Capítulo 13
Reflection

Introdução

Reflection (ou Reflexão) é a habilidade que temos em extrair e descobrir informações de metadados de um determinado *Assembly*. Os metadados descrevem os campos (propriedades, membros e eventos) de um tipo juntamente com seus métodos e, durante a compilação, o compilar gerará e armazenará metadados dentro do *Assembly*. São os metadados que permitem uma das maiores façanhas dentro da plataforma .NET, ou seja, escrevermos um componente em Visual C# e consumí-lo em uma aplicação em Visual Basic .NET.

Reflection não permite apenas extrair informações em runtime, mas também permitirá que se carreegar *Assemblies*, instancie classes, invoque seus métodos, etc.. *Reflection* é algo muito poderoso que existe e possibilita dar uma grande flexibilidade para a aplicação. O póprio .NET Framework utiliza Reflection internamente em diversos cenários, como por exemplo o *Garbage Collector* examina os objetos que um determinado objeto referencia para saber se o mesmo está ou não sendo utilizado. Além disso, quando serializamos um objeto, o .NET Framework utiliza *Reflection* para extrair todos os valores do membros internos do objeto para persistí-los. O próprio Visual Studio .NET utiliza informações extraídas via *Reflection* para habilitar o Intellisense e mais, quando está desenvolvendo um formalário e vai até a janela de propriedades de um determinado controle, o Visual Studio .NET extrai os membros do controle via *Reflection* para exibir e, conseqüentemente, alterar de acordo com a necessidade.

A idéia deste capítulo é apresentar como utilizar essa ferramenta poderosa que a Microsoft disponibilizou dentro do .NET Framework para extrair informações de metadados. Todas as classes que utilizaremos para Reflection estão contidas dentro do *namespace* **System.Reflection** e, na primeira parte do capítulo, veremos como é possível carregar Assemblies em memória, para em seguida, conseguir extrair as informações de classes, propriedades, métodos e eventos que um determinado tipo apresenta.

AppDomains

Historicamente, um processo é criado para isolar uma aplicação que está sendo executado dentro do mesmo computador. Cada aplicação é carregada dentro de um processo separado, que isola uma aplicação das outras. Esse isolamento é necessário porque os endereços são relativos à um determinado processo.

O código gerenciado passa por um processo de verificação que deve ser executado antes de rodar. Esse processo determina se o código pode acessar endereço de memória inválidos ou executar alguma outra ação que poderia causar alguma falha no processo. O código que passa por essa verificação é chamado de type-safe e permite ao CLR fornecer um grande nível de isolamento, como um processo, só que com muito mais performance.

AppDomain ou domínio de aplicação é um ambiente isolado, seguro e versátil que o CLR pode utilizar para isolar aplicações. Você pode rodar várias aplicações em um único processo Win32 com o mesmo nível de isolamento que existiria em um processo separado para cada uma dessas aplicações, sem ter o *overhead* que existe quando é necessário fazer uma chamado entre esses processos. Além disso, um *AppDomain* é seguro, já que o CLR garante que um *AppDomain* não conseguirá acessar os dados que estão contidos em outro *AppDomain*. Essa habilidade de poder rodar várias aplicações dentro de um único processo aumenta consideravelmente a escalabilidade do servidor.

Um *AppDomain* é criado para servir de container para uma aplicação gerenciada. A inicialização de um *AppDomain* consiste em algumas tarefas, como por exemplo, a criação da memória *heap*, onde todos os *reference-types* são alocados e de onde o lixo é coletado e a criação de um *pool* de *threads*, que pode ser utilizado por qualquer um dos tipos gerenciados que estão carregados dentro do processo.

O Windows não fornece uma forma de rodar aplicação .NET. Isso é feito a partir de uma **CLR Host**, que é uma aplicação responsável por carregar o CLR dentro de um processo, criando *AppDomains* dentro do mesmo e executando o código das aplicações que desenvolvemos dentro destes *AppDomains*.

Quando uma aplicação é inicializada, um *AppDomain* é criado para ela. Esse *AppDomain* também é conhecido como *default domain* e ele somente será descarregado quando o processo terminar. Sendo assim, o *Assembly* inicial rodará dentro deste *AppDomain* e, se desejar, você pode criar um novos *AppDomains* e carregar dentro destes outros *Assemblies* mas, uma vez carregado, você não pode descarregá-lo e isso somente acontecerá quando a *AppDomain* for descarregada.

O .NET Framework disponibiliza uma classe chamada **AppDomain** que representa e permite manipular *AppDomains*. Essa classe fornece vários métodos (alguns estáticos) que auxiliam desde a criação até o término de um *AppDomain*. Entre esses principais métodos, temos:

Método	**Descrição**
CreateDomain	Método estático que permite a criação de uma nova *AppDomain*.
CurrentDomain	Retorna um objeto do tipo **AppDomain** representando o *AppDomain* da *thread* corrente.
DoCallback	Executa um código em uma outra aplicação a partir de um *delegate*.
GetAssemblies	Retorna um *array* de objetos do tipo **Assembly**, onde cada elemento é um *Assembly* que foi carregado dentro do *AppDomain*.
IsDefaultAppDomain	Retorna uma valor boolano indicando se o *AppDomain* trata-se do *AppDomain* padrão.
Load	Permite carregar um determinado *Assembly* dentro do

	AppDomain.
Unload	Descarrega um determinado *AppDomain*.

Para exemplificar a criação e o descarregamento de um segundo *AppDomain*, vamos analisar o trecho código abaixo:

```
VB.NET
Imports System

Dim _domain As AppDomain = AppDomain.CreateDomain("TempDomain")
'...
AppDomain.Unload(_domain)

C#
using System;

AppDomain _domain = AppDomain.CreateDomain("TempDomain");
//...
AppDomain.Unload(_domain);
```

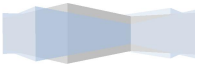
Assemblies

Nomenclatura dos Assemblies

A nomenclatura de um *Assembly* (conhecida como *display name* ou *identidade do Assembly*) consiste em 4 informações: nome (sem ".exe" ou ".dll"), versão, cultura e a *public key token* (que será nula se uma *strong name* não for definida). Para exemplificar isso, vamos analisar o *Assembly System.Data.dll* da versão 2.0 do .NET Framework que é responsável pelas classes de acesso a dados e também um *Assembly* customizado chamado *PeopleLibrary*, onde não foi criado uma *strong name*:

```
System.Data,            Version=2.0.0.0,            Culture=neutral,
PublicKeyToken=b77a5c561934e089

PeopleLibrary,          Version=1.0.0.0,            Culture=neutral,
PublicKeyToken=null
```

O .NET Framework fornece uma classe chamada que descreve a identidade do *Assembly*, chamada **AssemblyName**. Entre as várias propriedades que essa classe possui, podemos destacar as mais importantes: *CodeBase*, *CultureInfo*, *FullName*, *KeyPair*, *Name* e *Version*.



