FACULDADE DE TECNOLOGIA DE SÃO PAULO

LINQ APLICADO A XML

São Paulo
2011

FACULDADE DE TECNOLOGIA DE SÃO PAULO

LINQ APLICADO A XML

Bruno Marangoni dos Santos Leite

Monografia submetida como exigência parcial para obtenção do grau de Tecnólogo em Processamento de Dados, sob orientação do **Professor Irineu Francisco de Aguiar**

São Paulo
2011

“... e a ciência se multiplicará.”

Daniel 12:4

Agradecimentos

Agradeço ao meu Professor Orientador,

Professor Irineu Francisco de Aguiar, pelo apoio e tempo dedicado para discussões e correções desta monografia.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

LINQ – Language Integrated Query
XML – Extensible Markup Language
API – Application Programming Interface
SQL – Structured Query Language
DTD – Document Type Definitions
DOM – Document Object Model

CURRICULUM VITAE

Bruno Marangoni dos Santos Leite, nascido em trinta e um de agosto de mil novecentos e oitenta e oito na cidade de Santo André, estado de São Paulo, está cursando Tecnologia em Processamento de Dados pela Faculdade de Tecnologia de São Paulo (Fatec-SP). Trabalhou na Config Informática como Programador Jr., desenvolvendo em Delphi e .NET . Atualmente atua como Analista de Sistemas Pleno na BSI Tecnologia, desenvolvendo em .NET.

Resumo

O trabalho tem como finalidade desenvolver um material teórico, capaz de fornecer o embasamento necessário para um profissional de Tecnologia da Informação atuar no desenvolvimento, manutenção ou customização de softwares que utilizam XML e plataforma .NET , para assim aplicarem o componente LINQ como solução.

Abstract

This work aims to create a theoretical source from which IT Professionals may obtain the required knowledge from XML and .NET software development, maintenance and customization so that it is possible to apply the LINQ component as solution.

1. Introdução

No cenário atual houve uma atenção dos desenvolvedores ao LINQ devido a necessidade de se manipular XML com facilidade e velocidade para assim atender às demandas de desenvolvimento de software do mercado.

LINQ é um componente do Microsoft .NET Framework que adiciona funcionalidades de consulta em algumas linguagens de programação .NET .

XML um sistema padrão simples para criar códigos de referência.

Juntos se tornam uma poderosa maneira de manipular dados em aplicações que usam XML para troca de dados pela internet e fora dela.

O LINQ faz parte da plataforma .NET Framework, que é uma das mais usadas para desenvolvimento de software e no meio acadêmico em âmbito mundial. Aliada ao XML, que é utilizado para comunicação tanto na internet como em aplicações fora dela, formam um conjunto de requisitos que um profissional de TI da área de desenvolvimento deve conhecer para se adequar as tendências do mercado e obter vantagens no desenvolvimento de aplicações rápidas e de fácil implementação e manutenção.

O trabalho tem como finalidade desenvolver um material teórico, capaz de fornecer o embasamento necessário para um profissional de Tecnologia da Informação atuar no desenvolvimento, manutenção ou customização de softwares que utilizam XML e plataforma .NET , para assim aplicarem o componente LINQ como solução.

Para a obtenção das informações contidas neste trabalho, foi realizada uma pesquisa bibliográfica, onde foram consultados livros, artigos, páginas da Internet e manuais que tratam do assunto em questão.

2. Fundamentos do LINQ

2.1. Introdução ao LINQ

LINQ é uma metodologia que foi criada para auxiliar no desenvolvimento de software, especificamente ao manipular qualquer tipo de dados.

De acordo com PIALORSI E RUSSO (2010, p. 3), LINQ auxilia o desenvolvedor a manipular dados com muito mais rapidez e clareza na escrita.

> LINQ é um modelo de programação que introduz o conceito de buscas de primeira-classe dentro da plataforma Microsoft .NET. Porém, o completo suporte para o LINQ requer algumas extensões da linguagem utilizada. Estas extensões aumentam a produtividade, provendo uma sintaxe rápida, fácil e expressiva para manipular dados. (PIALORSI; RUSSO, 2010, p. 3)

Atualmente há vários domínios de dados, podendo ser estes: um vetor, um grafo, um documento XML, uma base de dados, um arquivo texto, uma chave de registro, uma mensagem de email e outros diversos tipos de objetos. Para manipular cada um desses é necessário utilizar de APIs específicas de cada tipo de objeto, de acordo com o funcionamento específico de cada API.

Alguns desses domínios são melhores representados através de modelos relacionais, outros são melhores representados através de modelos hierárquicos ou modelos gráficos.

Com o LINQ é possível manipular vários domínios diferentes de dados utilizando uma API única, a qual tem uma sintaxe similar a do SQL, mas sem fixar uma maneira de manipulação muito genérica o qual poderia distorcer a estrutura de dados que possuam modelos diferentes.

Para demonstrar, abaixo foi criada uma *query* feita no Microsoft Visual C# utilizando a framework 3.0, a qual retorna o nome de clientes que estão no Brasil:

```
var query =
  from cliente in Clientes
  where cliente.Pais == "Brasil"
  select cliente.NomeEmpresa;
```

O resultado da query é armazenado no objeto “query”, que é uma lista de string. É possível percorrer todo o resultado através de um laço e escrevê-lo na tela como no exemplo abaixo:

```
foreach ( string nome in query ) {
  Console.WriteLine( nome );
}
```

Para uma total compreensão da *query* escrita acima, é necessário saber a estrutura do objeto Clientes, o qual pode ser de vários tipos, podendo ser uma coleção de objetos (*collection of objects*), uma *DataTable* eu mesmo uma tabela física de uma base de dados. Isso mostra que LINQ pode ser produtivo através de ganhos com abstração e generalização de objetos.

O exemplo demonstrado acima é considerado uma query simples pois não contém nenhum relacionamento com outras entidades. De acordo com PIALORSI E RUSSO (2010, p. 9), LINQ não é limitado a apenas estruturas de dados simples.

> LINQ não é limitado a um único modelo de dados como o modelo relaciona, onde os relacionamentos entre as entidades são expressados dentro da busca e não no modelo de dados. (PIALORSI; RUSSO, 2010, p. 9).

No exemplo abaixo é demonstrado uma query na qual se obtém dados de duas estruturas de dados, na qual se relacionam entre si. Para se implementar tal query pode ser utilizado o modelo relacional:

```
var query =
  from cliente in Clientes
  join pedido in Pedidos
  on cliente.IDCliente equals pedido.IDCliente
  select new {cliente. IDCliente,cliente.NomeEmpresa,pedido.IDPedido};
```

A mesma query pode ser escrita utilizando o modelo hierárquico da seguinte forma:

```
var query =
  from cliente in Clientes
  from pedido in cliente.Pedidos
  select new {cliente.Nome, pedido.Quantidade, pedido.Produto.NomeProduto };
```

As *querys* demonstradas acima foram baseadas na seguinte estrutura:

```
public class Clientes {
  public string Nome;
  public string Cidade;
  public Pedido[] Pedidos;
}
public struct Pedido {
  public int Quantidade;
  public Produto Produto;
}
public class Produto {
  public int IDProduto;
  public decimal Preco;
  public string NomeProduto;
}
```

2.2. Fundamentos da Sintaxe do LINQ

LINQ "é baseado nos operadores de query, definido como métodos estendidos, que manipula qualquer objeto que implementa as interfaces IEnumerable<T> ou IQueryable<T>. Esta proximidade faz do LINQ uma framework de pesquisa multiuso, porque muitas coleções e tipos implementam IEnumerable<T> ou IQueryable<T> , e desenvolvedores podem definir suas próprias implementações. Esta infraestrutura de query é também altamente extensiva [...]" PIALORSI E RUSSO (2010, p. 23).

Para KLEIN (2008, p. 54), os operadores padrões utilizados no LINQ estão listados na tabela abaixo:

Tabela 1 – Operadores Padrões utilizados no LINQ

Standard Query Operator	**C#**	**Visual Basic**
All (Of T)	N/A	Into All(. . .)
Any	N/A	Into Any()
Average	N/A	Into Averate()
Cast (Of T)	An explicit range of variables	From. . .As. . .
Count	N/A	Into count()
Distinct	N/A	Distinct
GroupBy	group by	Group By

GroupJoin	join. . .in. . .on. . .into. . .	Group Join
Join	join. . .in. . .on. . .equals. . .	Join. . .As..IN. . .On. . . OR From x In..y In..Where. . .
LongCount	N/A	Into LongCount()
Max	N/A	Into Max()
Min	N/A	Into Min()
OrderBy	orderby	Order By
OrderByDescending	orderby desdending	Order By. . .Descending
Select	select	Select
SelectMany	Multiple from clauses	Multiple from clauses
Skip	N/A	Skip
SkipWhile	N/A	Skip While
Sum	N/A	Into Sum
Take	N/A	Take
TakeWhile	N/A	Take While
ThenBy	orderby	Order By
ThenByDescending	orderby descending	Order By. . .Descending
Where	where	Where

FONTE: KLEIN, SCOTT

Para se realizar consultas é necessário utilizar os operadores de projeção, que KLEIN (2008, p. 56) descreve como sendo Select e SelectMany.

