Capítulo 3
Utilização de Assemblies

Introdução

Até o momento, escrevemos o código dentro do Visual Studio .NET e, pressionando F5 ou Ctrl + F5 (*Build*) rodamos a aplicação para ela ser executada e, conseqüentemente, testamos para ver se o resultado é o esperado. Mas e nos bastidores, o que realmente acontece?

A finalidade deste capítulo é justamente abordar o processo que é feito quando "rodamos" a aplicação. O que acontece por detrás disso é a compilação do código escrito em um *Assembly*. Além de entedermos o processo de geração do Assembly, vamos analisar a forma que temos para "assinar" o *Assembly* e, assim, garantirmos uma maior segurança e unicidade. Ainda nesta primeira parte, analisaremos os tipos de *Assemblies* e também as formas que temos para distribuí-los; como embutir arquivos de recursos dentro de um *Assembly* e como acessá-lo.

Já na segunda parte do capítulo, será abordado as *Installer Classes*, quais fornecem uma possibilidade de automatizamos a instalação dos *Assemblies* nos clientes. Finalmente, veremos alguns funcionalidades disponíveis para criarmos seções de configurações customizadas dentro do arquivo de configuração da aplicação. Isso irá permitir termos uma aplicação mais flexível e bem mais intuitiva para os clientes que irão instalá-la.

O que são Assemblies?

No primeiro capítulo analisamos o **CLR – Common Language Runtime** que, como o próprio nome diz, é um runtime comum para todas as aplicações que utilizam o .NET Framework. Sendo assim, o CLR não conhece nada sobre a linguagem de programação que você escolheu para melhor expressar suas intenções. Quando você executa a aplicação dentro do Visual Studio .NET, o compilador da linguagem escolhida é encarregado verificação da sintaxe e da "escrita correta", analisando o seu código fonte e certificando do que aquilo que escreveu faz algum sentido. Entre os vários compiladores disponíveis, temos os mais comuns: **vbc.exe** (Visual Basic .NET) e **csc.exe** (Visual C#).

Para entendermos melhor o processo de compilação e criação do *Assembly*, vamos utilizar neste momento um utilitário de linha de comando para isso, deixando temporariamente, o Visual Studio .NET de lado. Sendo assim, você pode criar um arquivo de código (Visual Basic .NET ou Visual C#) no *Notepad*. Como não teremos o auxílio do Visual Studio .NET neste momento, deveremos utilizar o compilador correspondente da linguagem para efetuarmos a compilação. Independente de qual linguagem e qual compilador está utilizando, o resultado da compilação de um único arquivo será sempre o mesmo: um **módulo gerenciado**. Um módulo gerenciado é um *executável portável* (PE), que necessita obrigatoriamente do CLR para poder funcionar. A figura abaixo ilusta este processo:

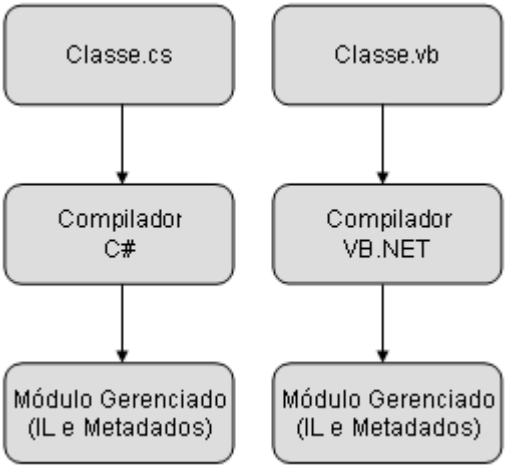


Imagem 3.1 – Criação de um módulo gerenciado.

Dentro da plataforma Microsoft .NET, um *Assembly* é um código parcialmente compilado, tratando-se de um agrupamento lógico de um ou mais módulos gerenciados ou arquivos de recursos. Além disso, um *Assembly* é a menor unidade de reutilização, segurança e controle de versão. Dizemos parcialmente compilado, porque o código está em uma linguagem intermediária, chamada de ***MSIL***: ***Microsoft Intermediate Language***.

Existem dois tipos de *Assembly*: *EXE (Executable)* e *DLL (Dynamic Link Library)*. O primeiro deles é gerado quando uma aplicação do tipo *Windows Forms*, *Windows Service* ou *Console* são criadas; já o segundo, a *DLL*, trata-se de uma biblioteca de tipos que podem ser utilizados por várias outras aplicações. Dentro de cada um destes tipos de Assemblies, ainda tipos um sub-tipo que são: *single-file assemblies* e *multi-file assemblies*.