Carregamento manual de Assemblies

Quando referenciamos um *Assembly* qualquer dentro de uma aplicação, o CLR decide quando carregá-lo. Quando você chama um método, o CLR verifica o código IL para ver quais tipos estão sendo referenciados e, finalmente, carrega os Assemblies onde os tais tipos estão sendo referenciados. Se um *Assembly* já estiver contido dentro do *AppDomain*, o CLR é inteligente ao ponto de conseguir identificar e não carregará o mesmo *Assembly* novamente.

Mas quando você quer extrair informações de metadados de um determinado *Assembly*, é necessário carregá-lo para dentro do seu *AppDomain* e, depois disso, poder extrair os tipos que ele expõe. Para que isso seja possível, o namespace **System.Reflection** possui uma classe chamada **Assembly** que possui, além de seus métodos de instância, alguns métodos estáticos que são utilizados para carregar um determinado *Assembly*. Entre esses métodos estáticos para carregamento do *Assembly* temos:

Método	**Descrição**
GetAssembly	Dado um determinado tipo, esse método consegue extrair o *Assembly* em qual esse tipo está contido.
GetCallingAssembly	Retorna um objeto do tipo **Assembly** que representa o *Assembly* onde o método está sendo invocado.
GetEntryAssembly	Retorna um objeto do tipo **Assembly** que está contido no *AppDomain* padrão (*default domain*).
GetExecutingAssembly	Retorna uma instância do *Assembly* onde o código está sendo executado.
Load	Um método com vários *overloads* que retorna um determinado *Assembly*. Um dos *overloads* mais comuns, é o que aceita uma *string*, conteudo o seu *fully qualified name* (nome, versão, cultura e token) como vimos acima. Se o *Assembly* especificado conter com uma *strong name*, o CLR então procura dentro do GAC, seguido pelo diretório base da aplicação e dos diretórios privados da mesma. Agora, se o *Assembly* especificado não conter uma *strong name*, ele apenas não procurará dentro do GAC e, para ambos os casos, se o Assembly não for encontrado, uma exceção do tipo **System.IO.FileNotFoundException**.
LoadFile	Permite carregar um *Assembly* a partir do caminho fiísico até o mesmo, podendo carregar um *Assembly* de qualquer local que ele esteja, não resolvendo as dependências. Esse método é utilizando em cenários mais limitados, onde não é possível a utilização do método *LoadFrom*, já que não permite carregar os *Assemblies* que tenham a mesma identidade, mas estão fisicamente em locais diferentes.
LoadFrom	Carrega um *Assembly* baseando-se no caminho físico,

	resolvendo todas as suas dependências (*ver nota abaixo*).
LoadWithPartialName	Carrega um *Assembly* do diretório da aplicação ou do GAC, mas trata-se de um método obsoleto e, ao invés dele, utilize o método *Load*. Esse método não deve ser utilizado, porque você nunca sabe a versão do Assembly que irá carregar.
ReflectionOnlyLoad	Carrega um *Assembly* em um contexto *reflection-only*, ou seja, o *Assembly* poderá apenas ser examinado, mas não executado.
ReflectionOnlyLoadFrom	Dado o caminho físico até o *Assembly*, o mesmo é carregado dentro do domínio do chamador, também em um contexto *reflection-only*.

Nota Importante: Vamos imaginar que temos uma aplicação Windows Forms (EXE) e uma biblioteca (DLL) que essa aplicação Windows utiliza. A estrutura é a seguinte:

```
C:\Program Files\PeopleUI\WinUI.exe
C:\Program Files\PeopleUI\PeopleLibrary.dll
```

Como podemos ver, a aplicação Windows (WinUI.exe) foi desenvolvida referenciando a biblioteca PeopleLibrary.dll. Imagine que, dentro do método *Main* ele carrega a seguinte DLL: "C:\PeopleLibrary.dll" através do método *LoadFrom* da classe **Assembly**. Imagine que, depois de carregado o mesmo *Assembly*, mas de um outro local, ao invocar um método dentro do PeopleLibrary.dll ele invocará de qual dos dois locais ("C:\" ou "C:\Program Files\PeopleUI\")? Como o CLR não pode supor que os *Assemblies* são iguais somente porque o nome do arquivo coincide, felizmente ele sabe qual deverá executar. Quando o método *LoadFrom* é executado, o CLR extrai informações sobre a identidade do *Assembly* (versão, cultura e *token*), passando essas informações para o método *Load* que, faz a busca pelo *Assembly*. Se um *Assembly* correspondente for encontrado, o CLR fará a comparação do caminho do arquivo especifico no método *LoadFrom* e do caminho encontrado pelo método *Load*. Se os caminhos forem idênticos, o *Assembly* será considerado parte da aplicação, caso contrário, será considerado um "arquivo de dados".

Sempre que possível, é bom evitar o uso do método *LoadFrom* e opte por utilizar o método *Load*. A razão para isso é que, internamente, o método *LoadFrom* invoca o método *Load*. Além disso, o *LoadFrom* trata o *Assembly* como um "arquivo de dados" e, se o CLR carrega dentro de um *AppDomain* o mesmo *Assembly* a partir de caminhos diferentes, uma grande quantidade de memória é desperdiçada.

Exemplo: O código abaixo exemplifica a utilização do método *LoadFrom* da classe **Assembly**. O trecho de código foi extraído de uma demonstração e, você pode consultar na íntegra no seguinte local: XXXXXXXXXXXXXXXXXXXXX.

```
VB.NET
Imports System
Imports System.Reflection

Dim asb As Assembly = Assembly.LoadFrom("C:\Comp.dll")

C#
using System;
using System.Reflection;

Assembly asb = Assembly.LoadFrom("C:\\Comp.dll");
```

Trabalhando com Metadados

Como já vimos anteriormente, os metadados são muito importantes dentro da plataforma .NET. Vimos também como criar *AppDomains* e carregar *Assemblies* dentro deles. Somente carregar os *Assemblies* dentro de um *AppDomain* não tem muitas utilidades.

