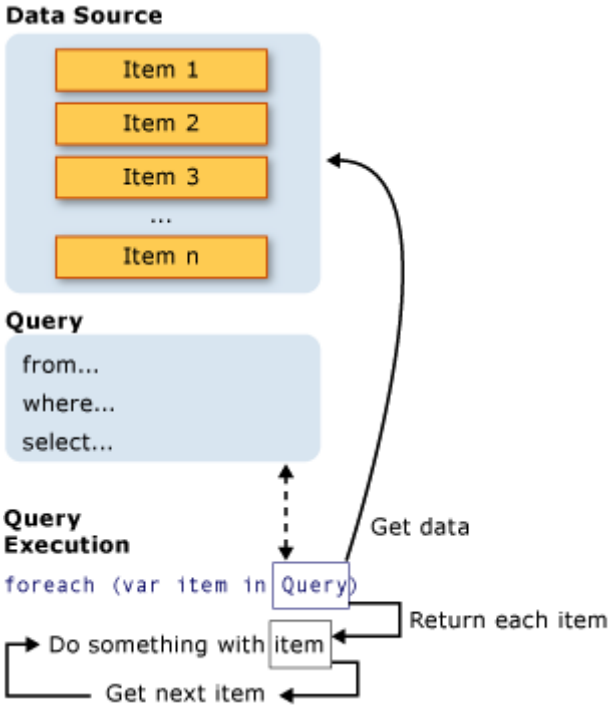

Figura 5 - As três partes de uma operação de pesquisa

O Código 23 mostra como as três partes de uma operação de pesquisa são expressas no código fonte. O exemplo usa um array como a fonte de dados. O mesmo conceito é aplicado para outras fontes dados.

```
// Data source
int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

// Query creation
IEnumerable<int> numQuery = from num in numbers
                            where (num % 2) == 0
                            select num;

// Query execution
foreach(int i in numQuery)
  Console.Write("{0} ", i);
```

Código 23 - As três partes de uma operação de pesquisa

4.1.4 Modelo de objetos

A vertente SQL da linguagem de pesquisa integrada à linguagem de programação, a qual é o foco deste trabalho usa um modelo de objetos expresso na linguagem de programação. Tal modelo é um mapeamento do modelo de dados de um banco de dados relacional. Operações nos dados são então conduzidas em termos do modelo de objetos.

Neste cenário, comandos do banco de dados não são ativados. Ao invés disso, os valores são modificados e métodos são executados dentro do modelo de objetos. Quando se quer pesquisar o banco de dados ou enviar para este as mudanças, a linguagem de pesquisa integrada à linguagem de programação traduz os pedidos para os comandos SQL adequados e envia tais comandos para o banco de dados.

A Figura 6 ilustra a relação entre o desenvolvedor e os dados:

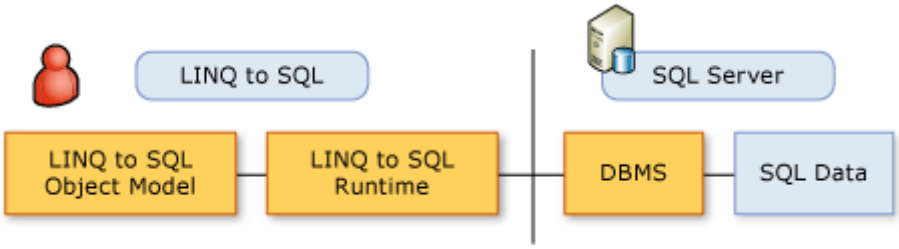

Figura 6 - Relação entre o desenvolvedor e os dados

Os elementos mais importantes do modelo de objetos da linguagem de pesquisa integrada à linguagem de programação e seu relacionamento com os elementos de um modelo de dados relacional estão sumarizados na Tabela 4:

Modelo de objetos	**Modelo de dados relacional**
Classe	Tabela ou view
Membro	Coluna
Relacionamento	Chave estrangeira
Método	Stored procedure ou function

Tabela 4 - Elementos do modelo de objetos versus elementos do modelo relacional

4.2 Estudo de caso

O modelo de objetos pode ser criado com o auxílio do Visual Studio [19]. Neste estudo de caso, o modelo de objetos é adicionado em uma aplicação cliente/servidor.

Um designer torna possível o modelamento das classes, membros, relacionamentos e métodos que mapeiam o esquema do modelo de dados relacional do banco de dados. Uma ferramenta chamada SQLMetal[e] também está disponível para esta tarefa.

A Figura 7 e a Figura 8 mostram respectivamente a criação de um arquivo que representa o modelo de objetos e a seleção dos elementos do banco de dados:

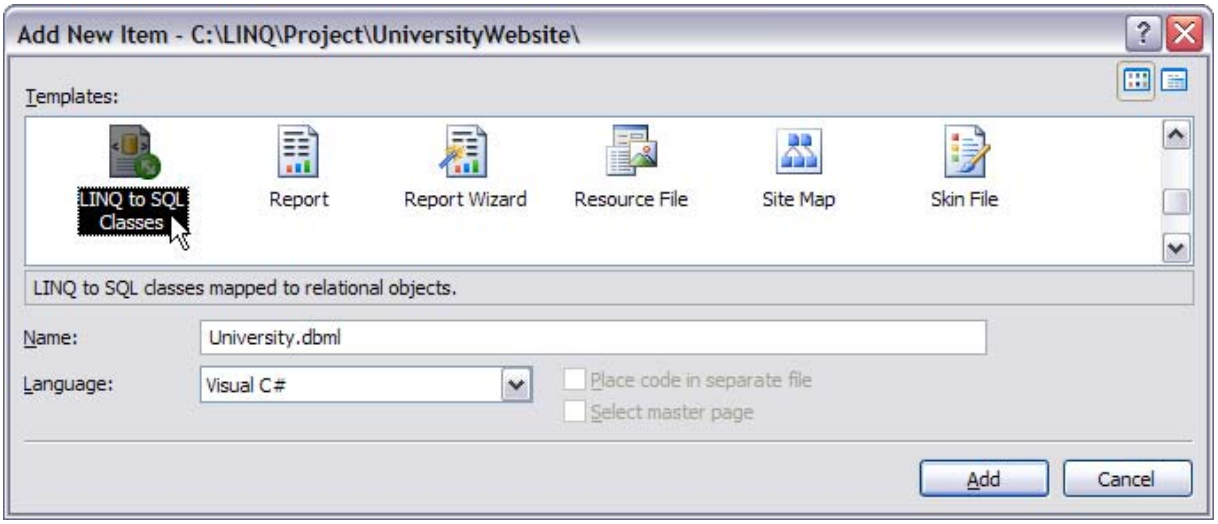

Figura 7 - Criação do modelo de objetos

e A ferramenta de linha de comando SqlMetal gera código que mapeia o modelo de objetos em relação ao banco de dados. Desenvolvedores podem usar esta opção em bancos de dados de grande porte. Haja vista que a ferramenta SQLMetal é usada na linha de comando, esta pode ser incluída no processo de construção da aplicação.

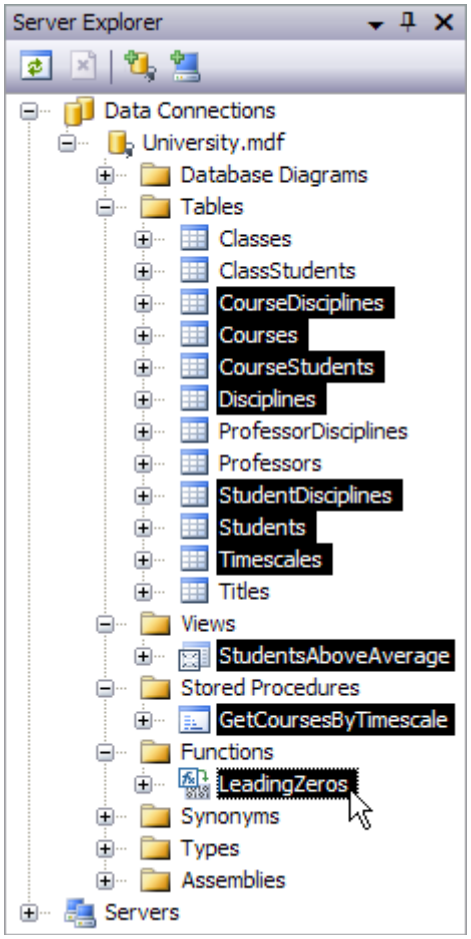

Figura 8 - Esquema do banco de dados

Ao mapear as 7 tabelas, 1 view, 1 stored procedure e 1 function do banco de dados University, são criadas 8 classes e 2 métodos como mostra a Figura 9:

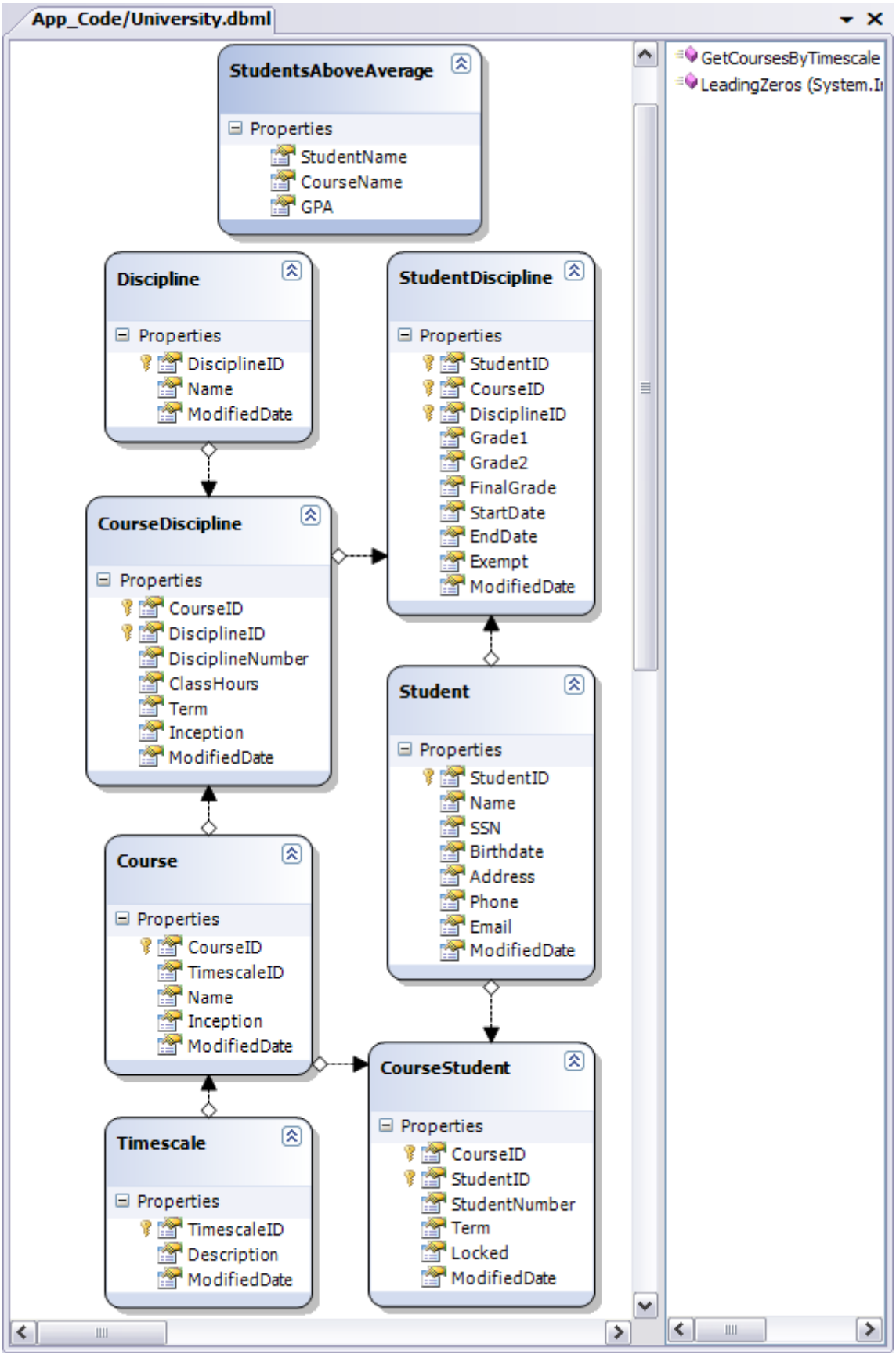

Figura 9 - Mapeamento do modelo relacional para o modelo de objetos

O modelo de objetos é um mapeamento do banco de dados como mostra o Código 24:

```
[Table(Name="dbo.Students")]
public partial class Student {
  [Column(IsPrimaryKey=true)]
  public int StudentID;

...

  private EntitySet<StudentDiscipline> _StudentDisciplines;
  [Association(Storage="_StudentDisciplines", OtherKey="StudentID")]
  public EntitySet<StudentDiscipline> StudentDisciplines {
    get { return this._StudentDisciplines; }
    set { this._StudentDisciplines.Assign(value); }
  }
}

[Table(Name = "dbo.StudentDisciplines")]
public partial class StudentDiscipline {
  [Column(IsPrimaryKey = true)]
    public int StudentID;
  [Column(IsPrimaryKey = true)]
    public int CourseID;
  [Column(IsPrimaryKey = true)]
    public int DisciplineID;

...

  private EntityRef<Student> _Student;
  [Association(Storage = "_Student", ThisKey = "StudentID")]
  public Student Student {
    get { return this._Student.Entity; }
    set { this._Student.Entity = value; }
  }

...

}
```

Código 24 - Mapeamento objeto-relacional

No código acima, o atributo Table mapeia uma classe para uma tabela do banco de dados. O atributo Column mapeia cada campo para uma coluna da tabela. Na tabela Student, StudentID é a chave primária e será usada para estabelecer a identidade do objeto mapeado. Isto é realizado através do parâmetro IsPrimaryKey.

Campos podem ser mapeados para colunas, mas na maioria dos casos propriedades são usadas ao invés de campos. Ao declarar propriedades públicas, é preciso especificar o campo de armazenamento correspondente através do uso do parâmetro Storage do atributo Column.

A tabela Students age como uma tabela tipificada e lógica para as pesquisas. Esta tabela não contém fisicamente todas as linhas da tabela do banco de dados, mas age como um

proxy[f] tipificado para a tabela do banco de dados.

A linguagem de pesquisa integrada à linguagem de programação permite que relacionamentos um para um e um para muitos sejam expressos através do uso dos tipos EntityRef e EntitySet. O atributo Association é usado para mapear um relacionamento. Ao criar o relacionamento entre as tabelas Students e StudentDisciplines, torna-se possível usar a propriedade StudentDiscipline.Student para relacionar diretamente o objeto Student apropriado. Ao especificar o relacionamento declarativamente, evita-se trabalhar com valores de chave estrangeira para associar os objetos correspondentes manualmente. O tipo EntiyRef é usado na classe StudentDiscipline porque existe somente um Student relacionado a um objeto StudentDiscipline.

4.2.1 Classes do modelo de objetos

A linguagem de pesquisa integrada à linguagem de programação permite o modelamento de classes que mapeiam o banco de dados. Tais classes representam as tabelas do banco de dados. As propriedades dessas classes representam as colunas das tabelas. Cada objeto de uma classe representa uma linha da tabela do banco de dados.

As classes definidas na linguagem de pesquisa integrada à linguagem de programação não precisam ser derivadas de uma classe base específica, o que significa que elas podem ser derivadas de qualquer objeto. Todas as classes do modelo de objetos são definidas como classes parciais. Isto significa que é possível criar novas propriedades, métodos, etc. para estas classes.

4.2.2 DataContext

Para cada modelo de objetos, uma classe do tipo DataContext (contexto de dados) é criada. Esta classe é o conduíte principal que permite a pesquisa dos dados provenientes do banco de dados. Esta classe fornece toda a infra-estrutura necessária para aplicar mudanças no banco de dados, pois possui membros que representam cada tabela e view modelada a partir do banco de dados, como também os métodos para cada stored procedure e funções modeladas.

A Figura 10 mostra a classe UniversityDataContext do modelo de objetos criado anteriormente:

[f] Proxy neste caso significa que a tabela tipificada e lógica funciona ou opera como uma representante da tabela do banco de dados.