O primeiro deles, *single-file assemblies*, são *Assemblies* que contém apenas um módulo gerenciado dentro dele, qual fará todo o trabalho necessário para a aplicação ou biblioteca funcionar. Já o segundo, *multi-file assemblies*, são compostos por mais de um módulo gerenciado, quais são colocados dentro deste *Assembly*. Vale lembrar que ainda podemos adicionar arquivos de recursos, como por exemplo: *.jpg, *.ico, *.txt, etc.. Veremos mais detalhadamente sobre arquivos de recurso ainda neste capítulo.

Dentro dos *Assemblies* também temos o manifesto. O manifesto contém todas as informações sobre os itens que estão contidos dentro do *Assembly*, incluindo tudo o que ele expõe ao mundo. Ele também informa todas as dependências existentes no seu *Assembly* para com outros *Assemblies*. Todo *Assembly* requer um manifesto.

Criação de Assemblies

Para efeitos de exemplo, vamos criar dois tipos de Assemblies: *single-file assemblies* e *multi-file assemblies*. A começar pelo *single-file*, devemos criar um arquivo distinto para cada linguagem, um para Visual Basic .NET e outro para Visual C#. O exemplo abaixo

exibe a criação destes códigos utilizando Notepad. Cada um deles são salva em um diretório temporário chamado **Temp** dentro da unidade **C** com suas respectivas extensões.

VB.NET

```
Imports System

Public Class BoasVindas

    Public Shared Sub Main()
        Console.WriteLine("Seja bem-vindo pelo VB.NET.")
        Console.ReadLine()
    End Sub

End Class
```

C#

```
using System;

public class BoasVindas
{
    public static void Main()
    {
        Console.WriteLine("Seja bem-vindo pelo Visual C#.");
        Console.ReadLine();
    }
}
```

Como pode reparar, trata-se de um arquivo simples. Para recapitular, cada um das linguagens tem um compilador próprio. No caso do Visual C#, o compilador é **csc.exe** e do Visual Basic .NET é o **vbc.exe**. Ambas estão contidos dentro do seguinte diretório: **C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727**. Como esses utilitários são executados a partir de linha de comando e você optar por abrir o *Visual Studio 2005 Command Prompt*, não será necessário digitar o caminho todo até o executável para executá-lo. Abaixo está a chamada para o compilador, passando como parâmetro o arquivo a ser compilado e, em seguida, o output de cada compilador:

VB.NET

```
C:\Temp>vbc BoasVindasVB.vb
Microsoft (R) Visual Basic Compiler version 8.0.50727.42
for Microsoft (R) .NET Framework version 2.0.50727.42
Copyright (c) Microsoft Corporation.  All rights reserved.
```

C#

```
C:\Temp>csc BoasVindasCS.cs
```

```
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights
reserved.
```

Com isso, finalizamos o processo de um *single-file assembly*. Criamos então dois executáveis: *BoasVindasVB.exe* e *BoasVindasCS.exe*. Agora, a idéia é termos mais de um arquivo (módulo gerenciado) e unir todos estes em um *multi-file assembly*. Para isso, devemos então criar dois arquivos de código. O primeiro trata-se de uma classe que será consumida por um outro código, em uma outra classe, em um arquivo distinto. O código abaixo mostra o arquivo *Aluno.cs* e *Aluno.vb*. Já os códigos que estão logo na seqüencia, utilizam as classes *Aluno.cs* e *Aluno.vb*:

VB.NET
```
Imports System

Public Class Aluno

    Public Sub Show(ByVal nome As String)
        Console.WriteLine("Seja Bem-vindo: " & nome)
    End Sub

End Class
```

C#
```
using System;

public class Aluno
{
    public void Show(string nome)
    {
        Console.WriteLine("Seja Bem-vindo: " + nome);
    }
}
```

VB.NET
```
Imports System

Public Class Escola

    Public Shared Sub Main()
        Dim aluno As New Aluno()
        aluno.Show("José Castro")
    End Sub
```

```
End Class

C#
using System;

public class Escola
{
    public static void Main()
    {
        Aluno aluno = new Aluno();
        aluno.Show("José Castro");
    }
}
```