Uma vez que ele encontra-se carregado, é perfeitamente possível extrair informações de metadados dos tipos que estão contidos dentro *Assembly* e, para isso, utilizaremos várias classes que estão contidas dentro do *namespace* **System.Reflection**. A partir de agora analisaremos essas classes que estão disponíveis para a criação e manipulação de tipos, como por exemplo, invocar métodos, recuperar ou definir valores para propriedades, etc..

Antes de mais nada, precisamos entender qual a hierarquia dos objetos que estão disponíveis para a manipulação dos metadados e para invocá-los dinamicamente. A imagem abaixo exibe tal hierarquia onde, como já era de se esperar, o ancestral comum é o **System.Object**.

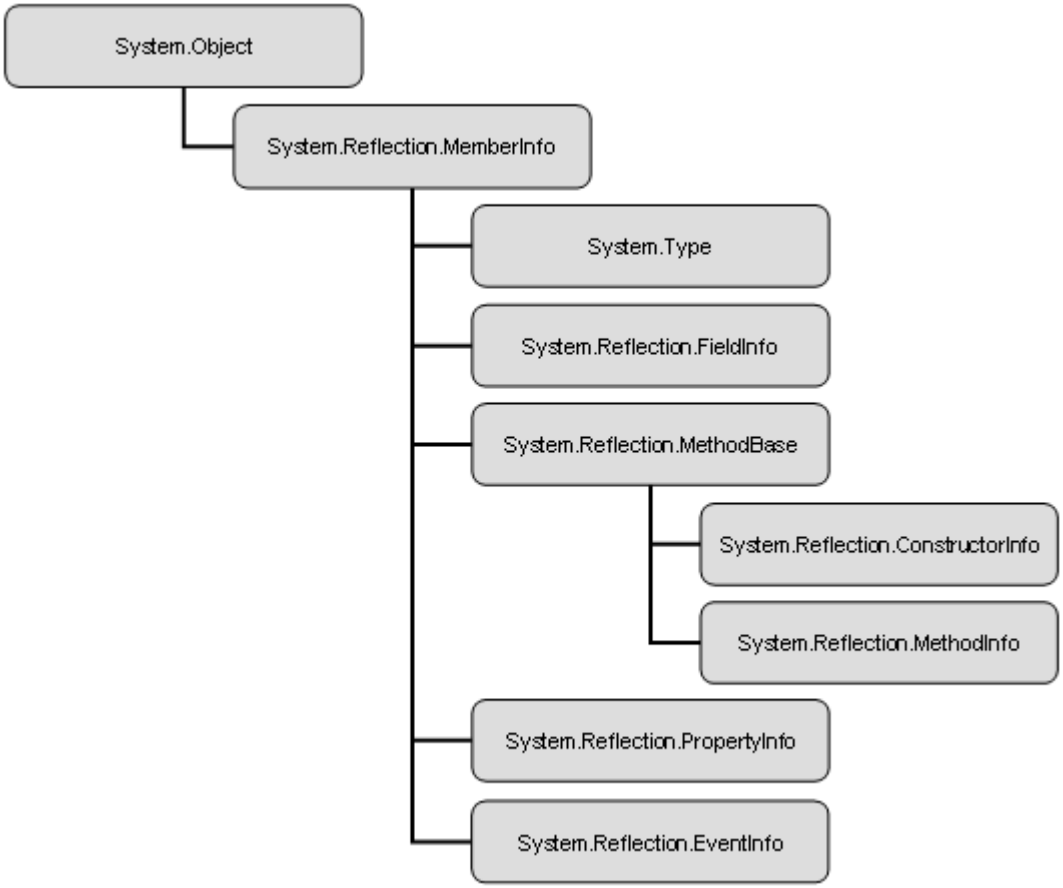


Imagem 15.1 – Hierarquia das classes para manipulação de metadados

System.Reflection.MemberInfo

MemberInfo é uma classe abstrata que é base para todas as classes utilizadas para resgatar informações sobre os membros sejam eles construtores, eventos, campos, métodos ou propriedades de uma determinada classe. Basicamente essa classe fornece as funcionalidades básicas para todos as classes que dela derivarem. Os membros desta classe abstrata são:

Membro	**Descrição**
Name	Retorna uma *string* representando o membro.
MemberType	Propriedade de somente leitura que retorna um item do enumerador **MemberTypes** indicando qual tipo de membro ele é. Entre os itens deste enumerador, temos: • **All** – Especifica todos os tipos • **Constructor** – Especifica que o membro é um construtor, representado pelo tipo **ConstructorInfo**. • **Custom** – Especifica que o membro é um membro customimzado.

	• **Event** – Especifica que o membro é um evento, representado pelo tipo **EventInfo**. • **Field** – Especifica que o membro é um campo, representado pelo tipo **FieldInfo**. • **Method** – Especifica que o membro é um método, representado pelo tipo **MethodInfo**. • **NestedType** – Especifica que o membro é um tipo aninhado, representado pelo tipo **MemberInfo**. • **Property** – Especifica que o membro é uma propriedade, representada pelo tipo **PropertyInfo**. • **TypeInfo** – Especifica que o membro é um tipo, representado pelo tipo **TypeInfo**.
DeclaringType	Propriedade de somente leitura que retorna um objeto do tipo **Type** indicando de qual tipo é o objeto.
ReflectedType	Propriedade de somente leitura que retorna um objeto do tipo **Type** indicando o tipo que foi utilizado para obter a instância deste membro.
GetCustomAttributes	Retorna um *array* contendo algum atributos customizados que o membro contém. Cada um dos elementos é representado por um **System.Object**.
IsDefined	Retorna um valor booleano indicando se existe ou não um determinado atributo especifico aplicado no membro.

System.Type

Essa é uma das mais importantes classes da plataforma .NET. Essa é uma das principais classes utilizadas para você poder extrair informações de um tipo. **System.Object** possui um método chamado *GetType* que retorna o tipo da instância corrente, sendo representado por um objeto do tipo **Type**. Sendo assim, todo e qualquer objeto possui esse método, já que todo tipo herda direta ou indiretamente dessa classe.