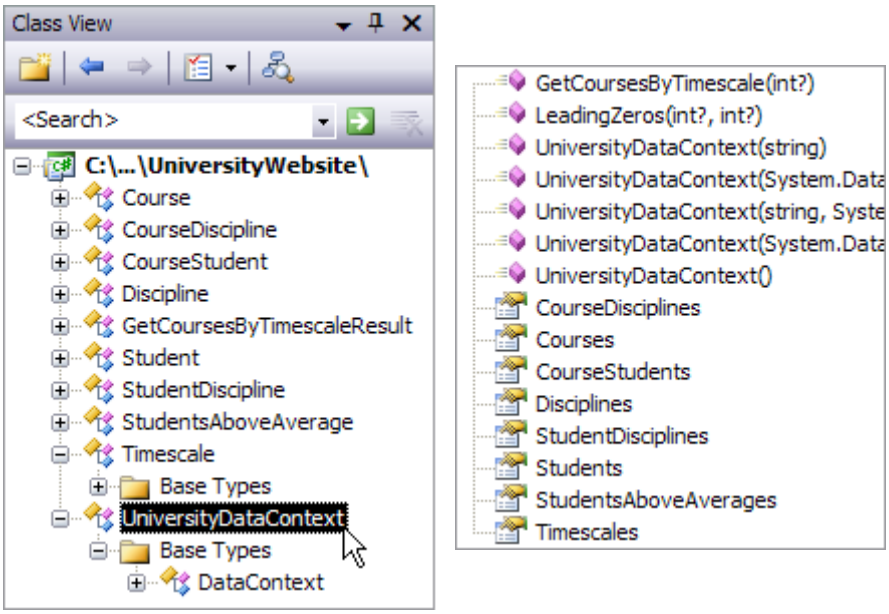

Figura 10 - DataContext

4.2.3 Relacionamentos

Os relacionamentos de chave primária e chave estrangeira das tabelas do banco de dados são inspecionados, e com base neles são criados relacionamentos entre as diferentes classes do modelo de objetos.

As setas entre as classes representam relacionamentos. A direção das setas indica se o relacionamento é do tipo 1:1 (um para um) ou 1:M (um para muitos). Propriedades fortemente tipificadas serão adicionadas nas classes com base nestes relacionamentos.

A Figura 11 mostra o relacionamento entre as tabelas Students e StudentDisciplines onde um relacionamento um-para-muitos pode ser deduzido:

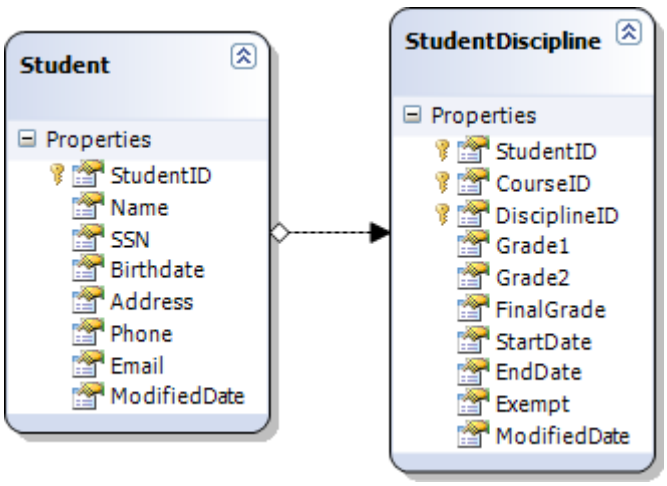

Figura 11 - Relacionamento 1:M (um para muitos)

O relacionamento define que um objeto Student tem uma propriedade

StudentDisciplines que é uma coleção de objetos do tipo StudentDiscipline, a qual pode ser usada para retornar todos os objetos do tipo StudentDiscipline relacionados ao objeto Student. Um objeto StudentDiscipline por sua vez tem uma propriedade Student, a qual pode ser usada para acessar o objeto Student relacionado ao objeto StudentDiscipline.

4.2.4 Pesquisa de dados

Pesquisas podem ser formuladas através de expressões de pesquisa.

Considere por exemplo o Código 25:

```
UniversityDataContext university = new UniversityDataContext();

var studentDisciplineGroups = from sd in university.StudentDisciplines
                              group sd by sd.CourseDiscipline.Course into
cg
                              select new
                              {
                                Course = cg.Key,
                                StudentDisciplines = cg,
                                Students = from sd in cg
                                           group sd by sd.Student into sg
                                           select new { Student = sg.Key,
StudentDisciplines = sg }
                              };

foreach(var c in studentDisciplineGroups)
{
  foreach(var s in c.Students)
  {
    var gpa = s.StudentDisciplines.Average(sd => (sd.Grade1 + sd.Grade2) /
2);

    if(gpa >= 7)
      Response.Write(string.Format("Student: {0} - Course: {1} - GPA: {2}
<br>", s.Student.Name, c.Course.Name, gpa));
  }
}
```

Código 25 - Pesquisa de dados

Note que um objeto do tipo UniversityDataContext é criado para fornecer o interfaceamento com o banco de dados.

Neste exemplo o operador de pesquisa padrão GroupBy é usado para agrupar a coleção de objetos do tipo StudentDiscipline. Primeiro é feito um agrupamento através da propriedade Course e depois para cada Course é feito um agrupamento através da propriedade Student de cada objeto StudentDiscipline relacionado ao objeto Course.

O código itera sobre o resultado da pesquisa e calcula a GPA[g] de cada objeto Student. O resultado da computação é enviado para o browser do usuário.

Há muita flexibilidade em como pesquisar os dados e é possível usufruir dos relacionamentos do modelo de objetos para executar pesquisas mais ricas e naturais.

4.2.5 Operações de insert, update e delete

A linguagem de pesquisa integrada à linguagem de programação automaticamente executará as instruções SQL apropriadas de insert, update e delete quando o método SubmitChanges do DataContext for invocado.

O Código 26 mostra como inserir um novo objeto Course no banco de dados, definindo valores para suas propriedades e então adicionando este na coleção Courses do DataContext.

```
UniversityDataContext university = new UniversityDataContext();

Course course = new Course();

course.Name = "Computer Engineering";
course.TimescaleID = 1;

university.Courses.InsertOnSubmit(course);

university.SubmitChanges();
```

Código 26 - Operação insert

Quando o método SubmitChanges for chamado, uma nova linha será criada na tabela Courses do banco de dados.

O Código 27 mostra como atualizar um objeto Course do banco de dados:

```
UniversityDataContext university = new UniversityDataContext();

Course course = university.Courses.Single(c => c.Name == "Computer
Engineering");

course.TimescaleID = 2;

university.SubmitChanges();
```

Código 27 - Operação update

O Código 28 mostra como deletar mais de um objeto do tipo Course do banco de dados:

[g] GPA é a média geral das notas. É um padrão de avaliação adotado pelos professores para avaliar o rendimento dos estudantes.

```
UniversityDataContext university = new UniversityDataContext();

var courses = from c in university.Courses
              where c.CourseStudents.Count == 0
              select c;

university.Courses.DeleteAllOnSubmit(courses);

university.SubmitChanges();
```

Código 28 - Operação delete

Uma seqüência de cursos que não têm alunos é retornada. Esta seqüência é passada para o método DeleteAllOnSubmit da coleção de Courses do DataContext. Quando o método SubmitChanges for chamado, todas as linhas do banco de dados correspondentes aos objetos do tipo Course serão deletadas.

Como pode ser visto o modelo unificado de programação para efetuar todo este trabalho de manipulação de dados é extremamente limpo e orientado a objetos.

5 CONCLUSÃO

Foram apresentadas e utilizadas as principais extensões que formam a base da linguagem de pesquisa integrada à linguagem programação.

Um modelo de objetos que mapeia o banco de dados relacional também foi criado e suas partes principais foram utilizadas e analisadas.

5.1 Avanços

Com relação à integração da linguagem de pesquisa SQL à linguagem de programação C# visando o trabalho com bancos de dados relacionais, foi constatado que a força deste novo modelo unificado de programação está na separação dos conceitos inerentes ao trabalho do desenvolvedor. Isto porque os detalhes da implementação do banco de dados ficam em segundo plano. O foco está no modelo de objetos que mapeia o banco de dados. Com o auxílio das ferramentas do desenvolvedor, a maior parte do código é gerada automaticamente permitindo o gerenciamento do modelo de objetos de maneira extremamente rápida.

A camada de dados da aplicação é modelada de maneira elegante e limpa. Uma vez definido o modelo de objetos, o trabalho com os dados torna-se fácil e eficiente. Esta qualidade em conjunto com um padrão de arquitetura para aplicações tal como o MVC[a] (Model-View-Controller) [20], permite a criação de um modelo de desenvolvimento estruturado que reforça a separação de conceitos dentro de uma aplicação.