De acordo com KLEIN (2008, p. 56), o operador Select projeta valores de uma única seqüência ou coleção, como abaixo:

```
var query =
  from c in contact
  where c.FirstName.StartsWith("S")
  select new {c.FirstName, c.LastName, c.EmailAddress}
```

Ainda, para KLEIN (2008, p. 56), o operador SelectMany tem a capacidade de combinar cada objeto dentre de uma única seqüência, como abaixo:

```
string[] owners =
  { new name { FirstName = "Scott", "Chris",
  Pets = new List<string>{"Yukon", "Fido"}},
  new name { FirstName = "Jason", "Steve",
  Pets = new List<string>{"Killer", "Fluffy"}},
  new name { FirstName = "John", "Joe",
```

```
  Pets = new List<string>{"Spike", "Tinkerbell"}}}

IEnumerable<string> query =
  names.AsQueryable().SelectMany(own => own.Pets);
```

O trecho de código acima resulta na lista abaixo:

```
Yukon
Fido
Killer
Fluffy
Spike
Tinkerbell
```

Para realizar restrições (filtros) nas *querys* é utilizado o operador where. O operador where é "operador de restrição. Ele aplica um critério de filtro na seqüência." KLEIN (2008, p. 57).

```
IEnumerable<string> query =
  from c in contact
  where c.FirstName.StartsWith("S")
  select new {c.FirstName, c.LastName, c.EmailAddress}
```

Igual ao SQL, o LINQ suporta operadores de ordenação. KLEIN (2008, p. 57) cita os seguintes operadores: OrderBy, OrderByDescending, ThenBy, ThenByDescending e Reverse.

O operador OrderBy "ordena os valores do resultado da seqüência em ordem ascendente." KLEIN (2008, p. 57). O código abaixo exemplifica:

```
var query =
  from c in contact
  where c.FirstName.StartsWith("S")
  orderby c.LastName
  select new {c.FirstName, c.LastName, c.EmailAddress}
```

Já o operador OrderByDescending "ordena os valores do resultado da seqüência em ordem descendente." KLEIN (2008, p. 58). O código abaixo exemplifica:

```
IEnumerable<string> query =
  from c in contact
  where c.FirstName.StartsWith("S")
  orderby c.LastName descending
  select new {c.FirstName, c.LastName, c.EmailAddress}
```

O operador ThenBy "aplica em prioridade secundária, uma ordenação ascendente à seqüência" KLEIN (2008, p. 58). O código abaixo exemplifica:

```
IEnumerable<string> query =
  from c in contact
  where c.FirstName.StartsWith("S")
  orderby c.LastName
  thenby c.FirstName
  select new {c.FirstName, c.LastName, c.EmailAddress}
```

Já o operador ThenByDescending "aplica em prioridade secundária, uma ordenação descendente à seqüência" KLEIN (2008, p. 58). O código abaixo exemplifica:

```
IEnumerable<string> query =
  (from c in contact
  where c.FirstName.StartsWith("S")
  orderby c.LastName descending
  select new {c.FirstName, c.LastName, c.
  ThenByDescending(c => c.FirstName);
```

Por último na categoria de ordenação, KLEIN (2008, p. 59) cita o operador Reverse o qual retorna o valor na ordem oposta (reversa) da qual ele é retornado da fonte de dados. O código abaixo exemplifica:

```
string[] names = {"Alex", "Chuck", "Dave", "Dinesh",
    "Joe", "John", "Sarah", "Scott", "Steve"}
string[] reversednames = names.Reverse().ToArray();
foreach (string str in reversednames)
  listbox1.Items.Add(chr)
```

Para se realizar consultas que envolvam mais de uma fonte de dados é necessário utilizar do recurso de *join*. De acordo com KLEIN (2008, p. 59), *Joining* é a ação de relacionar ou associar uma fonte de dados com uma segunda fonte de dados. As duas fontes de dados são associadas através de um valor comum ou atributo comum. A *join* do LINQ relaciona valores de fontes de dados que contenham chaves que se igualam. KLEIN cita dois operadores de JOIN: Join e GroupJoin.

O operador Join "combina uma fonte de dados a uma segunda fonte de dados, combinando através de valores iguais entra as duas fontes de dados." KLEIN (2008, p. 59). O exemplo abaixo mostra o operador Join.

```
from c in contact
join emp in employee on c.ContactID equals emp.ContactID
where c.FirstName.StartsWith("S")
orderby c.LastName
select new {emp.EmployeeID, c.FirstName, c.LastName,
c.EmailAddress, emp.Title, emp.HireDate}
```

O operador GroupJoin “combina cada valor ou elemento da primeira fonte de dados [...] com o valor selecionado correspondente da segunda fonte de dados” KLEIN (2008, p. 60). O exemplo abaixo mostra o operador GroupJoin.

```
List<Team> teams = new List<Team>{ new Team { name = "Yamaha"},
  new Team { name = "Honda"} ,
  new Team { name = "Kawasaki"} ,
  new Team { name = "Suzuki"}} ;
List<Rider> riders = new List<Rider> {
  new Rider { name = "Grant Langston", TeamName = "Yamaha"},
  new Rider { name = "Andrew Short", TeamName = "Honda"},
  new Rider { name = "James Steward", TeamName = "Kawasaki"},
  new Rider { name = " Broc Hepler", TeamName = "Yamaha"},
  new Rider { name = "Tommy Hahn", TeamName = "Honda"},
  new Rider { name = "Tim Ferry", TeamName = "Kawasaki"},
  new Rider { name = " Chad Reed", TeamName = "Yamaha"},
  new Rider { name = "Davi Millsaps", TeamName = "Honda"},
  new Rider { name = "Ricky Carmichael", TeamName = "Suzuki"},
  new Rider { name = "Kevin Windham", TeamName = "Honda"}};
var teamsandriders = teams.GroupJoin(riders,
  Team => Team.name,
  Rider => Rider.TeamName,
(team, teamRiders) => new {Team = team.name,
riders = teamRiders.Select(rider => rider.name)});
foreach (var tar in teamsandriders)
{
  listBox1.Items.Add(tar.Team);
foreach (string rider in tar.riders)
  listBox1.Items.Add(" " + rider);
}
```

O resultado desta consulta é:

```
Yamaha
Grant Langston
Broc Hepler
Chad Reed
Honda
Andrew Short
Tommy Hahn
Davi Millsaps
Kevin Windham
Kawasaki
James Stewart
Tim Ferry
Suzuki
Ricky Carmichael
```

“Agrupamento é o conceito de agrupar valores ou elementos de uma seqüência de acordo com um valor específico (seletor)” KLEIN (2008, p. 62). Para realizar agrupamentos como em SQL, LINQ possui um operador único, o GroupBy, utilizado no exemplo abaixo:

```
DataContext context = new DataContext("Initial
  Catalog=AdventureWorks;Integrated Security=sspi");
Table<SalesOrderHeader> orders = context.GetTable<SalesOrderHeader>();
var query = orders.Where(ord => ord.SalesPersonID > 0).GroupBy(order =>
order.SalesPersonID,
```

```
order => order.CustomerID);
foreach (var o in query)
{
  listBox1.Items.Add(o.Key);
foreach (int cust in o)
  listBox1.Items.Add(" " + cust);
}
```

O resultado desta consulta é:

```
268
697
47
471
548
167
...
275
504
618
17
486
269
276
510
511
259
384
650
...
```

LINQ também possui o operador de concatenação, que é “o processo de juntar dos dados de dois objetos. No LINQ, as *joins* de concatenação juntam duas coleções de dados em uma única [...]” KLEIN (2008, p. 63).

```
DataContext context = new DataContext("Initial
Catalog=@@taAdventureWorks;Integrated Security=sspi");
Table<Contact> contacts = context.GetTable<Contact>();
Table<SalesOrderHeader> orders = context.GetTable<SalesOrderHeader>();
var query = contacts.Select(con => con.LastName).Concat(orders.Select(order
@@@ta
=> order.CustomerID.ToString()));
foreach (var item in query)
{
  listBox1.Items.Add(item);
}
```

Para se agregar valores no LINQ é possível utilizar as funções de agregação, que “executam cálculos em uma parte do resultados e retornam um único valor, como uma soma ou contagem em valores de um determinado elemento.” KLEIN (2008, p. 64). No LINQ existem sete tipos de funções: Aggregate, Average, Count, LongCount, Max, Min e Sum.