O método *GetType* acima utilizamos quando já temos a instância do objeto. Mas e quando não a temos? Para esse caso, a classe **Type** fornece um método estático, também chamado de *GetType* que, dado um tipo (através de uma *string* contendo o **AssemblyQualifiedName**, que inclui o namespace até o nome do tipo que deve ser carregado), ele retorna o objeto *Type* que o representa e, conseqüentemente, conseguirá extrair as informações de metadados do mesmo. Além desse método, a classe **Type** ainda possui um método interessante chamado *GetArrayType*, que retorna um *array* de objetos do tipo **Type**, onde cada elemento representa o tipo correspondente do elemento do *array*. Entre todos os membros da classe **Type**, podemos destacar:

Membro	**Descrição**
Assembly	Propriedade de somente leitura que retorna um objeto do tipo **Assembly** em que o tipo é declarado.
BaseType	Propriedade de somente leitura que retorna um objeto do

	tipo **Type** que representa o tipo qual o tipo corrente foi herdado.
DeclaringType	Propriedade de somente leitura que retorna um objeto do tipo **Type** que representa o tipo do membro.
IsAbstract	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não abstrato.
IsArray	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não um *array*.
IsByRef	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não passado por referência.
IsClass	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não uma classe.
IsEnum	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não um enumerador.
IsGenericParameter	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo representa um *type parameter* de um tipo genérico ou uma definição de método genérico.
IsGenericType	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não um tipo genérico.
IsInterface	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não uma *Interface*.
IsNested	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não aninhado a outro tipo.
IsValueType	Propriedade de somente leitura que retorna um valor booleano indicando se o tipo é ou não um tipo-valor.
FindInterfaces	Retorna um array de objetos do tipo **Type** com todas as Interfaces implementadas ou herdadas pelo tipo.
FindMembers	Retorna um array de objetos do tipo **MemberInfo** com todos os membros de um determinado tipo.
GetConstructor	Método sobrecarregado que procura por um construtor de instância público. A busca é feita baseando-se em um *array* de objetos do tipo **Type** que é passado para esse método, que procurará pelo construtor que atender exatamente esses parâmetros. Se encontrado, uma instância da classe **ConstrutorInfo** é retornada.
GetConstructors	Retorna um *array* de objetos do tipo **ConstructorInfo**, onde cada elemento representa um construtor do tipo.
GetCustomAttributes	Retorna um *array* de objetos do tipo **System.Object**, onde cada elemento representa um atributo que foi aplicado ao membro.
GetDefaultMembers	Procura por membros que aplicam o atributo **DefaultMemberAttribute**. Se encontrado, um *array* de elementos do tipo **MemberInfo** é retornado, onde cada

	elemento representará um membro que aplica o atributo acima especificado. O atributo **DefaultMemberAttribute** define um determinado membro como sendo um membro padrão, que é invocado pelo método *InvokeMember* da classe **Type**.
GetEvent	Dado uma *string* com o nome de um evento existente no tipo, esse método retorna um objeto do tipo **EventInfo** representando o evento.
GetEvents	Retorna um *array* de objetos do tipo **EventInfo**, onde cada elemento representa um evento existente no tipo.
GetField	Dado uma *string* com o nome de um campo existente no tipo, esse método retorna um objeto do tipo **FieldInfo** representando o campo.
GetFields	Retorna um *array* de objetos do tipo **FieldInfo**, onde cada elemento representa um campo público existente no tipo.
GetInterface	Dado uma *string* com o nome de uma *Interface*, ele retornará um objeto do tipo **Type** que represente a mesma, desde que ela esteja implementada no tipo.
GetInterfaces	Retorna um array de objetos do tipo **Type** com todas as Interfaces implementadas ou herdadas pelo tipo.
GetMember	Dado uma *string* com o nome de um membro existente no tipo, esse método retorna um objeto do tipo **MemberInfo** representando o membro.
GetMembers	Retorna um *array* de objetos do tipo **MemberInfo**, onde cada elemento representa um membro (propriedades, métodos, campos e eventos) existente no tipo.
GetMethod	Dado uma *string* com o nome de um método existente no tipo, esse método retorna um objeto do tipo **MethodInfo** representando o método.
GetMethods	Retorna um *array* de objetos do tipo **MethodInfo**, onde cada elemento representa um método existente no tipo.
GetNestedType	Dado uma *string* com o nome de um tipo aninhado existente no tipo, esse método retorna um objeto do tipo **Type** representando o tipo aninhado.
GetNestedTypes	Retorna um *array* de objetos do tipo **Tipo**, onde cada elemento representa um tipo aninhado existente no tipo.
GetProperties	Retorna um *array* de objetos do tipo **PropertyInfo**, onde cada elemento representa uma propriedade existente no tipo.
GetProperty	Dado uma *string* com o nome de uma propriedade existente no tipo, esse método retorna um objeto do tipo **PropertyInfo** representando a propriedade.

Como temos duas formas de extrair o **Type** de algum objeto, o código abaixo exemplifica ambas, aplicando a lógica em cima, a primeira delas é utilizando o método *GetType* a partir da instância da classe e a outra forma é através do método estático *GetType* da classe **Type**.

```
VB.NET
Imports System

Dim id As Integer = 123
Dim forma1 As Type = id.GetType()
Dim forma2 As Type = Type.GetType("System.Int32")

Console.WriteLine(forma1.FullName)
Console.WriteLine(forma2.FullName)

C#
using System;

int id = 123;
Type forma1 = id.GetType();
Type forma2 = Type.GetType("System.Int32");

Console.WriteLine(forma1.FullName);
Console.WriteLine(forma2.FullName);
```

Ambos os códigos exibirão **System.Int32**, que representa o **FullName** do tipo inteiro.

Vimos acima todos os métodos que a classe **Type** fornece. Cada um dos métodos retornam objetos específicos para cada um dos membros que existem dentro de um determinada objeto. A partir daqui, como já somos capazes de extrair o tipo de um objeto, iremos analisar esses objetos específicos, responsáveis por representar os membros do tipo, onde poderemos extrair informações de metadados referentes a cada um deles.