Agora, ao trabalhar com dados provenientes de um banco de dados é possível usar os recursos presentes no ambiente de desenvolvimento como o debugger, IntelliSense e checagem instantânea de sintaxe, os quais enriquecem o trabalho e ajudam o desenvolvedor. A falta destes recursos era uma grande desvantagem no passado porque os desenvolvedores tinham que usar duas linguagens quando trabalhavam com dados: C# e SQL. Além do mais, as pesquisas eram armazenadas em strings, o que dificultava a escrita e manutenção do código, impossibilitando, por conseguinte o uso dos recursos anteriormente mencionados.

A arquitetura do projeto da linguagem de pesquisa integrada à linguagem de

[a] Model-view-controller (MVC) ou Modelo-Visão-Controlador, é um padrão de arquitetura usado na engenharia de software. Em aplicações complexas que apresentam uma grande quantidade de dados para o usuário, um desenvolvedor geralmente deseja separar a lógica dos dados (modelo) da interface do usuário (view), de modo que mudanças efetuadas na interface do usuário não afetem o manuseio dos dados, e que os dados possam ser reorganizados sem mudança na interface do usuário. Este padrão resolve este problema separando o acesso a dados e a lógica de negócios da apresentação dos dados e da interação do usuário, introduzindo para tanto um componente intermediário, o controlador.

programação é altamente modular, o que permite que outros desenvolvedores implementem os recursos de pesquisa em suas bibliotecas de classe. Já existem provedores para outros produtos de bancos de dados tais como MySQL, Oracle e PostgreSQL [21], o que mostra que a tecnologia permite ampla portabilidade e customização.

5.2 Limitações

A linguagem de pesquisa integrada à linguagem de programação possui novas construções de linguagem e como tal demanda muita abstração. É uma excelente tecnologia porque adiciona alguns níveis de complexidade e simultaneamente aumenta a quantidade de trabalho que o ambiente de desenvolvimento recebe de tal modo que são criadas oportunidades fantásticas no quesito performance. Isto é semelhante ao ótimo trabalho que a linguagem SQL executa ao otimizar pesquisas de um banco de dados, porém no atual estágio de desenvolvimento, a linguagem de pesquisa integrada à linguagem de programação mostra-se um pouco mais lenta que pesquisas SQL diretas [22].

Este novo modelo unificado de programação não é adequado para todas as situações. Usá-lo em cenários onde este não se adapta adequadamente, até mesmo com coleções de dados comuns, pode ocasionar desvantagens no quesito performance.

Do ponto de vista do desenvolvedor, a linguagem de pesquisa integrada à linguagem de programação coloca a complexidade da construção de pesquisas dentro do centro da lógica de programação. Neste caso, a lógica da aplicação será quebrada se existir uma mínima modificação na estrutura do esquema do banco de dados. Isto acontece porque até o presente momento, ainda não existe uma maneira automática de atualizar o modelo de objetos quando o esquema do banco de dados é modificado.

5.3 Trabalhos relacionados

Já se passaram algumas décadas desde o início das pesquisas com relação à integração de bancos de dados e linguagens de programação. Atkinson e Buneman [23] realizaram uma análise influente no final da década de 1980 e Paton et al. [24] fizeram uma análise mais atualizada aproximadamente uma década mais tarde. Dada a limitação de espaço não é possível prover uma análise detalhada destas pesquisas, mas esperançosamente estas análises provêem algum contexto histórico para a linguagem de pesquisa integrada à linguagem de programação.

Cook e Ibrahim [25] categorizaram soluções para o problema da combinação mal sucedida de acordo com o tipo de interface: persistência ortogonal ou execução explícita de pesquisa. A linguagem de pesquisa integrada à linguagem de programação se enquadra firmemente na última categoria.

Persistência ortogonal permite que objetos no tempo de execução persistam além da execução de um único programa. Tipicamente objetos persistem até o momento em que podem ser referenciados por algum objeto persistente raiz. A beleza de tais sistemas é que dados persistentes são programados transparentemente, ou seja, são tratados do mesmo modo que dados transientes. Isto torna o código sucinto, mas coloca uma grande carga no tempo de execução da linguagem.

Linguagens que provêem execução explícita de pesquisa permitem que pesquisas sejam escritas em alguma linguagem de pesquisa especializada; a principal vantagem é que desenvolvedores podem interagir diretamente com o banco de dados. Uma maneira simples de prover uma linguagem de pesquisa é usar um pré-processador. SQLJ [26], por exemplo, permite que comandos SQL sejam embutidos diretamente no código Java usando o prefixo #sql. A compilação de SQLJ consiste de dois estágios: primeiro faz-se um pré-processamento do SQL embutido, e depois a compilação pura da linguagem Java.

Uma alternativa, e por sinal uma aproximação dominante, é prover uma API padrão através da qual a linguagem de programação possa acessar o banco de dados. Para o .NET Framework esta pode ser ADO.NET e para Java pode ser JDBC. Infelizmente, estas APIs tipicamente oferecem somente garantias fracas no tempo de compilação, haja vista que a maioria dos comandos do banco de dados são passados como strings.

Várias propostas foram feitas para melhorar a integração de pesquisas de banco de dados com linguagens de programação. Apesar das vantagens óbvias de tais linguagens propostas, parece que suas aceitações foram impedidas pelo fato de que elas são “diferentes” das linguagens de programação mais comumente usadas, tais como Java e C. Por exemplo, HaskellDB [27] propõe extensões para a linguagem funcional Haskell e TL [28] é uma linguagem híbrida funcional/imperativa com tipos e módulos de sistema avançados.

A linguagem de pesquisa integrada à linguagem de programação usa muito da experiência destas pesquisas e transfere estas para uma máquina virtual moderna, comercial, com suporte a linguagens de programação de amplo uso, IDE já consolidado, como também provê uma sofisticada ferramenta de mapeamento entre o mundo relacional e o mundo dos objetos.

5.4 Trabalhos futuros

Construções de pesquisa integradas na linguagem de programação tornam fácil pesquisar qualquer fonte de dados (até sites como Google [29] , Flickr [30] e Amazon [31]), porém um dos problemas que ainda precisa ser resolvido é a “última milha” da programação de dados que é o mapeamento entre os modelos de dados, em particular o mapeamento entre o modelo de dados relacional e o modelo de objetos.

Atualmente, a maioria dos sistemas objeto-relacional (dentre eles o utilizado neste trabalho, o qual pode ser visto no Código 24) suporta somente o mapeamento monolítico entre tabelas, colunas e relacionamentos de um lado e classes, campos e propriedades de navegação baseadas em coleções do outro lado.

Este estilo de mapeamento é não composicional[b] em muitos aspectos, não importando se este usa atributos em linha ou um arquivo XML externo para descrever o mapeamento.

Em virtude dos bancos de dados relacionais relacionarem tabelas via relacionamentos de chave estrangeira, a tabela fonte (Students) não precisa antecipar a priori o relacionamento com a tabela alvo (StudentDisciplines). De fato, é possível definir um número ilimitado de tabelas que formam um relacionamento um-para-um ou um-para-muitos com a tabela Students. Infelizmente, quando tabelas são mapeadas para classes, é preciso decidir de antemão quais relacionamentos são necessários. Estes relacionamentos são explicitamente definidos no tipo fonte, o que quebra a composibilidade.

Em uma futura versão da linguagem de pesquisa integrada à linguagem de programação o suporte aos relacionamentos poderia ser melhorado no nível da linguagem de programação. Neste caso, o que é necessário é uma camada bem fina de mapeamento padrão não programável na margem entre o mundo dos relacionamentos e o mundo dos objetos.

Esta primeira versão da linguagem de programação tratou do problema da combinação mal sucedida. A próxima versão poderia evitar o problema do mapeamento com algo melhor (mapeamento composicional e programável).

Um outro ponto que pode ser tratado é a atualização automática do modelo de objetos quando o esquema do banco de dados é modificado.

---

[b] Composibilidade lida com o projeto de sistemas no tocante aos inter-relacionamentos dos componentes. Um sistema altamente composicional provê a recombinação de componentes que podem ser selecionados e montados em várias combinações para satisfazer os requerimentos do usuário. Os atributos essenciais que formam um componente composicional são: independência (ex.: pode ser distribuído independentemente - note que o componente pode cooperar com outros componentes, mas componentes dependentes são substituíveis.), não possui estado (ex.: trata cada requisição como uma transação independente, sem ralação com requisições anteriores).