A função de Aggregate “agrupa valores de uma determinada seqüência ou coleção” KLEIN (2008, p. 64). O exemplo abaixo mostra o uso da função Aggregate:

```
string Names = "Steve, Scott, Joe, John, Chris, Jason";
string[] name = Names.Split(', ');
string newName = name.Aggregate(workingName, next) =>
next + " " + workingName);
listbox.Items.Add(newName);
```

A função de Average “calcula a média de uma seqüência com vales numéricos” KLEIN (2008, p. 64). O exemplo abaixo mostra o uso da função Average:

```
List<int> quantity = new List<int> {99, 48, 120, 73, 101, 81, 56};
double average = quantity.Average();
listbox1.items.add(average);
```

Outra função de agregação é a Count, que “conta o número de elementos em uma coleção” KLEIN (2008, p. 65). O exemplo abaixo mostra o uso da função Count:

```
List<int> quantity = new List<int> {99, 48, 120, 73, 101, 81, 56};
int cnt = quantity.Count;
listbox1.items.add(cnt);
```

Similar a Count, a função LongCount “conta o número de elementos em uma coleção”, mas “ em uma coleção grande.” KLEIN (2008, p. 66). O exemplo abaixo mostra o uso da função LongCount:

```
List<Int64> quantity = new List<Int64> {99, 48, 120, 73, 101, 81, 56};
Int64 cnt = quantity.LongCount();
listbox1.items.add(cnt);
```

A função Max “retorna o valor máximo contido da seqüência. Como a função Average, Max funciona com muitos tipos de dados, incluindo decimal, integers e doubles” KLEIN (2008, p. 66). O exemplo abaixo mostra o uso da função Max:

```
List<int> quantity = new List<int> {99, 48, 120, 73, 101, 81, 56};
int cnt = quantity.Max();
listbox1.items.add(cnt);
```

Ao contrário da função Max, a função Min “retorna o valor mínimo contido da seqüência. Como a função Average, Max funciona com muitos tipos de dados, incluindo decimal, integers e doubles” KLEIN (2008, p. 67). O exemplo abaixo mostra o uso da função Min:

```
List<int> quantity = new List<int> {99, 48, 120, 73, 101, 81, 56};
int cnt = quantity.Min();
listbox1.items.add(cnt);
```

A função Sum “calcula a soma dos valores selecionados pertencentes à coleção” KLEIN (2008, p. 67). O exemplo abaixo mostra o uso da função Sum:

```
List<int> quantity = new List<int> {99, 48, 120, 73, 101, 81, 56};
int cnt = quantity.Sum();
listbox1.items.add(cnt);
```

LINQ também dispõe de operadores de definição, os quais “agem nos elementos e seqüência, e retornam uma seleção de valores.” KLEIN (2008, p. 68). Os operadores são: Distinct, Union, Intersect e Except.

O operador Distinct “remove valores duplicados da coleção e retorna valores distintos da mesma seqüência.” KLEIN (2008, p. 68), como mostra o exemplo:

```
List<int> quantity = new List<int> {1, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 9, 10};
IEnumerable<int> val = numbers.Distinct();
foreach (int num in val)
listbox1.Items.Add(num);
```

O resultado desta consulta é:

```
1
2
3
4
5
6
7
8
9
10
```

O operador Union “retona os valores distintos da união de duas seqüência ou coleções. Isto é diferente do operador concat, este retorna valores distintos, e o operador concat retorna todos os valores.” KLEIN (2008, p. 70), como mostra o exemplo:

```
int[] numbers1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ;
int[] numbers2 = { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20} ;
IEnumerable<int> union = numbers1.Union(numbers2);
foreach (int num in union)
listBox1.Items.Add(num);
```

O operador Intersect “retorna a intersecção de duas seqüências” KLEIN (2008, p. 70), como mostra o exemplo:

```
int[] numbers1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ;
int[] numbers2 = { 2, 4, 6, 8, 10} ;
IEnumerable<int> shared = numbers1.Intersect(numbers2);
foreach (int num in shared)
listBox1.Items.Add(num);
```

O resultado desta consulta é:

```
2
4
6
8
10
```

O operador Except “é o oposto do operador intersect, no qual este retorna a diferença entre dias seqüência — em outras palavras, este retorna valores que são únicos (não duplicados) em todos os valores da seqüência (valores que aparecem na primeira seqüência mas não aparecem na segunda). Em outras palavras, este é ‘os elementos da seqüência A menos os elementos da seqüência B’ ” KLEIN (2008, p. 70), como mostra o exemplo:

```
int[] numbers1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ;
int[] numbers2 = { 2, 4, 6, 8, 10} ;
IEnumerable<int> shared = numbers1.Except(numbers2);
foreach (int num in shared)
listBox1.Items.Add(num);
```

O resultado desta consulta é:

```
1
3
5
7
9
```

O operador Generation “cria novas seqüências a partir dos valores existentes nas seqüência atual” KLEIN (2008, p. 71). Dentre eles estão os operadores Empty, Range e Repeat.

O primeiro, o operador Empty “retorna uma coleção vazia de acordo com o tipo especificado” KLEIN (2008, p. 71). O exemplo a seguir mostrar o operador:

```
string[] name1 = { "Scott", "Steve"} ;
string[] name2 = { "Joe", "John", "Jim", "Josh", "Joyce"} ;
string[] name3 = { "Dave", "Dinesh", "Doug", "Doyle"} ;
List<string[]> names = new List<string[]> { name1, name2, name3} ;
IEnumerable<string> namelist = names.Aggregate(Enumerable.Empty<string>(),
(current, next) => next.Length > 2 ? current.Union(next) : current);
foreach (string item in namelist)
listBox1.Items.Add(item);
```

O resultado desta consulta é:

```
Joe
```

```
John
Jim
Josh
Joyce
Dave
Dinesh
Doug
Doyle
```

O segundo, o operador Range “cria uma coleção que contém uma seqüência de números. Ele recebe dois parâmetros. O primeiro é um valor inteiro que representa o início da seqüência, e o segundo é a quantidade de números para gerar” KLEIN (2008, p. 72). O exemplo a seguir mostrar o operador:

```
var coolmath = Enumerable.Range(1, 10);
for each (int num in coolmath)
  listbox1.Items.Add(num);
```

O resultado desta consulta é:

```
1
2
3
4
6
7
8
9
10
```

O terceiro, o operador Repeat “cria uma única seqüência de valores e a repete um número específico de vezes” KLEIN (2008, p. 73). O exemplo a seguir mostrar o operador:

```
var coolphrase = Enumerable.Repeat("LINQ ROCKS!", 10);
for each (string phrase in coolphrase)
  listbox1.Items.Add(phrase);
```

Para realizar conversões de tipos, o LINQ disponibiliza os seguintes operadores: AsEnumerable, Cast, OfType, ToArray, ToDictionary, ToList, ToLookup.

O operador AsEnumerable “retorna a entrada da query no tipo IEnumerable(Of T) ” KLEIN (2008, p. 74). O exemplo a seguir mostrar o operador:

```
DataContext context = new DataContext("Initial Catalog=AdventureWorks;@@ta
Integrated Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
IEnumerable<Contact> query =
contact.AsEnumerable().Where(con => con.FirstName.Contains("K"));
foreach (Contact item in query)
  listBox1.Items.Add(item.FirstName);
```

O resultado desta consulta é:

```
Kim
Keyley
Karel
Karen
Kris
Kevin
...
```

O operador Cast “converte o elemento de uma coleção IEnumerable para um tipo específico” KLEIN (2008, p. 74). O exemplo a seguir mostrar o operador:

```
ArrayList names = new ArrayList();
names.Add("Alex");
names.Add("Chuck");
names.Add("Dave");
names.Add("Dinesh");
names.Add("Joe");
names.Add("John");
names.Add("Sarah");
names.Add("Steve");
IEnumerable<string> query = names.Cast<string>().Select(name => name);
foreach (string item in query)
  listBox1.Items.Add(item);
```

O operador OfType “permite filtrar elementos de uma objeto IEnumerable baseado em um tipo específico” KLEIN (2008, p. 75). O exemplo a seguir mostrar o operador:

```
ArrayList names = new ArrayList(7);
names.Add("Scott");
names.Add(1);
names.Add("Dave");
names.Add(2);
names.Add("Dave");
names.Add(3);
names.Add("Steve");
names.Add(4);
names.Add("Joe");
IEnumerable<int> query = names.OfType<int>();
foreach (int item in query)
  listBox1.Items.Add(item);
```

O resultado desta consulta é:

```
1
2
3
4
```

O operador ToArray “cria um vetor a partir de uma seqüência IEnumerable” KLEIN (2008, p. 75). O exemplo a seguir mostrar o operador:

```
DataContext context = new DataContext("Initial Catalog =
AdventureWorks;Integrated Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = contact.Select(con => con.FirstName).ToArray();
foreach (string item in query)
  listBox1.Items.Add(item);
```

O resultado desta consulta é:

```
Gustavo
Catherine
Kim
Humberto
Pilar
Frances
Margeret
Carla
Jay
```