Mas para que podemos entender o grande potencial do *Reflection*, vamos criar uma classe customizada, que possua membros, propriedades, métodos e eventos para que possamos extrair as informações de metadados da mesma. Isso não quer dizer que não seja possível extrair as mesmas informações de uma classe de dentro do .NET Framework. Para fins de exemplo, vamos criar uma classe chamada *Cliente* que será composta por alguns membros que iremos utilizar como base para os futuros exemplos:

```
VB.NET
Public Class Cliente
    Public X As Integer
    Private _id As Integer
    Private _nome As String
```

```
    Public Event AlterouDados As EventHandler

    Public Sub New()
    End Sub

    Public Sub New(ByVal id As Integer, ByVal nome As String)
        Me._id = id
        Me._nome = nome
    End Sub

    Public Property Id() As Integer
        Get
            Return Me._id
        End Get
        Set(ByVal value As Integer)
            Me._id = value
            RaiseEvent AlterouDados(Me, EventArgs.Empty)
        End Set
    End Property

    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(ByVal value As String)
            Me._nome = value
            RaiseEvent AlterouDados(Me, EventArgs.Empty)
        End Set
    End Property

    Public Function ExibeDados() As String
        Dim msg As String = String.Format("{0} - {1}", Id, Nome)
        Return msg
    End Function
End Class
```

C#

```
public class Cliente
{
    public int X;
    private int _id;
    private string _nome;
    public event EventHandler AlterouDados;

    public Cliente() { }

    public Cliente(int id, string nome)
    {
        this._id = id;
        this._nome = nome;
```

```
    }

    public int Id
    {
        get { return _id; }
        set {
            _id = value;
            if    (AlterouDados    !=    null)    AlterouDados(this,
EventArgs.Empty);
        }
    }

    public string Nome
    {
        get { return _nome; }
        set {
            _nome = value;
            if    (AlterouDados    !=    null)    AlterouDados(this,
EventArgs.Empty);
        }
    }

    public string ExibeDados()
    {
        string msg = string.Format("{0} - {1}", Id, Nome);
        return msg;
    }
}
```

Como podemos ver, a classe possui dois membros, um do tipo inteiro e uma *string* e, para cada um deles, temos uma propriedade de escrita/leitura que encapsulam o acesso aos mesmos e um evento que é disparado quando o valor da propriedade é alterado; além disso, há também dois construtores, onde um deles não possui nenhum parâmetro e o outro recebe o numero e nome do cliente; finalmente, temos um método chamado *ExibeDados* que retorna uma *string* com o código e nome do cliente.

Antes de extrairmos as informações relacionados a cada um dos membros internos, precisamos primeiramente recupera o tipo através do método *GetType*, fornecido pela instância da classe *Cliente*, que será armazenado em um objeto chamado *tipo* qual iremos utilizar por todos os exemplos adiante.

VB.NET

```
Dim cliente As New Cliente()
cliente.Id = 123
cliente.Nome = "José Torres"

Dim tipo As Type = cliente.GetType()
```

```
C#
Cliente cliente = new Cliente();
cliente.Id = 123;
cliente.Nome = "José Torres";

Type tipo = cliente.GetType();
```

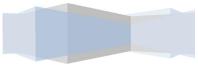
System.Reflection.FieldInfo

Essa classe representa um membro público de um determinado tipo, fornecendo informações de metadata e atributos que possam estar aplicados ao membro. Essa classe não possui um construtor público e, instâncias da mesma, são retornadas a partir dos métodos *GetField* e *GetFields* da classe **Type**. Abaixo é exibido a forma que podemos utilizar e extrair os membros públicos da classe *Cliente* a partir do método *GetFields*:

```
VB.NET
Imports System.Reflection

For Each fi As FieldInfo In tipo.GetFields()
    Console.WriteLine( _
        String.Format("Nome: {0} - Tipo: {1}", _
            fi.Name, fi.FieldType))
Next

C#
using System.Reflection;

foreach (FieldInfo fi in tipo.GetFields())
{
    Console.WriteLine(
        string.Format("Nome: {0} - Tipo: {1}",
            fi.Name, fi.FieldType));
}
```

```
Output
Nome: X - Tipo: System.Int32
```

A propriedade *FieldType* retorna um objeto do tipo **Type** representando o tipo do membro. Além desta propriedades existem algumas outras que merecem ser comentadas, como por exemplo, *IsPublic*, *IsPrivate* e *IsStatic*, que são auto-explicativas.



System.Reflection.MethodBase

A classe **MethodBase** trata-se de uma classe abstrata que fornece informações a respeito de métodos e construtores de um determinado tipo, atributos que são aplicados aos mesmos e métodos e propriedades para manipulação de métodos genéricos. Ela fornece também um método chamado *GetParameters*, que retorna uma coleção de objetos do tipo **ParameterInfo**, onde cada um deles representa um parâmetro que pode existir no método.

A classe **ParameterInfo** possui uma propriedade chamada *ParameterType* que retorna um objeto do tipo **Type** representando o tipo do parâmetro e, além dela, a classe **ParameterInfo** possui algumas outras propriedades úteis para extrairmos informações relacionadas a cada parâmetro e, entre elas, temos a propriedade *IsOut*, que retorna um valor booleano indicando se o parâmetro trata-se de um parâmetro de *output*.

A classe **MethodBase** ainda possui um método, um tanto quanto interessante, chamado *GetMethodBody*. Esse método retorna um objeto do tipo **MethodBody**, contendo o *MSIL stream*, variáveis locais (declaradas dentro do método) e estruturas de exceções.

Finalmente, a classe **MethodBase** classe serve como base para as classes concretas **ConstructorInfo** e **MethodInfo**, que trazem informações customizadas para cada um dos tipos.

O primeiro deles, **ConstructorInfo**, trata-se de uma classe que representa um determinado construtor público de um tipo, fornecendo informações de metadados à respeito do construtor e também descobrindo os atributos que o construtor pode ter. Essa classe não possui um construtor público e, instâncias da mesma, são retornadas a partir dos métodos *GetConstructor* e *GetConstructos* da classe **Type**. O código abaixo exemplifica a utilização da classe **ConstructorInfo**, extraindo os construtores da classe *Cliente*:

VB.NET
```
Imports System.Reflection

For Each ci As ConstructorInfo In tipo.GetConstructors()
    Console.WriteLine(String.Format("Construtor: {0}", ci.Name))
    For Each pi As ParameterInfo In ci.GetParameters()
        Console.WriteLine( _
            String.Format("   Parâmetro: {0} - Tipo: {1}", _
                pi.Name, pi.ParameterType))
    Next
Next
```

C#
```
using System.Reflection;

foreach (ConstructorInfo ci in tipo.GetConstructors())
```

```
{
    Console.WriteLine(String.Format("Construtor: {0}", ci.Name));
    foreach (ParameterInfo pi in ci.GetParameters())
    {
        Console.WriteLine(
            string.Format("   Parâmetro: {0} - Tipo: {1}",
                pi.Name, pi.ParameterType));
    }
}
```

```
Output
Construtor: .ctor
Construtor: .ctor
   Parâmetro: id - Tipo: System.Int32
   Parâmetro: nome - Tipo: System.String
```

Como podemos ver, temos um laço *For* aninhado que é utilizado para recuperar os parâmetros de um determinado construtor a partir de uma instância de um objeto **ConstructorInfo**.