Finalmente, a força computacional dos processadores com múltiplos núcleos, a pesquisa avançada em usabilidade e os novos modelos de programação e plataformas prometem mudar a maneira que as pessoas interagem com computadores e dados. Assim, outros trabalhos também podem atacar os problemas ainda abertos que estão relacionados com paralelismo de pesquisas e concorrência[c] de dados.

[c] Concorrência é uma propriedade de sistemas, a qual consiste de computações que executam ao mesmo tempo e pode permitir que recursos sejam compartilhados entre tais computações. O uso concorrente de recursos compartilhados é uma fonte de muitos problemas relacionados ao trabalho com dados.

6 BIBLIOGRAFIA

[1] Meijer, Erik. **LINQ 2.0: Democratizing the Cloud**. My Homepage. 2007. Disponível em <http://research.microsoft.com/~emeijer/Papers/LINQ20.pdf>. Acesso em 20 de Setembro de 2007.

[2] Dave Thomas. **The impedance imperative tuples + objects + infosets = too much stuff!** Wiener, Richard, [ed.]. 5, Zurich : ETH Zurich, Setembro-Outubro de 2003, Journal of Object Technology, Vol. 2, pp. 7–12.

[3] Microsoft Corporation. **Language-Integrated Query (LINQ)**. 2007. Disponível em <http://msdn2.microsoft.com/en-us/library/bb397926(VS.90).aspx>. Acesso em 27 de Agosto de 2007.

[4] Duffy, Joe e Essey, Ed. **Parallel LINQ: Running Queries On Multi-Core Processors**. MSDN Magazine. Outubro de 2007. Disponível em <http://msdn.microsoft.com/msdnmag/issues/07/10/PLINQ/default.aspx>. Acesso em 21 de Outubro de 2007.

[5] Subieta, Kazimierz. **What are query languages?**. 14 de Dezembro de 2005. Disponível em <http://www.sbql.pl/Topics/What%20are%20query%20lang.html>. Acesso em 20 de Novembro de 2007.

[6] International Organization for Standardization. **ISO 5127:2001: Information and documentation -- Vocabulary**. Geneva, Switzerland : s.n., 15 de Julho de 2001.

[7] MacLennan, Bruce J. **Principles of Programming Languages**. New York : Oxford University Press, 1987.

[8] O'Reilly Media Corporation. **The History of Programming Languages**. O'Reilly. 2007. Disponível em <http://www.oreilly.com/news/graphics/prog_lang_poster.pdf>. Acesso em 9 de Novembro de 2007.

[9] Subieta, Kazimierz. **Impedance mismatch**. 26 de Julho de 2007. Disponível em <http://www.sbql.pl/Topics/ImpedanceMismatch.html>. Acesso em 20 de Novembro de 2007.

[10] Meyer, Bertrand. **Object-Oriented Software Construction**. Indianapolis : Prentice Hall PTR, 2000.

[11] Booch, Grady. **Object-Oriented Analysis and Design with Applications**. 2nd ed. Redwood City : Benjamin-Cummings Publishing, 1993.

[12] Armstrong, Deborah J. **The Quarks of Object-Oriented Development**. Communications of the ACM. Fevereiro de 2006, Vol. 49, 2, pp. 123-128.

[13] Microsoft Corporation. **C# 3.0 Language Specification**. MSDN - Microsoft Developer Network. Agosto de 2007. Disponível em <http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc>. Acesso em 23 de Novembro de 2007.

[14] Teorey, Toby J. **Database Modeling and Design**. 3rd ed. Burlington : Morgan Kaufmann, 1999.

[15] Sheldon, Robert. **SQL: A Beginner's Guide**. 2nd ed. Emeryville : McGraw-Hill Osborne Media, 2003.

[16] Wikipedia contributors. **Relational database**. 6 de Dezembro de 2007. Disponível em <http://en.wikipedia.org/w/index.php?title=Relational_database&oldid=176239141>. Acesso em 9 de Dezembro de 2007.

[17] Microsoft Corporation. **.NET Framework Conceptual Overview**. MSDN - Microsoft Developer Network. Disponível em <http://msdn2.microsoft.com/en-us/library/zw4w595w(VS.90).aspx>. Acesso em 20 de Novembro de 2007.

[18] Calvert, Charlie. **Partial Methods**. Charlie Calvert's Community Blog. 11 de Novembro de 2007. Disponível em <http://blogs.msdn.com/charlie/archive/2007/11/11/partial-methods.aspx>. Acesso em 11 de Novembro de 2007.

[19] Microsoft Corporation. **Visual Studio**. Disponível em <http://msdn2.microsoft.com/en-us/vstudio/default.aspx>. Acesso em 20 de Novembro de 2007.

[20] Guthrie, Scott. **ASP.NET MVC Framework (Part 1)**. ScottGu's Blog. 13 de Novembro de 2007. Disponível em <http://weblogs.asp.net/scottgu/archive/2007/11/13/asp-net-mvc-framework-part-1.aspx>. Acesso em 27 de Novembro de 2007.

[21] Moudry, Jiri George. **DbLinq Project: Linq Provider for MySql, Oracle and PostgreSQL**. 2007. Disponível em <http://code2code.net/DB_Linq/>. Acesso em 20 de Novembro de 2007.

[22] Mariani, Rico. **Rico Mariani's Performance Tidbits**. 22 de Junho de 2007. Disponível em <http://blogs.msdn.com/ricom/archive/2007/06/22/dlinq-linq-to-sql-performance-part-1.aspx>. Acesso em 20 de Novembro de 2007.

[23] Malcolm P. Atkinson e O. Peter Buneman. **Types and Persistence in Database Programming Languages.** 2, New York : ACM, 1987, Vol. 19.

[24] Paton, N., et al. **Database Programming Languages**. Hemel Hempstead : Prentice-Hall International, 1996.

[25] W. R. Cook e A. H. Ibrahim. **Programming Languages and Databases: What's the Problem?** s.l. : Unpublished paper, 2005.

[26] Price, Jason. **Java Programming with Oracle SQLJ**. s.l. : O'Reilly, 2001.

[27] Leijen, Daan e Meijer, Erik. **Domain Specific Embedded Compilers**. In Proceedings of Conference on Domain-Specific Languages. 1999.

[28] F. Matthes, S. Mubig e J. W. Schmidt. **Persistent Polymorphic Programming in Tycoon: An introduction.** s.l. : University of Glasgow, 1994.

[29] scobrown. **LINQ Provider for Google**. 6 de Dezembro de 2007. Disponível em <http://www.codeplex.com/glinq>. Acesso em 10 de Dezembro de 2007.

[30] mehfuzh. **LINQ Provider for Flickr**. CodePlex. 16 de Novembro de 2007. Disponível em <http://www.codeplex.com/LINQFlickr>. Acesso em 10 de Dezembro de 2007.

[31] Marguerie, Fabrice. **LINQ Provider for Amazon**. LINQ in Action. Disponível em <http://linqinaction.net/blogs/main/archive/2006/06/26/introducing_linq_to_amazon.aspx>. Acesso em 10 de Dezembro de 2007.

[32] Microsoft Corporation. **Code Generation Tool (SqlMetal.exe)**. MSDN - Microsoft Developer Network. MSDN - Microsoft Developer Network. Disponível em <http://msdn2.microsoft.com/en-us/library/bb386987(VS.90).aspx>. Acesso em 27 de Novembro de 2007.

[33] Marguerie, Fabrice, Eichert, Steve e Wooley, Jim. **LINQ in Action**. Greenwich : Manning Publications, 2007.

[34] Microsoft Corporation. **Microsoft C# Language Specifications**. 1st ed. Redmond : Microsoft Press, 2001.

[35] Bierman, Gavin, Meijer, Erik e Torgersen, Mads. **Lost In Translation: Formalizing Proposed Extensions to C#**. My Homepage. 2007. Disponível em <http://research.microsoft.com/~emeijer/Papers/oopslafp017-bierman.pdf>. Acesso em 29 de Novembro de 2007.

[36] Meijer, Erik. **Confessions of a Used Programming Language Salesman**. 2007. Disponível em <http://research.microsoft.com/~emeijer/Papers/es012-meijer.pdf>. Acesso em 30 de Novembro de 2007.

[37] Horton, Anson. **The Evolution Of LINQ And Its Impact On The Design Of C#**. MSDN Magazine - The Microsoft Journal for Developers. Junho de 2007. Disponível em <http://msdn.microsoft.com/msdnmag/issues/07/06/CSharp30/default.aspx>. Acesso em 3 de Dezembro de 2007.