O operador ToDictionary "insere todos os elementos retornados na seqüência em um Dictionary(Of TKey, TValue)." KLEIN (2008, p. 75). O exemplo a seguir mostrar o operador:

```
DataContext context = new DataContext("Initial Catalog =
AdventureWorks;Integrated Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
Dictionary<string, Contact> dict = contact.ToDictionary(con => con.FirstName);
foreach (KeyValuePair<string, Contact> item in dict)
  listBox1.Items.Add(item.Key + " " + item.Value.FirstName + " " +
  item.Value.LastName);
```

O resultado desta consulta é:

```
1 Gustavo Achong
2 Catherine Abel
3 Kim Abercrombie
4 Humberto Acevedo
5 Pilar Ackerman
6 Frances Adams
7 Margeret Smith
8 Carla Adams
```

O operador ToList "converte uma seqüência IEnumerable em uma coleção List(Of T). Isto também força a execução imediata da query." KLEIN (2008, p. 76). O exemplo a seguir mostrar o operador:

```
DataContext context = new DataContext("Initial Catalog =
AdventureWorks;Integrated Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = (from c in contact
select c.FirstName).ToList();
foreach (string item in query)
  listBox1.Items.Add(item);
```

O operador ToLookup "coloca o elemento retornado em uma Lookup(Of TKey, TElement), baseado em uma chave específica" KLEIN (2008, p. 77). O exemplo a seguir mostrar o operador:

```
DataContext context = new DataContext("Initial Catalog =
AdventureWorks;Integrated Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
```

```
Lookup<string, string> lkp = contact.ToLookup(con => con.FirstName,
con => con.MiddleName + " " + con.LastName);
foreach (IGrouping<string, string> lkpgrp in lkp)
{
  listBox1.Items.Add(lkpgrp.Key);
foreach (string item in lkpgrp)
  listBox1.Items.Add(" " + item);
}
```

Os operadores de elementos “retornam um único e específico elemento de uma seqüência” KLEIN (2008, p. 77). Os operadores são DefaultEmpty, ElementAt, ElementAtOrDefault, First, Last, FirstOrDefault, LastOrDefault, Single e SingleOrDefault.

O operador DefaultIfEmpty “substitui uma coleção vazia com uma coleção que contenha um valor padrão. Ela pode ser utilizada para retornar um valor padrão no caso da seqüência retornada ser vazia [...]” KLEIN (2008, p. 78). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial Catalog =
AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.FirstName.StartsWith("Z")
select c.FirstName;
foreach (string item in query.DefaultIfEmpty())
  listBox1.Items.Add(item);
```

O operador ElementAt “retorna um elemento de um determinado índice da coleção. A coleção inicia no zero e o valor retornado é o elemento na posição especificada” KLEIN (2008, p. 78). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial Catalog =
AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.FirstName.StartsWith("S")
select c.FirstName;
listBox1.Items.Add(query.ElementAt(0));
```

O resultado desta consulta é:

```
Zheng
```

O operador ElementAtOrDefault “combina o operador ElementAt com alguma das funcionalidades do operador DefaultIfEmpty retornando o elemento do índice específico ou o valor padrão se o índice estiver fora do intervalo” KLEIN (2008, p. 79). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial Catalog =
AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.FirstName.StartsWith("S")
select c.FirstName;
listBox1.Items.Add(query.ElementAtOrDefault(50000));
```

O operador First “retorna o primeiro elemento de uma coleção” KLEIN (2008, p. 79). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial Catalog = @@ta
AdventureWorks;Integrated Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.FirstName.StartsWith("S")
select c.FirstName;
listBox1.Items.Add(query.First());
```

O operador Last “retorna o último elemento de uma coleção” KLEIN (2008, p. 80). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial Catalog = @@ta
AdventureWorks;Integrated Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.FirstName.StartsWith("S")
select c.FirstName;
listBox1.Items.Add(query.Last());
```

O operador FirstOrDefault “retorna o primeiro elemento de uma coleção ou, se nenhum elemento for encontrado, um valor padrão” KLEIN (2008, p. 80). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial
Catalog=AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.FirstName.StartsWith("ZZ")
select c.FirstName;
listBox1.Items.Add(query.FirstOrDefault());
```

O operador LastOrDefault “retorna o último elemento de uma coleção ou, se nenhum elemento for encontrado, um valor padrão” KLEIN (2008, p. 80). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial
Catalog=AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.FirstName.StartsWith("ZZ")
select c.FirstName;
listBox1.Items.Add(query.LastOrDefault());
```

O operador Single “retorna um único elemento da seqüência, ou o único elemento que obedeça a condição especificada” KLEIN (2008, p. 80). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial
Catalog=AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.LastName.Equals("Kobylinski")
select c.FirstName;
listBox1.Items.Add(query.Single());
```

O operador SingleOrDefault “retorna um único elemento da seqüência, mas também retorna o valor padrão se nenhum elemento for encontrado” KLEIN (2008, p. 82). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial
Catalog=AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.LastName.StartsWith("Kobylinski")
select c.FirstName;
listBox1.Items.Add(query.SingleOrDefault());
```

Os operadores de igualdade “comparam duas seqüências para verificar se seus elementos são iguais. Seqüências são consideradas iguais de elas tiverem o mesmo número de elementos e os valores dos elementos forem os mesmos” KLEIN (2008, p. 82).

O operador SequenceEqual “determina se duas coleções são iguais” KLEIN (2008, p. 81). O exemplo abaixo mostra o uso do operador:

```
int[] numbers1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ;
int[] numbers2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ;
bool eq = numbers1.SequenceEqual(numbers2);
listBox1.Items.Add(eq);
```

Os operadores de quantidade “retorna um Boolean que indica se um ou mais elementos na seqüência obedece certa condição” KLEIN (2008, p. 83). Os operadores são: All, Any e Contains.

O operador All “determina se todos os valores na coleção satisfazem uma condição específica” KLEIN (2008, p. 83). O exemplo abaixo mostra o uso do operador:

```
Names[] friends = {new Names { Name = "Steve"},
  new Names { Name = "Dave"},
  new Names { Name = "Joe"},
  new Names { Name = "John"},
  new Names { Name = "Bill"},
};
bool firstnames = friends.All(name => name.Name.StartsWith("J"));
listBox1.Items.Add(firstnames).ToString();
```

O operador Any “determina se qualquer um dos valores da coleção satisfaz uma condição ou se a seqüência contém algum valor” KLEIN (2008, p. 84). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial
Catalog=AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
where c.LastName.StartsWith("Z")
select c.FirstName;
listBox1.Items.Add(query.Any());
```

O operador Contains “determina se uma coleção contém um valor específico” KLEIN (2008, p. 85). O exemplo abaixo mostra o uso do operador:

```
DataContext context = new DataContext("Initial
Catalog=AdventureWorks;Integrated
Security=sspi");
Table<Contact> contact = context.GetTable<Contact>();
var query = from c in contact
select c.LastName;
listBox1.Items.Add(query.Contains("Kleinerman"));
```

Os operadores de particionamento “dividem uma única entrada em duas ou mais seções ou seqüências sem mudar a ordem dos elementos, e então retornando uma

das novas seções formadas” KLEIN (2008, p. 86). Os operadores são: Skip, Skipwhile, Take e TakeWhile..

O operador Skip “ignora elementos acima de uma posição especificada em uma seqüência” KLEIN (2008, p. 86). O exemplo abaixo mostra o uso do operador:

```
Int[] randomNumbers = {86, 2, 77, 94, 100, 65, 5, 22, 70};
IEnumerable<int> skipLowerFour =
randomNumbers.OrderBy(num => num).Skip(4);
foreach (int number in skipLowerFour)
  listbox1.Items.Add(number);
```

O resultado desta consulta é:

```
70
77
86
94
100
```

O operador SkipWhile “ignora elementos em uma determinada condição, e continua ignorando os elementos enquanto a condição for verdadeira” KLEIN (2008, p. 86). O exemplo abaixo mostra o uso do operador:

```
Int[] randomNumbers = {86, 2, 77, 94, 100, 65, 5, 22, 70, 55, 81, 66, 45};
IEnumerable<int> skipLessThan50 =
randomNumbers.OrderBy(num => num).SkipWhile(num =>
num < 50);
foreach (int number in skipLowerFour)
  listbox1.Items.Add(number);
```

O resultado desta consulta é:

```
55
65
66
70
77
81
86
94
100
```

O operador Take “retorna continuamente elementos da seqüência, iniciando no começo até uma posição especificada” KLEIN (2008, p. 87). O exemplo abaixo mostra o uso do operador:

```
Int[] randomNumbers = {86, 2, 77, 94, 100, 65, 5, 22, 70, 55, 81, 66, 45};
IEnumerable<int> takeTopFour =
randomNumbers.OrderByDescending(num => num).Take(4);
foreach (int number in takeTopFour)
  listbox1.Items.Add(number);
```