A segunda e última classe que deriva de **MethodBase**, a classe **MethodInfo**, tem um comportamento muito parecido com a classe **ConstructorInfo**, só que neste caso, cada objeto deste representa um método público de um tipo. A classe **MethodInfo** também pode receber parâmetros e ter atributos. A única exceção é que a classe **MethodInfo** pode ter um tipo de retorno, ou seja, uma função que retorna um valor qualquer e, para isso, a classe MethodInfo fornece uma propriedade chamado *ReturnType* que retorna um objeto do tipo **Type** representando o tipo que o método retorna. Essa classe não possui um construtor público e, instâncias da mesma, são retornadas a partir dos métodos *GetMethod* e *GetMethods* da classe **Type**.

Através do código abaixo podemos visualizar a utilização da classe **MethodInfo**. Vale lembrar que todas as classes herdam direta ou indiretamente da classe **System.Object** e, conseqüentemente, todos os métodos públicos lá contidos, como por exemplo, *GetType*, *Equals*, *ToString* etc. são também exibidos.

```
VB.NET
Imports System.Reflection

For Each mi As MethodInfo In tipo.GetMethods()
    Console.WriteLine( _
        String.Format("Método: {0} - Retorno: {1}", _
            mi.Name, mi.ReturnType))

    For Each pi As ParameterInfo In mi.GetParameters()
```

```
        Console.WriteLine( _
            String.Format("   Parametro: {0} - Tipo: {1}", _
                pi.Name, pi.ParameterType))
    Next
Next
```

C#

```
using System.Reflection;

foreach (MethodInfo mi in tipo.GetMethods())
{
    Console.WriteLine(
        String.Format("Método: {0} - Retorno: {1}",
            mi.Name, mi.ReturnType));

    foreach (ParameterInfo pi in mi.GetParameters())
    {
        Console.WriteLine(
            string.Format("   Parâmetro: {0} - Tipo: {1}",
                pi.Name, pi.ParameterType));
    }
}
```

Output

```
Método: add_AlterouDados - Retorno: System.Void
   Parâmetro: value - Tipo: System.EventHandler
Método: remove_AlterouDados - Retorno: System.Void
   Parâmetro: value - Tipo: System.EventHandler
Método: get_Id - Retorno: System.Int32
Método: set_Id - Retorno: System.Void
   Parâmetro: value - Tipo: System.Int32
Método: get_Nome - Retorno: System.String
Método: set_Nome - Retorno: System.Void
   Parâmetro: value - Tipo: System.String
Método: ExibeDados - Retorno: System.String
Método: GetType - Retorno: System.Type
Método: ToString - Retorno: System.String
Método: Equals - Retorno: System.Boolean
   Parâmetro: obj - Tipo: System.Object
Método: GetHashCode - Retorno: System.Int32
```

System.Reflection.PropertyInfo

17

Essa classe representa uma propriedade pública de um determinado tipo, fornecendo informações de metadata e atributos que possam estar aplicados à propriedade. Essa classe não possui um construtor público e, instâncias da mesma, são retornadas a partir

dos métodos *GetProperty* e *GetProperties* da classe **Type**. A classe **PropertyInfo** possui uma propriedade chamada *PropertyType*, que retorna um objeto do tipo **Type** representando o tipo da propriedade e ainda, possui duas propriedades chamadas *CanRead* e *CanWrite*, que retornam um valor booleano indicando se a propriedade pode ser lida e escrita, respectivamente. Abaixo é exibido a forma que podemos utilizar e extrair os membros públicos da classe *Cliente* a partir do método *GetProperties*:

```
VB.NET
Imports System.Reflection

For Each pi As PropertyInfo In tipo.GetProperties()
    Console.WriteLine( _
        String.Format("Propriedade: {0} - Tipo: {1}", _
            pi.Name, pi.PropertyType))
Next

C#
using System.Reflection;

foreach (PropertyInfo pi in tipo.GetProperties())
{
    Console.WriteLine(
        string.Format("Propriedade: {0} - Tipo: {1}",
            pi.Name, pi.PropertyType));
}
```

```
Output
Propriedade: Id - Tipo: System.Int32
Propriedade: Nome - Tipo: System.String
```

System.Reflection.EventInfo

Essa classe representa um evento público de um determinado tipo, fornecendo informações de metadata e atributos que possam estar aplicados ao evento. Essa classe não possui um construtor público e, instâncias da mesma, são retornadas a partir dos métodos *GetEvent* e *GetEvents* da classe **Type**. A classe **EventInfo** possui uma propriedade chamada *EventHandlerType*, que retorna um objeto do tipo **Type** representando o tipo do *delegate* utilizado para declarar o evento. Abaixo é exibido a forma que podemos utilizar e extrair os membros públicos da classe *Cliente* a partir do método *GetEvents*:


```
VB.NET
Imports System.Reflection
```

```
For Each ei As EventInfo In tipo.GetEvents()
    Console.WriteLine( _
        String.Format("Evento: {0} - Tipo: {1}", _
            ei.Name, ei.EventHandlerType))
Next
```

C#

```
using System.Reflection;

foreach (EventInfo ei in tipo.GetEvents())
{
    Console.WriteLine(
        string.Format("Evento: {0} - Tipo: {1}",
            ei.Name, ei.EventHandlerType));
}
```

Output

```
Evento: AlterouDados - Tipo: System.EventHandler
```

Invocando os membros em runtime

Agora que já sabemos como extrair as informações dos tipos, como podemos fazer para invocar esses membros automaticamente? Mas para fazermos isso, é necessário entendermos o conceito de *Binding*.

Binding é o processo de localizar a implementação de um determinado tipo e existem dois tipos: *Early Binding* e *Late Binding*. *O Early Binding* ocorre quando você declara uma variável já especificando um tipo ao invés de um **System.Object** (que é a base), fortemente tipando e já tendo conhecimento do objeto em tempo de desenvolvimento e, além disso, você terá a checagem de tipos e conversões sendo feitas em *compile-time*, ou seja, não precisará esperar a aplicação ser executada para detectar possíveis problemas. Isso também irá permitir que o compilador trabalhe de forma mais eficiente, fazendo otimizações antes da aplicação efetivamente ser executada.

Já o *Late Binding* é o processo inverso, ou seja, você somente irá conhecer o tipo em *runtime*. O *Late Binding* é menos performático que o *Early Binding*, mas te dará uma maior flexibilidade, flexibilidade qual é necessária quando trabalhamos com *Reflection* para criar e instanciar os membros de um determinado tipo. É justamente esse tipo de *binding* que vamos utilizar aqui.