ANEXOS

Blocos de código da aplicação cliente/servidor.

Arquivo XML responsável pelo mapeamento das tabelas do banco de dados

```
<?xml version="1.0" encoding="utf-16"?>
<Database Name="C:\LINQ\PROJECT\UNIVERSITYWEBSITE\APP_DATA\UNIVERSITY.MDF"
Class="UniversityDataContext" Serialization="Unidirectional"
xmlns="http://schemas.microsoft.com/linqtosql/dbml/2007">
  <Connection Mode="WebSettings" ConnectionString="Data
Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\University.mdf;Integra
ted Security=True;Connect Timeout=30;User Instance=False"
SettingsObjectName="System.Configuration.ConfigurationManager.ConnectionStr
ings" SettingsPropertyName="UniversityConnectionString"
Provider="System.Data.SqlClient" />
  <Table Name="dbo.Disciplines" Member="Disciplines">
    <Type Name="Discipline">
      <Column Name="DisciplineID" Type="System.Int32" DbType="Int NOT NULL
IDENTITY" IsPrimaryKey="true" IsDbGenerated="true" CanBeNull="false" />
      <Column Name="Name" Type="System.String" DbType="NVarChar(64) NOT
NULL" CanBeNull="false" />
      <Column Name="ModifiedDate" Type="System.DateTime" DbType="DateTime
NOT NULL" CanBeNull="false" />
      <Association Name="Discipline_CourseDiscipline"
Member="CourseDisciplines" OtherKey="DisciplineID" Type="CourseDiscipline"
/>
    </Type>
  </Table>
  <Table Name="dbo.CourseDisciplines" Member="CourseDisciplines">
    <Type Name="CourseDiscipline">
      <Column Name="CourseID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="DisciplineID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="DisciplineNumber" Type="System.String"
DbType="VarChar(8) NOT NULL" CanBeNull="false" />
      <Column Name="ClassHours" Type="System.Int32" DbType="Int NOT NULL"
CanBeNull="false" />
      <Column Name="Term" Type="System.Int32" DbType="Int NOT NULL"
CanBeNull="false" />
      <Column Name="Inception" Type="System.DateTime" DbType="DateTime NOT
NULL" CanBeNull="false" />
      <Column Name="ModifiedDate" Type="System.DateTime" DbType="DateTime
NOT NULL" CanBeNull="false" />
      <Association Name="CourseDiscipline_StudentDiscipline"
Member="StudentDisciplines" OtherKey="CourseID,DisciplineID"
Type="StudentDiscipline" />
      <Association Name="Discipline_CourseDiscipline" Member="Discipline"
ThisKey="DisciplineID" Type="Discipline" IsForeignKey="true" />
      <Association Name="Course_CourseDiscipline" Member="Course"
ThisKey="CourseID" Type="Course" IsForeignKey="true" />
    </Type>
  </Table>
  <Table Name="dbo.Timescales" Member="Timescales">
    <Type Name="Timescale">
      <Column Name="TimescaleID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
```

```
      <Column Name="Description" Type="System.String" DbType="NVarChar(16)
NOT NULL" CanBeNull="false" />
      <Column Name="ModifiedDate" Type="System.DateTime" DbType="DateTime
NOT NULL" CanBeNull="false" />
      <Association Name="Timescale_Course" Member="Courses"
OtherKey="TimescaleID" Type="Course" />
    </Type>
  </Table>
  <Table Name="dbo.Courses" Member="Courses">
    <Type Name="Course">
      <Column Name="CourseID" Type="System.Int32" DbType="Int NOT NULL
IDENTITY" IsPrimaryKey="true" IsDbGenerated="true" CanBeNull="false" />
      <Column Name="TimescaleID" Type="System.Int32" DbType="Int NOT NULL"
CanBeNull="false" />
      <Column Name="Name" Type="System.String" DbType="NVarChar(32) NOT
NULL" CanBeNull="false" />
      <Column Name="Inception" Type="System.DateTime" DbType="DateTime NOT
NULL" CanBeNull="false" />
      <Column Name="ModifiedDate" Type="System.DateTime" DbType="DateTime
NOT NULL" CanBeNull="false" />
      <Association Name="Course_CourseDiscipline"
Member="CourseDisciplines" OtherKey="CourseID" Type="CourseDiscipline" />
      <Association Name="Course_CourseStudent" Member="CourseStudents"
OtherKey="CourseID" Type="CourseStudent" />
      <Association Name="Timescale_Course" Member="Timescale"
ThisKey="TimescaleID" Type="Timescale" IsForeignKey="true" />
    </Type>
  </Table>
  <Table Name="dbo.CourseStudents" Member="CourseStudents">
    <Type Name="CourseStudent">
      <Column Name="CourseID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="StudentID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="StudentNumber" Type="System.String" DbType="VarChar(8)
NOT NULL" CanBeNull="false" />
      <Column Name="Term" Type="System.Int32" DbType="Int NOT NULL"
CanBeNull="false" />
      <Column Name="Locked" Type="System.Boolean" DbType="Bit NOT NULL"
CanBeNull="false" />
      <Column Name="ModifiedDate" Type="System.DateTime" DbType="DateTime
NOT NULL" CanBeNull="false" />
      <Association Name="Course_CourseStudent" Member="Course"
ThisKey="CourseID" Type="Course" IsForeignKey="true" />
      <Association Name="Student_CourseStudent" Member="Student"
ThisKey="StudentID" Type="Student" IsForeignKey="true" />
    </Type>
  </Table>
  <Table Name="dbo.StudentDisciplines" Member="StudentDisciplines">
    <Type Name="StudentDiscipline">
      <Column Name="StudentID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="CourseID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="DisciplineID" Type="System.Int32" DbType="Int NOT NULL"
IsPrimaryKey="true" CanBeNull="false" />
      <Column Name="Grade1" Type="System.Decimal" DbType="Decimal(3,1)"
CanBeNull="true" />
      <Column Name="Grade2" Type="System.Decimal" DbType="Decimal(3,1)"
CanBeNull="true" />
```

```
      <Column Name="FinalGrade" Type="System.Decimal" DbType="Decimal(3,1)"
CanBeNull="true" />
      <Column Name="StartDate" Type="System.DateTime" DbType="DateTime NOT
NULL" CanBeNull="false" />
      <Column Name="EndDate" Type="System.DateTime" DbType="DateTime"
CanBeNull="true" />
      <Column Name="Exempt" Type="System.Boolean" DbType="Bit NOT NULL"
CanBeNull="false" />
      <Column Name="ModifiedDate" Type="System.DateTime" DbType="DateTime
NOT NULL" CanBeNull="false" />
      <Association Name="CourseDiscipline_StudentDiscipline"
Member="CourseDiscipline" ThisKey="CourseID,DisciplineID"
Type="CourseDiscipline" IsForeignKey="true" />
      <Association Name="Student_StudentDiscipline" Member="Student"
ThisKey="StudentID" Type="Student" IsForeignKey="true" />
    </Type>
  </Table>
  <Table Name="dbo.Students" Member="Students">
    <Type Name="Student">
      <Column Name="StudentID" Type="System.Int32" DbType="Int NOT NULL
IDENTITY" IsPrimaryKey="true" IsDbGenerated="true" CanBeNull="false" />
      <Column Name="Name" Type="System.String" DbType="VarChar(64) NOT
NULL" CanBeNull="false" />
      <Column Name="SSN" Type="System.String" DbType="VarChar(16) NOT NULL"
CanBeNull="false" />
      <Column Name="Birthdate" Type="System.DateTime" DbType="DateTime NOT
NULL" CanBeNull="false" />
      <Column Name="Address" Type="System.String" DbType="VarChar(128) NOT
NULL" CanBeNull="false" />
      <Column Name="Phone" Type="System.String" DbType="VarChar(16) NOT
NULL" CanBeNull="false" />
      <Column Name="Email" Type="System.String" DbType="VarChar(32)"
CanBeNull="true" />
      <Column Name="ModifiedDate" Type="System.DateTime" DbType="DateTime
NOT NULL" CanBeNull="false" />
      <Association Name="Student_CourseStudent" Member="CourseStudents"
OtherKey="StudentID" Type="CourseStudent" />
      <Association Name="Student_StudentDiscipline"
Member="StudentDisciplines" OtherKey="StudentID" Type="StudentDiscipline"
/>
    </Type>
  </Table>
  <Table Name="dbo.StudentsAboveAverage" Member="StudentsAboveAverages">
    <Type Name="StudentsAboveAverage">
      <Column Name="StudentName" Type="System.String" DbType="VarChar(64)
NOT NULL" CanBeNull="false" />
      <Column Name="CourseName" Type="System.String" DbType="NVarChar(32)
NOT NULL" CanBeNull="false" />
      <Column Name="GPA" Type="System.Decimal" DbType="Decimal(3,1)"
CanBeNull="true" />
    </Type>
  </Table>
  <Function Name="dbo.GetCoursesByTimescale"
Method="GetCoursesByTimescale">
    <Parameter Name="timescaleID" Type="System.Int32" DbType="Int" />
    <ElementType Name="GetCoursesByTimescaleResult">
      <Column Name="CourseID" Type="System.Int32" DbType="Int NOT NULL"
CanBeNull="false" />
      <Column Name="Name" Type="System.String" DbType="NVarChar(32) NOT
NULL" CanBeNull="false" />
```

```
      <Column Name="Inception" Type="System.DateTime" DbType="DateTime NOT
NULL" CanBeNull="false" />
    </ElementType>
  </Function>
  <Function Name="dbo.LeadingZeros" Method="LeadingZeros"
IsComposable="true">
    <Parameter Name="value1" Type="System.Int32" DbType="Int" />
    <Parameter Name="value2" Type="System.Int32" DbType="Int" />
    <Return Type="System.String" />
  </Function>
</Database>
```