O resultado desta consulta é:

```
100
94
86
81
```

O operador TakeWhile “retorna elementos baseados em uma condição específica, e continua retornando elemento enquanto a condição seja verdadeira” KLEIN (2008, p. 87). O exemplo abaixo mostra o uso do operador:

```
Int[] randomNumbers = {86, 2, 77, 94, 100, 65, 5, 22, 70, 55, 81, 66, 45};
IEnumerable<int> takeGreaterThan50 =
randomNumbers.OrderByDescending(num => num).TakeWhile(num => num > 50);
foreach (int number in takeGreaterThan50)
  listbox1.Items.Add(number);
```

O resultado desta consulta é:

```
100
94
86
81
77
70
66
65
55
```

3. Fundamentos do XML

3.1. O que é XML

De acordo com HUNTER (2007, p. 3), XML (*Extensible Markup Language*) é uma tecnologia com o objetivo de descrever e estruturar dados. O XML pode ter seu conteúdo armazenado em um arquivo texto normal. Seu conteúdo consiste em markups que definem e armazenam qualquer tipo de dados, formando o arquivo XML, como no exemplo abaixo:

```
<name>
   <first>John</first>
   <last>Doe</last>
</name>
```

"XML também agrupa as informações de forma hierárquica" HUNTER (2007, p. 15). Esses itens podem ser por relacionamentos Pai/Filho e irmão/irmão, chamados Elementos, como no exemplo abaixo:

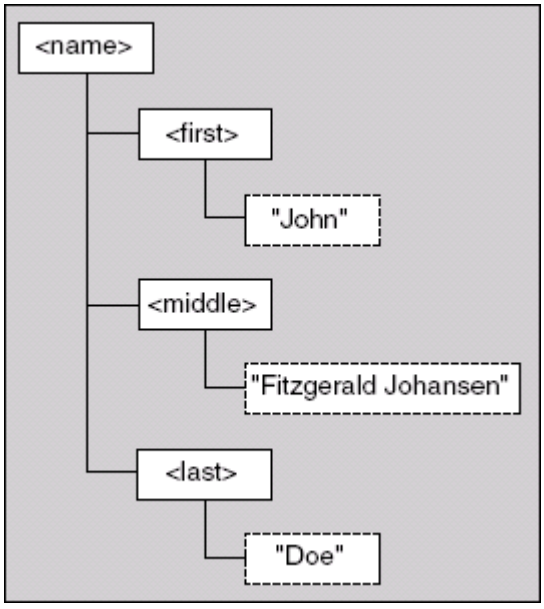

Figura 1 – Hierarquia de um XML

Na figura acima, <name> é o pai de <first>. <first>, <middle> e <last> são irmãos um do outro (e são todos filhos de <name>). O texto que está contido dentro dos

elementos citado é considerado filho do elemento. Chama-se esta estrutura de árvore, e qualquer parte da árvore que contenha um filho é chamada de galho (ou ramificação), e os elementos que não contenham filhos são chamados de folhas.

Como o elemento <name> contém apenas elementos, e não contém texto, diz-se que ele possui elementos de conteúdo. Para os elementos <first>,<middle> e <last>, como eles não possuem elementos de conteúdo, mas sim apenas texto, diz-se que eles possuem conteúdo simples.

De acordo com HUNTER (2007, p. 16), elementos podem conter tanto texto como outros elementos, que neste caso eles são chamados de conteúdo simples, como no exemplo abaixo:

```
<doc>
   <parent>this is some <em>text</em> in my element</parent>
</doc>
```

A estrutura do código acima é demonstrada na figura abaixo:

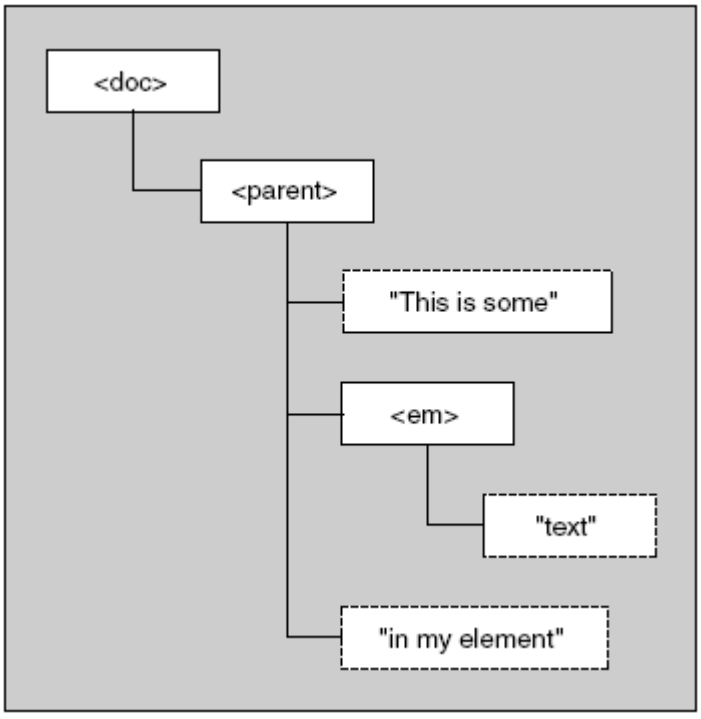

Figura 2 – Hierarquia de um XML com conteúdo misto

3.2. XML bem-formados

HUNTER (2007, p. 22) explica que um XML bem formado é aquele que atende corretamente as regras de sintaxe da norma XML 1.0. O processador de XML é chamado *parser*, e ele simplesmente interpreta o XML e fornece à aplicação as informações que ela necessita. Muitas aplicações contêm *parsers* de XML, sendo alguns exemplos: Microsoft Internet Explorer, Apache Xerces e Expad.

De acordo com HUNTER (2007. p. 24), o texto começado com um caractere < e terminado com o caractere > é chamado tag de XML. No exemplo abaixo é demonstrado um XML com tags:

```
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

As *tags* são formadas em pares, na qual uma abre (no caso do exemplo acima, <name>), e a outra *tag* fecha (</name>). Isto é chamado de *tag* de abertura e *tag* de fechamento.

Para que o *parser* consiga interpretar corretamente as *tags*, é necessário que as mesmas sejam escritas corretamente, conforme o exemplo acima. Escritas contendo espaços entre o caractere de abertura e o nome da *tag* causam um erro de sintaxe, acarretando assim em um XML mal formado. O exemplo abaixo mostra um XML uma *tag* escrita incorretamente:

```
< first >John< /first >
```

As regras à serem respeitada para que um XML seja bem formado, de acordo com HUNTER (2007, p. 31), são:

- Todo *tag* de abertura deve ter uma *tag* de fechamento, a não ser que ela se auto encerre (<exemplo/>).

- Elementos não podem se sobrepor, devem estar propriamente posicionados.
- Documentos XML só podem ter um elemento raiz.
- Os nomes dos elementos devem obedecer às convenções de nomes do XML.
- XML é *case sensitive*.
- XML irá manter espaços em branco.

De acordo com HUNTER (2007, p. 39), em adição às *tags* e elementos, documentos XML podem também incluir atributos. Atributos são simplesmente pares de nome/valor associados a um elemento. Eles podem ser adicionados às *tags* de abertura, mas não às de encerramento, como no exemplo abaixo:

```
<name nickname=”Shiny John”>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
  <last>Doe</last>
</name>
```

Atributos podem conter valores mesmo que estes sejam vazio ou branco (“”). Os valores devem estar entre aspas. O exemplo abaixo dois atributos escritos para HTML, mas que em XML não é permitido:

```
<input checked>
<input checked=true>
```

Em relação ao uso dos atributos, de acordo com HUNTER (2007, p. 44), faz sentido utilizar atributos quando se pretende separar um tipo de informação de outra.

Para se identificar um arquivo XML em alguns sistemas operacionais basta nomear o arquivo como .xml, mas em outros sistemas isso não será o bastante. Neste caso é possível utilizar o recurso de cabeçalho do XML, o qual fornece ao *parser* mais algumas informações sobre o documento XML. O exemplo abaixo mostra uma declaração comum de um XML com cabeçalho:

```
<?xml version=’1.0’ encoding=’UTF-16’ standalone=’yes’?>
<name nickname=’Shiny John’>
  <first>John</first>
  <!--John lost his middle name in a fire-->
  <middle/>
  <last>Doe</last>
</name>
```

Algumas regras do cabeçalho de um XML:

- A declaração inicia com os caracteres <?xml e termina com ?> .
- Se for incluída a declaração, é necessário incluir a versão, mas atributos *enconding* (codificação) e standalone são opcionais.
- Os atributos versão, codificação e standalone devem estar nesta ordem.
- A versão pode ser 1.0 ou 1.1 .
- A declaração XML precisa estar no início do arquivo.

De acordo com HUNTER (2007, p. 51), o atributo de versão específica qual versão de especificação de XML o documento está escrito Há duas versões de especificação XML, 1.0 e 1.1 . Aparentemente ambas as versões são as mesma coisas, se diferenciando apenas pelos tipo de codificação suportadas.

Enconding é o tipo de codificação utilizada para representar os caracteres do XML. Alguns tipos de codificação permitem certos caracteres como acentuação, outras não.

HUNTER (2007, p. 53) explica que StandAlone indica se o documento existe por si só, sem dependência de nenhum outro arquivo. Caso contrário o documento XML pode ter alguma dependência de um arquivo DTD.

Para se escrever alguns caracteres especiais como < ou &, HUNTER (2007, p. 60) explica que é necessário substituir qualquer caractere < por < e caractere & por & . Os seguintes caracteres são considerados especiais:

- amp;—o caractere &
- <— o caractere <
- >— o caractere >

- '— o caractere ‘
- "— o caractere “

3.3. XML Namespaces

Para HUNTER (2007, p. 67), com XML *namespaces* é possível ter diferentes elementos e atributos de diferentes tipos de documentos XML combinados dentro de outro documento, ou até mesmo processar múltiplos documentos simultaneamente.

Para usar XML *namespaces* em um documento XML, são atribuídos nomes qualificados aos elementos (QName). Eles são formados de duas partes: a parte local, que é o nome do elemento, e o nome da *namespace* a qual o elemento pertence. O exemplo abaixo mostra um XML com *namespace*:

```
<?xml version=”1.0”?>
<pers:person>
  <pers:name>
    <pers:title>Sir</pers:title>
    <pers:first>John</pers:first>
    <pers:middle>Fitzgerald Johansen</pers:middle>
    <pers:last>Doe</pers:last>
  </pers:name>
  <pers:position>Vice President of Marketing</pers:position>
  <pers:résumé>
    <xhtml:html>
      <xhtml:head><xhtml:title>Resume of John Doe</xhtml:title></xhtml:head>
      <xhtml:body>
        <xhtml:h1>John Doe</xhtml:h1>
        <xhtml:p>John’s a great guy, you know?</xhtml:p>
      </xhtml:body>
    </xhtml:html>
  </pers:résumé>
</pers:person>
```

4. LINQ aplicado a XML

4.1. Manipulando XML com LINQ

De acordo com PIALORSI E RUSSO (2010, p. 360), LINQ aplicado a XML provê o poder do DOM e a expressividade do XPath e XQuery através de métodos que podem manipular e buscar em memória elementos em um XML. Considerando o seguinte XML introdutório:

```
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
<customer id="C01">
  <firstName>Paolo</firstName>
  <lastName>Pialorsi</lastName>
  <addresses>
    <address type="email">paolo@devleap.it</address>
    <address type="url">http://www.devleap.it/</address>
    <address type="home">Brescia - Italy</address>
  </addresses>
</customer>
```

Para construir em tempo de execução no Microsoft Visual C#, sem usar o LINQ,o mesmo XML, o código seria algo similar ao exemplo abaixo:

```
// Create the XmlDocument
XmlDocument customerDocument = new XmlDocument();
// Define processing instruction and document element (root element)
customerDocument.AppendChild(customerDocument.CreateProcessingInstruction
("xml", "version='1.0' encoding='UTF-16' standalone='yes'"));
customerDocument.AppendChild(customerDocument.CreateElement("customer"));
customerDocument.DocumentElement.SetAttribute("id", "C01");
// Create and add "firstName" child element to the document element
XmlElement firstNameElement = customerDocument.CreateElement("firstName");
firstNameElement.InnerText = "Paolo";
customerDocument.DocumentElement.AppendChild(firstNameElement);
// Create and add "lastName" child element to the document element
XmlElement lastNameElement = customerDocument.CreateElement("lastName");
lastNameElement.InnerText = "Pialorsi";
customerDocument.DocumentElement.AppendChild(lastNameElement);
// Create "addresses" element
XmlElement addressesElement = customerDocument.CreateElement("addresses");
// Create and add "email address" child element to the "addresses" element
XmlElement emailAddressElement = customerDocument.CreateElement("address");
emailAddressElement.SetAttribute("type", "email");
emailAddressElement.InnerText = "paolo@devleap.it";
addressesElement.AppendChild(emailAddressElement);
// Create and add "url address" child element to the "addresses" element
XmlElement urlAddressElement = customerDocument.CreateElement("address");
urlAddressElement.SetAttribute("type", "url");
urlAddressElement.InnerText = "http://www.devleap.it/";
addressesElement.AppendChild(urlAddressElement);
// Create and add "home address" child element to the "addresses" element
XmlElement homeAddressElement = customerDocument.CreateElement("address");
homeAddressElement.SetAttribute("type", "home");
homeAddressElement.InnerText = "Brescia - Italy";
addressesElement.AppendChild(homeAddressElement);
// Add "addresses" child element to the document element
customerDocument.DocumentElement.AppendChild(addressesElement);
```

Para este código acima, PIALORSI E RUSSO (2010, p. 361) observa que utilizar a sintaxe para manipular DOM requer muito código *procedural* para definir cada nó do XML. Agora, no exemplo abaixo o mesmo documento criado com LINQ:

```
XDocument customer =
  new XDocument(
  new XDeclaration("1.0", "UTF-16", "yes"),
  new XElement("customer",
  new XAttribute("id", "C01"),
  new XElement("firstName", "Paolo"),
  new XElement("lastName", "Pialorsi"),
  new XElement("addresses",
  new XElement("address",
  new XAttribute("type", "email"),
    "paolo@devleap.it"),
  new XElement("address",
  new XAttribute("type", "url"),
    "http://www.devleap.it/"),
  new XElement("address",
  new XAttribute("type", "home"),
    "Brescia - Italy"))));
```

Para PIALORSI E RUSSO (2010, p. 362), o exemplo anterior é chamado construção funcional. O código revela quão simples e intuitivo é definir a hierarquia dos nós de um XML usando LINQ aplicado a XML.

Também é possível criar um XML dinamicamente atribuindo ao componente diretamente seu conteúdo, através de uma literal que representa o conteúdo de um XML, como no exemplo abaixo, apenas possível no Microsoft Visual Basic 2008:

```
Dim customerXml As XDocument =
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
<customer id="C01">
<firstName>Paolo</firstName>
<lastName>Pialorsi</lastName>
<addresses>
<address type="email">paolo@devleap.it</address>
<address type="url">http://www.devleap.it/</address>
<address type="home">Brescia - Italy</address>
</addresses>
</customer>
```

É possível notar o poder e a expressividade da sintaxe, o qual traz muita facilidade ao desenvolvedor. O exemplo a seguir mostra a facilidade para se fazer uma consulta no mesmo XML e escrevê-lo na tela:

```
foreach(XElement a in customer.Descendants("addresses").Elements()) {
   Console.WriteLine(a);
}
```

De acordo com PIALORSI E RUSSO (2010, p. 363), é possível usar LINQ para buscar conteúdo em XML de maneira completamente independente, e usando LINQ aplicado a XML como única API. Esta nova API foi construída com o Infoset do *World Wide Web Consortium* (W3C). A figura abaixo mostra o modelo hierárquico da API do LINQ:

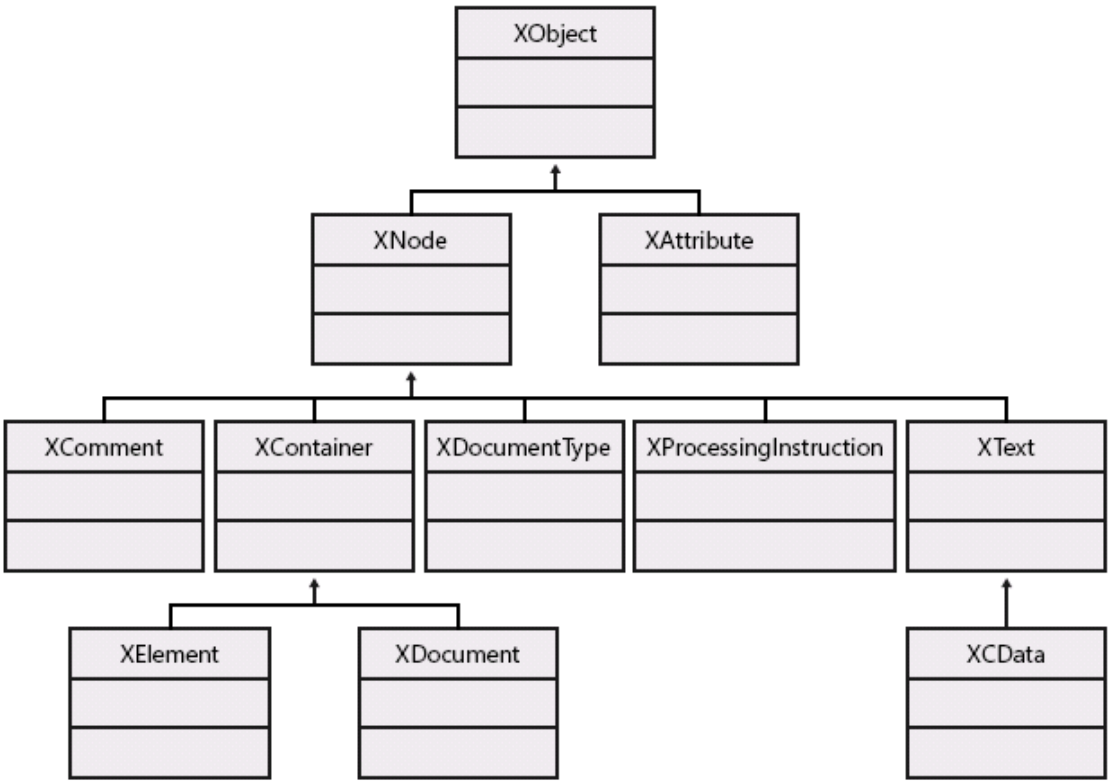

Figura 3 – Modelo Hierárquico da API do LINQ

A classe XDocument, de acordo com o PIALORSI E RUSSO (2010, p. 364), representa a *Infoset* de uma instância de um documento XML, o qual contém o nó raiz. O exemplo abaixo mostra os construtores da classe:

```
public XDocument();
public XDocument(params object[] content);
public XDocument(XDeclaration declaration, params Object[] content);
public XDocument(XDocument other);
```

A classe XElement, para PIALORSI E RUSSO (2010, p. 365), é uma das principais classes do LINQ aplicado a XML. A classe é usada para armazenar e

manipular elementos de um documento XML. O exemplo abaixo mostra os construtores da classe:

```
public XElement(XElement other);
public XElement(XName name);
public XElement(XStreamingElement other);
public XElement(XName name, object content);
public XElement(XName name, params object[] content);
```

O exemplo abaixo mostra a criação de um XML e adicionando nós com elementos, utilizando um objeto DOM:

```
XmlDocument customerDocument = new XmlDocument();
XmlElement customerElement = customerDocument.CreateElement("customer");
XmlElement firstNameElement = customerDocument.CreateElement("firstName");
firstNameElement.InnerText = "Paolo";
customerElement.AppendChild(firstNameElement);
customerDocument.AppendChild(customerElement);
```

Agora no exemplo abaixo, o mesmo XML criado com LINQ, no qual é instanciado um objeto da classe XElement, utilizando a declaração através de literal do Microsoft Visual Basic:

```
Dim customerName As String = "Paolo"
Dim customerTag As XElement =
<customer>
<firstName><%= customerName %></firstName>
</customer>
```

De acordo com PIALORSI E RUSSO (2010, p. 368), é possível converter uma instância de um XElement para string chamando o método ToString. Além disso, XElement dispõe de cast direto do seu conteúdo utilizando implementação do operador explicitamente. O exemplo abaixo mostra um *cast* explícito:

```
XElement order =
new XElement("order",
new XElement("quantity", 10),
new XElement("price", 50),
new XAttribute("idProduct", "P01"));
Decimal orderTotalAmount =
(Decimal)order.Element("quantity") *
(Decimal)order.Element("price");
Console.WriteLine("Order total amount: {0}", orderTotalAmount);
```

Além do XElement, PIALORSI E RUSSO (2010, p. 369) explicam o XAttribute, que "a classe representa uma instância de um atributo de XML.". Eles explicam também que é possível "adicionar ele a qualquer XContainer (por exemplo um

XDocument ou XElement) usando seus construtores e o construtor funcional do LINQ aplicado a XML.”. O exemplo abaixo mostra o uso do XAttribute:

```
XElement customerTag = new XElement(
  "customer",
  new XAttribute("id", "C01"), // The attribute added to customerTag
  "Paolo Pialorsi");
```

De acordo com PIALORSI E RUSSO (2010, p. 370), a classe base de muitas classes X* é a XNode. De acordo com eles, ela implementa o árvore inteira do XML, provendo métodos para adicionar, mover, remover e substituir nós. O exemplo abaixo mostra o uso da XNodes:

```
XElement customer = XElement.Load(@"customer.xml");
// This sentence selects the first address element, child of addresses
XElement firstAddress =
(customer.Descendants("addresses").Elements("address")).First();
firstAddress.AddAfterSelf(
new XElement("address",
new XAttribute("type", "IT-blog"),
"http://blogs.devleap.com/"),
new XElement("address",
new XAttribute("type", "US-blog"),
"http://weblogs.asp.net/PaoloPia/"));
```

Como mostrado no exemplo acima, é possível adicionar vários nós em apenas uma operação, graças aos *overloads* dos métodos, como abaixo:

```
public void AddAfterSelf(Object content);
public void AddBeforeSelf(Object content);
public void AddAfterSelf(params Object[] content);
public void AddBeforeSelf(params Object[] content);
```

No LINQ aplicado a XML também é possível manipular *namespaces* de XML com grande facilidade. No exemplo abaixo é criado um XML utilizando o modo clássico através de objeto DOM:

```
XmlDocument document = new XmlDocument();
XmlElement customer = document.CreateElement("c", "customer",
"http://schemas.devleap.com/Customer");
document.AppendChild(customer);
XmlElement firstName = document.CreateElement("c", "firstName",
"http://schemas.devleap.com/Customer");
firstName.InnerText = "Paolo Pialorsi";
customer.AppendChild(firstName);
```

O resultado do código abaixo é o seguinte XML:

```
<c:customer xmlns:c="http://schemas.devleap.com/Customer">
<c:firstName>Paolo Pialorsi</c:firstName>
</c:customer>
```

O mesmo XML pode ser criado com muito mais facilidade através da sintaxe LINQ, como no exemplo abaixo:

```
XNamespace ns = "http://schemas.devleap.com/Customer";
XElement customer = new XElement(ns + "customer",
new XAttribute("id", "C01"),
new XElement(ns + "firstName", "Paolo"),
new XElement(ns + "lastName", "Pialorsi"));
```

4.2. Buscando dados em nós com LINQ

PIALORSI E RUSSO (2010, p. 385) explicam que é possível utilizar os métodos padrão de busca para realizar busca em XML com LINQ.

No exemplo abaixo é realizado uma busca que em um XML através de um atributo:

```
XElement xmlCustomer = XElement.Load(@"..\..\customer.xml");
Console.WriteLine("Attributes with name \"id\" count: {0}",
xmlCustomer.Attributes().Where(a => a.Name == "id").Count());
Console.WriteLine("\"id\" attribute value: {0}",
xmlCustomer.Attribute("id").Value);
```

O resultado da busca será:

```
Attributes with name "id" count: 1
"id" attribute value: C01
```

Para PIALORSI E RUSSO (2010, p. 386), o método Element interage através dos nós filhos do XContainer e retorna o primeiro XElement, o qual o nome corresponde ao argumento fornecido no parâmetro XName. O exemplo abaixo mostra:

```
XNamespace ns = "http://schemas.devleap.com/Customers";
XElement xmlCustomers = new XElement(ns + "customers",
from c in customers
where c.Country == Countries.Italy
select new XElement(ns + "customer",
  new XAttribute("name", c.Name),
  new XAttribute("city", c.City),
  new XAttribute("country", c.Country)));
XElement element = xmlCustomers.Element(ns + "customer");
```

Para se obter todos os elementos “customer”, é utilizado o método Elements:

```
var elements = xmlCustomers.Elements();
foreach (XElement e in elements) {
  Console.WriteLine(e);
}
```

O resultado será:

```
<customer name="Paolo" city="Brescia" country="Italy" />
```

```
<customer name="Marco" city="Torino" country="Italy" />
```

De acordo com PIALORSI E RUSSO (2010, p. 392), a classe XNode provê vários métodos que são úteis para obter elementos e nós relacionados a um nó. O exemplo abaixo mostra os *overloads* de métodos utilizados para filtrar elementos por XName:

```
public IEnumerable<XElement> ElementsBeforeSelf();
public IEnumerable<XElement> ElementsBeforeSelf(XName name);
public IEnumerable<XElement> ElementsAfterSelf();
public IEnumerable<XElement> ElementsAfterSelf(XName name);
```

Além dos métodos citados, há também os métodos NodesBeforeSelf e NodesAfterSelf que retornam a seqüência do tipo IEnumerable<XNode> que contém todos os nós, como no exemplo:

```
public IEnumerable<XNode> NodesAfterSelf();
public IEnumerable<XNode> NodesBeforeSelf();
```

PIALORSI E RUSSO (2010, p. 392) explicam que ambos os métodos retornam qualquer nó (XNode), excluindo atributos. O exemplo abaixo mostra o uso dos métodos NodesBeforeSelf e NodesAfterSelf:

```
XElement customer = XElement.Load(@"..\..\customer.xml");
var firstAddress = customer.Element("addresses").FirstNode;
var nodesAfterSelf = firstAddress.NodesAfterSelf();
Console.WriteLine("Here is the first address:\n\t{0}",
firstAddress);
Console.WriteLine("There are {0} addresses after the first address",
nodesAfterSelf.Count());
foreach (var addressNode in nodesAfterSelf) {
  Console.WriteLine("\t{0}", addressNode);
}
Console.WriteLine();
var lastAddress = customer.Element("addresses").LastNode;
var nodesBeforeSelf = lastAddress.NodesBeforeSelf();
Console.WriteLine("Here is the last address:\n\t{0}",
lastAddress);
Console.WriteLine("There are {0} addresses before the last address",
nodesBeforeSelf.Count());
foreach (var addressNode in nodesBeforeSelf) {
  Console.WriteLine("\t{0}", addressNode);
}
```

O resultado é mostrado abaixo:

```
Here is the first address:
<address type="email">paolo@devleap.it</address>
There are 2 addresses after the first address
<address type="url">http://www.devleap.it/</address>
<address type="home">Brescia - Italy</address>
Here is the last address:
<address type="home">Brescia - Italy</address>
There are 2 addresses before the last address
<address type="email">paolo@devleap.it</address>
```

```
<address type="url">http://www.devleap.it/</address>
```

O último método citado por PIALORSI E RUSSO (2010, p. 393) é o InDocumentOrder, o qual ordena uma seqüência de nós IEnumerable<XNode> relacionados a um mesmo XDocument usando uma classe XNodeDocumentOrderParser, que tem seu comportamento implementado no método CompareDocumentOrder. Este método entendido é muito útil quando se necessita selecionar nós que estão ordenados na ordem que aparecem no documento. O exemplo abaixo mostra:

```
foreach (XNode a in xmlCustomers.DescendantsAndSelf().InDocumentOrder()) {
  Console.WriteLine("+ " + a);
}
```

PIALORSI E RUSSO (2010, p. 395) citam um exemplo no qual se pretende criar uma seqüência de itens nos quais os valores se encontram em um XML. Para escrevê-lo em LINQ seria conforme o trecho abaixo:

```
var customersFromXml =
from c in xmlCustomers.Elements("customer")
where (String)c.Attribute("country") == "Italy"
orderby (String)c.Element("name")
select new {
Name = (String)c.Element("name"),
City = (String)c.Attribute("city")
};
foreach (var customer in customersFromXml) {
  Console.WriteLine(customer);
} var cities = xmlCustomers.DescendantsAndSelf("city");
Console.WriteLine("\nBefore XML source modification");
foreach (var city in cities) {
  Console.WriteLine(city);
}
```

O resultado será:

```
{ Name = Marco, City = Torino }
{ Name = Paolo, City = Brescia }
```

Em uma situação hipotética na qual é necessário ler uma lista de clientes e seus pedidos de um XML. O exemplo abaixo mostra:

```
public class Customer {
  public string Name;
  public string City;
  public Countries Country;
  public Order[] Orders;
  public override string ToString(){
```

```
    return String.Format("Name: {0} - City: {1} - Country: {2}",
      this.Name, this.City, this.Country);
  }
}
public class Order {
  public int IdOrder;
  public int Quantity;
  public bool Shipped;
  public string Month;
  public int IdProduct;
  public override string ToString() {
    return String.Format("IdOrder: {0} - IdProduct: {1} - Quantity:
      {2} - Shipped: {3} - Month: {4}",
    this.IdOrder, this.IdProduct, this.Quantity, this.Shipped,
    this.Month);
  }
}
```

O XML de origem dos dados poderia ser similar a este abaixo:

```
<?xml version="1.0" encoding="utf-8"?>
<customers>
  <customer name="Paolo" city="Brescia" country="Italy">
    <orders>
      <order id="1" idProduct="1" quantity="3" shipped="false" month="January"
/>
      <order id="2" idProduct="2" quantity="5" shipped="true" month="May" />
    </orders>
  </customer>
  <customer name="Marco" city="Torino" country="Italy">
    <orders>
      <order id="3" idProduct="1" quantity="10" shipped="false" month="July"
/>
      <order id="4" idProduct="3" quantity="20" shipped="true"
month="December" />
    </orders>
  </customer>
</customers>
```

De acordo com PIALORSI E RUSSO (2010, p. 398), é possível carregar o conteúdo do XML através do LINQ usando um objeto XElement, e então buscar em seus nós para construir uma lista de Customers e Orders em memória, como no exemplo abaixo:

```
XElement xmlCustomers = XElement.Load("customersWithOrdersDataSource.xml");
var customersWithOrders =
from c in xmlCustomers.Elements(“customer”)
select new Customer {
Name = (String)c.Attribute(“name”),
City = (String)c.Attribute(“city”),
Country = (Countries)Enum.Parse(typeof(Countries),
(String)c.Attribute(“country”), true),
Orders = (
  from o in c.Descendants(“order”)
  select new Order {
IdOrder = (Int32)o.Attribute(“id”),
IdProduct = (Int32)o.Attribute(“idProduct”),
Quantity = (Int32)o.Attribute(“quantity”),
Month = (String)o.Attribute(“month”),
Shipped = (Boolean)o.Attribute(“shipped”)
}
).ToArray()
};
foreach (Customer c in customersWithOrders) {
  Console.WriteLine(c);
foreach (Order o in c.Orders) {
  Console.WriteLine(" {0}", o);
```

```
  }
}
```

Também é possível transformar um documento XML através do LINQ. PIALORSI E RUSSO (2010, p. 401) mostram como é possível transformar atributos em elementos e aplicar uma *namespace*, conforme exemplo abaixo:

```
<?xml version="1.0" encoding="utf-8"?>
<customers>
  <customer name="Paolo" city="Brescia" country="Italy" />
  <customer name="Marco" city="Torino" country="Italy" />
  <customer name="James" city="Dallas" country="USA" />
  <customer name="Frank" city="Seattle" country="USA" />
</customers>
```

Através do seguinte trecho de código é possível transformar o XML acima.

```
XNamespace ns = "http://schemas.devleap.com/Customers";
XElement destinationXmlCustomers =
new XElement(ns + "customers",
new XAttribute(XNamespace.Xmlns + "c", ns),
from c in sourceXmlCustomers.Elements("customer")
where c.Attribute("country").Value == "Italy"
select new XElement(ns + "customer",
new XElement(ns + "name", c.Attribute("name")),
new XElement(ns + "city", c.Attribute("city"))));
```

O XML resultante será:

```
<?xml version="1.0" encoding="utf-8"?>
<c:customers xmlns:c="http://schemas.devleap.com/Customers">
  <c:customer>
    <c:name>Paolo</c:name>
    <c:city>Brescia</c:city>
  </c:customer>
  <c:customer>
    <c:name>Marco</c:name>
    <c:city>Torino</c:city>
  </c:customer>
</c:customers>
```

LINQ também suporta o recurso de serialização. Para PIALORSI E RUSSO (2010, p. 410), o tipo XElement é serializável usando o mecanismo XmlSerializer ou DataContractSerializer. O exemplo abaixo mostra uma serialização:

```
XElement xmlCustomers = new XElement("customers",
from c in listCustomers
select new XElement("customer",
new XElement("name", c.Name),
new XAttribute("city", c.City),
new XAttribute("country", c.Country)));
MemoryStream mem = new MemoryStream();
DataContractSerializer dc = new DataContractSerializer(typeof(XElement));
dc.WriteObject(mem, xmlCustomers);
```

O XML resultante será:

```
<customers><customer city="Brescia"
country="Italy"><name>Paolo</name></customer><customer city="Torino"
country="Italy"><name>Marco</name></customer><customer city="Dallas"
country="USA"><name>James</name></customer><customer city="Seattle"
country="USA"><name>Frank</name></customer></customers>
```

5. Conclusão

Devido a complexidade e ampla capacidade de armazenamento de dados que o documento XML possui, a tarefa de criá-lo, manipulá-lo com rapidez e facilidade de manutenção acaba se tornando complexa se não possuir as ferramentas corretas.

PIALORSI E RUSSO demonstram que as maneiras de se manipular XML via código são muito complexas, de difícil compreensão e requerem mais tempo de codificação por parte do desenvolvedor.

Um dos fatores que mais ajudam o desenvolver do LINQ aplicado a XML é a grande expressividade da linguagem o qual faz a tarefa manipular um XML quase que natural em termos de sintaxe.

Mesmo requerendo um nível de aprendizado e pouca familiarização da sintaxe LINQ por parte do desenvolvedor, o tempo ganho e facilidade de manutenção é rapidamente ganho durante o desenvolvimento.

O autor concluiu que LINQ aplicado a XML é uma solução viável e poderosa para a manipulação total de arquivos XML, sendo muito mais vantajosa do que os velhos métodos e objetos disponíveis atualmente na plataforma .NET .

Referências Bibliográficas

HUNTER, David. **Beginning XML .** Indianapolis: Editora: Wiley Publishing, Inc. , 2007. 1039 p.

KLEIN, Scott. e*t al.* **Professional LINQ .** Indianapolis: Editora: Wiley Publishing, Inc., 2008. 357 p.

PIALORSI, Paolo; RUSSO, Marco. **Programming Microsoft LINQ in Microsoft .NET Framework .** Sebastopol: Editora: O'Reilly Media, Inc., 2010. 675 p.