Antes de executar qualquer método ou propriedade, temos primeiramente que instanciar a classe em runtime e, para isso, existem duas formas. A primeira delas é utilizando o método de instância chamado *CreateInstance* da classe **Assembly** ou o método estático,

também chamado *CreateInstance*, da classe **Activator**. A diferença entre eles é que, no caso da classe **Assembly**, o tipo informado será procurado dentro do *Assembly* que a instância da classe **Assembly** está referenciado; já no caso da classe **Activator**, recebe o nome (identidade) de um *Assembly* de qual ela efetivamente deverá criar a instância da classe. Internamente, o método *CreateInstance* da classe **Assembly**, invoca o método *CreateInstance* da classe **Activator**. Ambos os métodos retornam um objeto do tipo **System.Object** com a instância da classe especifica criada.

Ainda utilizando o exemplo da classe *Cliente* que construímos acima, vamos analisar como devemos procede para criar a instância da mesma em runtime. A classe Cliente está agora em um outro *Assembly* (uma DLL) em local físico diferente. Para fins de exemplo, a instância será criada a partir do método *CreateInstance* da instância da classe **Assembly**, qual foi criada com o retorno do método estático *LoadFrom*, também da classe **Assembly**, qual analisamos anteriormente. O código abaixo exemplifica como instanciar a classe *Cliente*:

```
VB.NET
Imports System.Reflection

Dim asb As Assembly = Assembly.LoadFrom("C:\PeopleLibrary.dll")
Dim cliente As Object = asb.CreateInstance( _
                            "PeopleLibrary.Cliente", _
                            False, _
                            BindingFlags.CreateInstance, _
                            Nothing, _
                            Nothing, _
                            Nothing, _
                            Nothing)

C#
using System.Reflection;

Assembly asb = Assembly.LoadFrom(@"C:\PeopleLibrary.dll");
object cliente = asb.CreateInstance(
                     "PeopleLibrary.Cliente",
                     false,
                     BindingFlags.CreateInstance,
                     null,
                     null,
                     null,
                     null);
```

O método *CreateInstance* da classe **Assembly** é sobrecarregado e, em um dos seus *overloads*, é possível passar vários parâmetros para que o método cria a instância do objeto. Entre esses parâmetros, temos:

Parâmetro	Descrição
typeName	Uma *string* representando o tipo que deverá ser instanciado.
ignoreCase	Um valor booleano indicando se a busca pelo tipo deverá ou não ser *case-sensitive*.
bindingAttr	Uma combinação de valores disponibilizados no enumerador **BindingFlags** que afetará na forma que a busca pelo tipo será efetuada. Entre os valores disponibilizados por esse enumerador, temos os principais deles descritos abaixo: • **CreateInstance** – Especifica que o Reflection deverá criar uma instância do tipo especificado e chama o construtor que se enquandra com o array de System.Objects que pode ser passado para o método CreateInstance. • **DeclaredOnly** – Somente membros declarados no mesmo nível da hierarquia do tipo será considerado na busca por membros e tipos. • **Default** – Especifica o flag de *no-binding*. • **ExactBinding** – Esta opção especifica que os tipos dos argumentos fornecidos devem exatamente coincidir com os tipos correspondentes dos parâmetros. Uma *Exception* é lançada se o chamador informar um *Binder*. • **FlattenHierarchy** – Especifica que membros estáticos públicos e protegidos devem ser retornados. Membros estáticos privados não são retornados. Membros estáticos incluem campos, métodos, eventos e propriedades. Tipos aninhados não são retornados. • **GetField** – Espefica que o valor de um campo especifico deve ser retornado. • **GetProperty** –Especifica que o valor de uma propriedade específica deve ser retornada. • **IgnoreCase** – Especifica que o *case-sensitive* deve ser considerado quando efetuar o *binding*. • **IgnoreReturn** – Usada em interoperabilidade com COM, ignora o valor de retorno do método. • **Instance** – Especifica que membros de instância são incluídos na pesquisa do membro. • **InvokeMethod** – Especifica que o método será invocado. • **NonPublic** – Especifica que membros não-públicos serão incluídos na pesquisa do membro. • **OptionalParamBinding** – Esta opção é utilizada quando o método possui parâmetros *default*. • **Public** – Especifica que membros públicos serão incluídos na pesquisa de membros e tipos. • **SetField** – Este valor especifica que o valor de um membro

	específico deve ser definido. • **SetProperty** – Espeifica que o valor de uma propriedade específica deve ser definido. • **Static** – Especifica que membros estáticos serão incluídos na pesquisa do membro.
binder	Um objeto que habilita o binding, coerção de tipos do argumentos, invocação dos membros e recuperar a instância de objetos **MemberInfo** via *Reflection*. Se nenhum *binder* for informado, um *binder* padrão é utilizado.
args	Um *array* de **System.Object** contendo os argumentos que devem ser passados para o construtor. Se o construtor for sobrecarregado, a escolha do qual utilizar será feito a partir do número de tipo dos parâmetros informados quando invocar o objeto.
culture	Uma instância da classe **CultureInfo** que é utilizada para a coerção de tipos. Se uma referência nula for passado para este parâmetro, a cultura da *thread* corrente será utilizada.
activationAttributes	Um *array* de elementos do tipo **System.Object** contendo um ou mais atributos que podem ser utilizados durante a ativação do objeto.

Se adicionarmos um construtor na classe *Cliente*, então devemos passar ao parâmetro *args* um *array* com os valores para satisfazer o *overload*, onde é necessário estar com o mesmo número de parâmetros, ordem e tipos. Para fins de exemplo, vamos, através do código abaixo, invocar a classe *Cliente* passando os objetos para o construtor que aceita um inteiro e uma *string*. O código muda ligeiramente:

```
VB.NET
Imports System.Reflection

Dim asb As Assembly = Assembly.LoadFrom("C:\PeopleLibrary.dll")
Dim constrArgs() As Object = {123, "José Torres"}
Dim cliente As Object = asb.CreateInstance( _
                            "PeopleLibrary.Cliente", _
                            False, _
                            BindingFlags.CreateInstance, _
                            Nothing, _
                            constrArgs, _
                            Nothing, _
                            Nothing)

C#
using System.Reflection;

Assembly asb = Assembly.LoadFrom(@"C:\PeopleLibrary.dll");
object[] constrArgs = { 123, "José Torres" };
object cliente = asb.CreateInstance(
```

```
                    "PeopleLibrary.Cliente",
                    false,
                    BindingFlags.CreateInstance,
                    null,
                    constrArgs,
                    null,
                    null);
```