Classe UniversityDataContext

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="C:\\LINQ\\PROJECT\\UNIVERSITYWEBSITE\\APP_DATA\\UNIVERSITY.MDF")]
public partial class UniversityDataContext : System.Data.Linq.DataContext
{

  private static System.Data.Linq.Mapping.MappingSource mappingSource = new AttributeMappingSource();

  #region Extensibility Method Definitions
  partial void OnCreated();
  partial void InsertDiscipline(Discipline instance);
  partial void UpdateDiscipline(Discipline instance);
  partial void DeleteDiscipline(Discipline instance);
  partial void InsertCourseDiscipline(CourseDiscipline instance);
  partial void UpdateCourseDiscipline(CourseDiscipline instance);
  partial void DeleteCourseDiscipline(CourseDiscipline instance);
  partial void InsertTimescale(Timescale instance);
  partial void UpdateTimescale(Timescale instance);
  partial void DeleteTimescale(Timescale instance);
  partial void InsertCourse(Course instance);
  partial void UpdateCourse(Course instance);
  partial void DeleteCourse(Course instance);
  partial void InsertCourseStudent(CourseStudent instance);
  partial void UpdateCourseStudent(CourseStudent instance);
  partial void DeleteCourseStudent(CourseStudent instance);
  partial void InsertStudentDiscipline(StudentDiscipline instance);
  partial void UpdateStudentDiscipline(StudentDiscipline instance);
  partial void DeleteStudentDiscipline(StudentDiscipline instance);
  partial void InsertStudent(Student instance);
  partial void UpdateStudent(Student instance);
  partial void DeleteStudent(Student instance);
  #endregion

  static UniversityDataContext()
  {
  }

  public UniversityDataContext(string connection) :
      base(connection, mappingSource)
  {
    OnCreated();
  }

  public UniversityDataContext(System.Data.IDbConnection connection) :
      base(connection, mappingSource)
  {
    OnCreated();
  }

  public UniversityDataContext(string connection,
System.Data.Linq.Mapping.MappingSource mappingSource) :
      base(connection, mappingSource)
  {
    OnCreated();
  }

  public UniversityDataContext(System.Data.IDbConnection connection,
System.Data.Linq.Mapping.MappingSource mappingSource) :
      base(connection, mappingSource)
```

```
    {
      OnCreated();
    }

    public UniversityDataContext() :

base(global::System.Configuration.ConfigurationManager.ConnectionStrings["U
niversityConnectionString"].ConnectionString, mappingSource)
    {
      OnCreated();
    }

    public System.Data.Linq.Table<Discipline> Disciplines
    {
      get
      {
        return this.GetTable<Discipline>();
      }
    }

    public System.Data.Linq.Table<CourseDiscipline> CourseDisciplines
    {
      get
      {
        return this.GetTable<CourseDiscipline>();
      }
    }

    public System.Data.Linq.Table<Timescale> Timescales
    {
      get
      {
        return this.GetTable<Timescale>();
      }
    }

    public System.Data.Linq.Table<Course> Courses
    {
      get
      {
        return this.GetTable<Course>();
      }
    }

    public System.Data.Linq.Table<CourseStudent> CourseStudents
    {
      get
      {
        return this.GetTable<CourseStudent>();
      }
    }

    public System.Data.Linq.Table<StudentDiscipline> StudentDisciplines
    {
      get
      {
        return this.GetTable<StudentDiscipline>();
      }
    }

    public System.Data.Linq.Table<Student> Students
```

```
  {
    get
    {
      return this.GetTable<Student>();
    }
  }

  public System.Data.Linq.Table<StudentsAboveAverage> StudentsAboveAverages
  {
    get
    {
      return this.GetTable<StudentsAboveAverage>();
    }
  }

  [Function(Name="dbo.GetCoursesByTimescale")]
  public ISingleResult<GetCoursesByTimescaleResult>
GetCoursesByTimescale([Parameter(DbType="Int")] System.Nullable<int>
timescaleID)
  {
    IExecuteResult result = this.ExecuteMethodCall(this,
((MethodInfo)(MethodInfo.GetCurrentMethod())), timescaleID);
    return
((ISingleResult<GetCoursesByTimescaleResult>)(result.ReturnValue));
  }

  [Function(Name="dbo.LeadingZeros", IsComposable=true)]
  public string LeadingZeros([Parameter(DbType="Int")] System.Nullable<int>
value1, [Parameter(DbType="Int")] System.Nullable<int> value2)
  {
    return ((string)(this.ExecuteMethodCall(this,
((MethodInfo)(MethodInfo.GetCurrentMethod())), value1,
value2).ReturnValue));
  }
}
```

Mapeamento da tabela StudentDisciplines

```
[Table(Name="dbo.StudentDisciplines")]
[DataContract()]
public partial class StudentDiscipline : INotifyPropertyChanging,
INotifyPropertyChanged
{

  private static PropertyChangingEventArgs emptyChangingEventArgs = new
PropertyChangingEventArgs(String.Empty);

  private int _StudentID;

  private int _CourseID;

  private int _DisciplineID;

  private System.Nullable<decimal> _Grade1;

  private System.Nullable<decimal> _Grade2;

  private System.Nullable<decimal> _FinalGrade;

  private System.DateTime _StartDate;

  private System.Nullable<System.DateTime> _EndDate;

  private bool _Exempt;

  private System.DateTime _ModifiedDate;

  private EntityRef<CourseDiscipline> _CourseDiscipline;

  private EntityRef<Student> _Student;

  #region Extensibility Method Definitions
  partial void OnLoaded();
  partial void OnValidate();
  partial void OnCreated();
  partial void OnStudentIDChanging(int value);
  partial void OnStudentIDChanged();
  partial void OnCourseIDChanging(int value);
  partial void OnCourseIDChanged();
  partial void OnDisciplineIDChanging(int value);
  partial void OnDisciplineIDChanged();
  partial void OnGrade1Changing(System.Nullable<decimal> value);
  partial void OnGrade1Changed();
  partial void OnGrade2Changing(System.Nullable<decimal> value);
  partial void OnGrade2Changed();
  partial void OnFinalGradeChanging(System.Nullable<decimal> value);
  partial void OnFinalGradeChanged();
  partial void OnStartDateChanging(System.DateTime value);
  partial void OnStartDateChanged();
  partial void OnEndDateChanging(System.Nullable<System.DateTime> value);
  partial void OnEndDateChanged();
  partial void OnExemptChanging(bool value);
  partial void OnExemptChanged();
  partial void OnModifiedDateChanging(System.DateTime value);
  partial void OnModifiedDateChanged();
  #endregion
```

```
  public StudentDiscipline()
  {
    this.Initialize();
  }

  [Column(Storage="_StudentID", DbType="Int NOT NULL", IsPrimaryKey=true)]
  [DataMember(Order=1)]
  public int StudentID
  {
    get
    {
      return this._StudentID;
    }
    set
    {
      if ((this._StudentID != value))
      {
        if (this._Student.HasLoadedOrAssignedValue)
        {
          throw new
System.Data.Linq.ForeignKeyReferenceAlreadyHasValueException();
        }
        this.OnStudentIDChanging(value);
        this.SendPropertyChanging();
        this._StudentID = value;
        this.SendPropertyChanged("StudentID");
        this.OnStudentIDChanged();
      }
    }
  }