O método *CreateInstance* retorna um **System.Object** representando a instância da classe informada. Se caso o tipo não for encontrado, o método retorna uma valor nulo e, sendo assim, é necessário testar essa condição para não deixar a aplicação falhar. Em seguida, depois do objeto devidamente instanciado, vamos analisar como fazer para invocar os métodos e propriedades que o objeto possui. Para que possamos invocar os tipos do objeto, é necessário extrairmos o **Type** do mesmo como já vimos acima e, dando seqüência ao exemplo, as próximas linhas de código ilustram como extrair o objeto **Type**, através do método *GetType* da variável *cliente*:

```
VB.NET
Dim tipo As Type = cliente.GetType()

C#
Type tipo = cliente.GetType();
```

Essa classe tem um método denominado *ExibeDados*, que retorna uma *string* com o *Id* e o *Nome* do cliente concatenados. Como definimos no construtor da classe os valores 123 e "José Torres", ao invocar o método *ExibeDados*, esses valores deverão ser exibidos. Para que seja possível invocar o método em *runtime*, necessitamos utilizar o método *InvokeMember* da classe *Type*, que retorna um **System.Object** com o valor retornado pelo método. O exemplo abaixo ilustra como utilizá-lo:

```
VB.NET
Console.WriteLine(tipo.InvokeMember( _
    "ExibeDados", _
    BindingFlags.InvokeMethod, _
    Nothing, _
    cliente, _
    Nothing))

C#
Console.WriteLine(tipo.InvokeMember(
                    "ExibeDados",
                    BindingFlags.InvokeMethod,
                    null,
                    cliente,
```

```
                        null));
```

Entre os parâmetros que esse método utiliza, temos (na ordem em que aparecem no código acima): uma *string* contendo o nome do construtor, método, propriedade ou campo a ser invocado (se informar uma *string* vazia, o membro padrão será invocado); uma combinação das opções fornecidas pelo enumerador **BindingFlags** (detalhado mais acima) indicando como a busca pelo membro será efetuada; *binder* a ser utilizado para a pesquisa do membro; a instância do objeto de onde o método será pesquisado e executado e, finalmente, um *array* de elementos do tipo **System.Object**, contendo os parâmetros necessários para ser passado para o método a ser executado.

Além do método, ainda há a possibilidade de invocarmos propriedades em *runtime*. Podemos além de ler as informações de cada uma delas, podemos definir os valores a elas. Para isso, utilizamos o método *GetProperty* da classe **Type**, que retorna uma instância da classe **PropertyInfo**, que representa a propriedade e, através dela, definimos os valores e extraimos para escrevê-los, assim como é mostrado no trecho de código abaixo:

```
VB.NET
Dim nome As PropertyInfo = tipo.GetProperty("Nome")
Dim id As PropertyInfo = tipo.GetProperty("Id")

nome.SetValue(cliente, "Mario Oliveira", Nothing)
id.SetValue(cliente, 456, Nothing)

Console.WriteLine(nome.GetValue(cliente, Nothing))
Console.WriteLine(id.GetValue(cliente, Nothing))

C#
PropertyInfo nome = tipo.GetProperty("Nome");
PropertyInfo id = tipo.GetProperty("Id");

nome.SetValue(cliente, "Mario Oliveira", null);
id.SetValue(cliente, 456, null);

Console.WriteLine(nome.GetValue(cliente, null));
Console.WriteLine(id.GetValue(cliente, null));
```

Basicamente, quando chamamos o método *SetValue*, passamos a instância do objeto onde as propriedades serão manipuladas; o novo valor a ser definido para a propriedade e, finalmente, um objeto do tipo **System.Object**, quando a propriedade se tratar de uma propriedade indexada. Já o método *GetValue* é quase idêntico, apenas não temos o valor a ser definido, pois como o próprio nome do método diz, ele é utilizado para ler o conteúdo da propriedade.

Finalmente, se quisermos extrair os eventos que a classe possui e vincularmos dinamicamente para que o mesmo seja disparado, podemos fazer isso através do método *AddEventHandler* fornecido pelo classe **EventInfo**. Como sabemos, a classe *Cliente* fornece um evento chamado *AlterouDados*, qual podemos utilizar para que quando um valor for alterado em uma das propriedades, esse evento seja disparado. O código abaixo ilustra como configurar para vincular dinamicamente o evento:

```
VB.NET
Dim ev As EventInfo = tipo.GetEvent("AlterouDados")
ev.AddEventHandler(cliente, New EventHandler(AddressOf Teste))

'...

Private Sub Teste(ByVal sender As Object, ByVal e As EventArgs)
    Console.WriteLine("Alterou...")
End Sub

C#
EventInfo ev = tipo.GetEvent("AlterouDados");
ev.AddEventHandler(cliente, new EventHandler(Teste));

//...

private void Teste(object sender, EventArgs e)
{
    Console.WriteLine("Alterou...");
}
```

Criação dinâmica de Assemblies

Há um *namespace* dentro de **System.Reflection** chamado de **System.Reflection.Emit**. Dentro deste *namespace* existem várias classes que são utilizadas para criarmos dinamicamente um *Assembly* e seus respectivos tipos.

Essas classes são também conhecidas como *builder classes*, ou seja, para cada um dos membros que vimos anteriormente, como por exemplo, *Assembly, Type*, *Constructor*, *Event*, *Property*, etc., existem uma classe correspodente com um sufixo em seu nome, chamado *XXXBuilder*, indicando que é um construtor de um dos itens citados. Para a criação de um *Assembly* dinâmico, temos as seguintes classes:

Classe	Descrição
ModuleBuilder	Cria e representa um módulo.
EnumBuilder	Representa um enumerador.
TypeBuilder	Fornece um conjunto de rotinas que são utilizados para

	criar classes, podendo adicionar métodos e campos.
ConstructorBuilder	Define um construtor para uma classe.
EventBuilder	Define um evento para uma classe.
FieldBuilder	Define um campo.
PropertyBuilder	Define uma propriedade para uma determinada classe.
MethodBuilder	Define um método para uma classe.
ParameterBuilder	Define um parâmetro.
GenericTypeParameterBuilder	Define um parâmetro genérico para classes e métodos.
LocalBuilder	Cria uma variável dentro de um método ou construtor.
ILGenerator	Gera código MSIL (Microsoft Intermediate Language).