  [Column(Storage="_CourseID", DbType="Int NOT NULL", IsPrimaryKey=true)]
  [DataMember(Order=2)]
  public int CourseID
  {
    get
    {
      return this._CourseID;
    }
    set
    {
      if ((this._CourseID != value))
      {
        if (this._CourseDiscipline.HasLoadedOrAssignedValue)
        {
          throw new
System.Data.Linq.ForeignKeyReferenceAlreadyHasValueException();
        }
        this.OnCourseIDChanging(value);
        this.SendPropertyChanging();
        this._CourseID = value;
        this.SendPropertyChanged("CourseID");
        this.OnCourseIDChanged();
      }
    }
  }

  [Column(Storage="_DisciplineID", DbType="Int NOT NULL",
IsPrimaryKey=true)]
  [DataMember(Order=3)]
  public int DisciplineID
```

```
    {
      get
      {
        return this._DisciplineID;
      }
      set
      {
        if ((this._DisciplineID != value))
        {
          if (this._CourseDiscipline.HasLoadedOrAssignedValue)
          {
            throw new
System.Data.Linq.ForeignKeyReferenceAlreadyHasValueException();
          }
          this.OnDisciplineIDChanging(value);
          this.SendPropertyChanging();
          this._DisciplineID = value;
          this.SendPropertyChanged("DisciplineID");
          this.OnDisciplineIDChanged();
        }
      }
    }

    [Column(Storage="_Grade1", DbType="Decimal(3,1)")]
    [DataMember(Order=4)]
    public System.Nullable<decimal> Grade1
    {
      get
      {
        return this._Grade1;
      }
      set
      {
        if ((this._Grade1 != value))
        {
          this.OnGrade1Changing(value);
          this.SendPropertyChanging();
          this._Grade1 = value;
          this.SendPropertyChanged("Grade1");
          this.OnGrade1Changed();
        }
      }
    }

    [Column(Storage="_Grade2", DbType="Decimal(3,1)")]
    [DataMember(Order=5)]
    public System.Nullable<decimal> Grade2
    {
      get
      {
        return this._Grade2;
      }
      set
      {
        if ((this._Grade2 != value))
        {
          this.OnGrade2Changing(value);
          this.SendPropertyChanging();
          this._Grade2 = value;
          this.SendPropertyChanged("Grade2");
          this.OnGrade2Changed();
```

```
    }
  }
}

[Column(Storage="_FinalGrade", DbType="Decimal(3,1)")]
[DataMember(Order=6)]
public System.Nullable<decimal> FinalGrade
{
  get
  {
    return this._FinalGrade;
  }
  set
  {
    if ((this._FinalGrade != value))
    {
      this.OnFinalGradeChanging(value);
      this.SendPropertyChanging();
      this._FinalGrade = value;
      this.SendPropertyChanged("FinalGrade");
      this.OnFinalGradeChanged();
    }
  }
}

[Column(Storage="_StartDate", DbType="DateTime NOT NULL")]
[DataMember(Order=7)]
public System.DateTime StartDate
{
  get
  {
    return this._StartDate;
  }
  set
  {
    if ((this._StartDate != value))
    {
      this.OnStartDateChanging(value);
      this.SendPropertyChanging();
      this._StartDate = value;
      this.SendPropertyChanged("StartDate");
      this.OnStartDateChanged();
    }
  }
}

[Column(Storage="_EndDate", DbType="DateTime")]
[DataMember(Order=8)]
public System.Nullable<System.DateTime> EndDate
{
  get
  {
    return this._EndDate;
  }
  set
  {
    if ((this._EndDate != value))
    {
      this.OnEndDateChanging(value);
      this.SendPropertyChanging();
      this._EndDate = value;
```

```
            this.SendPropertyChanged("EndDate");
            this.OnEndDateChanged();
        }
      }
    }

    [Column(Storage="_Exempt", DbType="Bit NOT NULL")]
    [DataMember(Order=9)]
    public bool Exempt
    {
      get
      {
        return this._Exempt;
      }
      set
      {
        if ((this._Exempt != value))
        {
          this.OnExemptChanging(value);
          this.SendPropertyChanging();
          this._Exempt = value;
          this.SendPropertyChanged("Exempt");
          this.OnExemptChanged();
        }
      }
    }

    [Column(Storage="_ModifiedDate", DbType="DateTime NOT NULL")]
    [DataMember(Order=10)]
    public System.DateTime ModifiedDate
    {
      get
      {
        return this._ModifiedDate;
      }
      set
      {
        if ((this._ModifiedDate != value))
        {
          this.OnModifiedDateChanging(value);
          this.SendPropertyChanging();
          this._ModifiedDate = value;
          this.SendPropertyChanged("ModifiedDate");
          this.OnModifiedDateChanged();
        }
      }
    }

    [Association(Name="CourseDiscipline_StudentDiscipline",
  Storage="_CourseDiscipline", ThisKey="CourseID,DisciplineID",
  IsForeignKey=true)]
    public CourseDiscipline CourseDiscipline
    {
      get
      {
        return this._CourseDiscipline.Entity;
      }
      set
      {
        CourseDiscipline previousValue = this._CourseDiscipline.Entity;
        if (((previousValue != value)
```

```
            || (this._CourseDiscipline.HasLoadedOrAssignedValue == false)))
      {
        this.SendPropertyChanging();
        if ((previousValue != null))
        {
          this._CourseDiscipline.Entity = null;
          previousValue.StudentDisciplines.Remove(this);
        }
        this._CourseDiscipline.Entity = value;
        if ((value != null))
        {
          value.StudentDisciplines.Add(this);
          this._CourseID = value.CourseID;
          this._DisciplineID = value.DisciplineID;
        }
        else
        {
          this._CourseID = default(int);
          this._DisciplineID = default(int);
        }
        this.SendPropertyChanged("CourseDiscipline");
      }
    }
  }

  [Association(Name="Student_StudentDiscipline", Storage="_Student",
ThisKey="StudentID", IsForeignKey=true)]
  public Student Student
  {
    get
    {
      return this._Student.Entity;
    }
    set
    {
      Student previousValue = this._Student.Entity;
      if (((previousValue != value)
            || (this._Student.HasLoadedOrAssignedValue == false)))
      {
        this.SendPropertyChanging();
        if ((previousValue != null))
        {
          this._Student.Entity = null;
          previousValue.StudentDisciplines.Remove(this);
        }
        this._Student.Entity = value;
        if ((value != null))
        {
          value.StudentDisciplines.Add(this);
          this._StudentID = value.StudentID;
        }
        else
        {
          this._StudentID = default(int);
        }
        this.SendPropertyChanged("Student");
      }
    }
  }

  public event PropertyChangingEventHandler PropertyChanging;
```

```
  public event PropertyChangedEventHandler PropertyChanged;

  protected virtual void SendPropertyChanging()
  {
    if ((this.PropertyChanging != null))
    {
      this.PropertyChanging(this, emptyChangingEventArgs);
    }
  }

  protected virtual void SendPropertyChanged(String propertyName)
  {
    if ((this.PropertyChanged != null))
    {
      this.PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
    }
  }

  private void Initialize()
  {
    OnCreated();
    this._CourseDiscipline = default(EntityRef<CourseDiscipline>);
    this._Student = default(EntityRef<Student>);
  }

  [OnDeserializing()]
  private void OnDeserializing(StreamingContext context)
  {
    this.Initialize();
  }
}
```

Exemplo de relacionamento M:M (muitos para muitos)

```
public partial class Student
{
  protected EntitySet<Discipline> disciplines;

  /// <summary>
  /// Returns assigned disciplines to this student.
  /// </summary>
  public EntitySet<Discipline> Disciplines
  {
    get
    {
      disciplines = new EntitySet<Discipline>();

      using(UniversityDataContext db = new UniversityDataContext())
      {
        var query = from d in db.Disciplines
                    join studentDiscipline in db.StudentDisciplines
                    on d.DisciplineID equals studentDiscipline.DisciplineID
                    where studentDiscipline.StudentID == this.StudentID
                    select d;

        foreach(var discipline in query)
          disciplines.Add(discipline);

        return disciplines;
      }
    }
  }
}
```