Mais uma vez, utilizaremos os compiladores de cada linguagem para gerar o executável. Só que agora temos uma ligeira mudança em relação ao *single-file assembly*. Para cada um dos arquivos de código que temos, necessitamos criar um módulo gerenciado e, para isso, utilizamos a mesma técnica que o *single-file assembly*. Agora, quando referenciamos esta classe em um outro módulo gerenciado, é necessário primeiro adicionarmos o módulo pré-criado ao Assembly que será gerado. Os compiladores fornecem um parâmetro chamado addmodule que permitem adicionarmos a referência para o(s) módulo(s) que será(ão) utilizado(s). A sintaxe para isso é exibida através do *output* que o compilador gerou ao efetuar o processo:

```
VB.NET
C:\Temp>vbc /t:module Aluno.vb
Microsoft (R) Visual Basic Compiler version 8.0.50727.42
for Microsoft (R) .NET Framework version 2.0.50727.42
Copyright (c) Microsoft Corporation.  All rights reserved.

C:\Temp>vbc /addmodule:Aluno.netmodule /out:Escola.exe Escola.vb
Microsoft (R) Visual Basic Compiler version 8.0.50727.42
for Microsoft (R) .NET Framework version 2.0.50727.42
Copyright (c) Microsoft Corporation.  All rights reserved.

C#
C:\Temp>csc /t:module Aluno.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights
reserved.

C:\Temp>csc /addmodule:Aluno.netmodule /out:Escola.exe Escola.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights
reserved.
```

Claro que todo este processo está manual justamente por questões de entendimento. Quando você utiliza uma IDE, como por exemplo o Visual Studio .NET, tudo isso é feito de forma transparente, gerando apenas o EXE ou a DLL.

O arquivo AssemblyInfo

Este arquivo é criado por padrão nas maiorias das aplicações e é através dele que podemos colocar várias informações a nível de *Assembly*. Através de vários atributos fornecidos pelo .NET Framework, podemos definir várias informações como por exemplo: nome do produto, versão, empresa, descrição, cultura, segurança, etc.. Quando você cria uma aplicação utilizando o Visual Studio .NET, esse arquivo já é gerado automaticamente e você pode abrí-lo e configurar de acordo com a sua necessidade.

O código abaixo mostra um arquivo AssemblyInfo padrão:

VB.NET

```
Imports System
Imports System.Reflection
Imports System.Runtime.InteropServices

<Assembly: AssemblyTitle("VB")>
<Assembly: AssemblyDescription("Exemplos em VB.NET.")>
<Assembly: AssemblyCompany("People Computação")>
<Assembly: AssemblyProduct("VB")>
<Assembly: AssemblyCopyright("Copyright ©  2007")>
<Assembly: AssemblyTrademark("")>
<Assembly: ComVisible(False)>
<Assembly: Guid("4db1f6e9-653c-479f-ab0a-0c713b7d7822")>
<Assembly: AssemblyVersion("1.0.0.0")>
<Assembly: AssemblyFileVersion("1.0.0.0")>
```

C#

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("CS")]
[assembly: AssemblyDescription("Exemplos em C#.")]
[assembly: AssemblyCompany("People Computação")]
[assembly: AssemblyProduct("CS")]
[assembly: AssemblyCopyright("Copyright ©  2007")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: Guid("653551f4-88ef-4c39-bcd9-0a00e39d4e42")]
```

```
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Strong Names

Imaginem que temos um *Assembly* chamado *CalculosDiversos.dll* criado pela empresa *ABC Dev Company*. Esse *Assembly* contém uma porção de tipos que são utilizados para os mais diversos cálculos. Esse *Assembly* será, muito provavelmente, por várias aplicações.

Agora, temos um diretório conhecido que serve como repositório para os *Assemblies* que são utilizados pelas mais diversas aplicações e, um deles, é o *CalculosDiversos.dll*. Bem, neste momento, temos uma outra empresa que decide criar um mesmo *Assembly*, com o mesmo nome de arquivo e, se colocado dentro do diretório que é o nosso repositório de *Assemblies*, as aplicações que fazem o uso deste *Assembly* deixará de funcionar, pois a DLL foi sobrescrita e, como a última vence, estamos novamente com o inferno das DLLs em plena era .NET.

Como é possível notar, diferenciar um *Assembly* apenas pelo nome de arquivo não é necessário para garantir a unicidade. Felizmente o .NET fornece uma forma de identificarmos o *Assembly* como sendo único e, para isso, utilizamos **strong names**. As informações que compõem um *Assembly* que é assinado por uma *strong name* consiste em quatro atributos, que o identificarão unicamente: o nome do arquivo (sem extensão), o número de versão, a cultura e um *token* de chave pública.

Como o nome, cultura e a versão do *Assembly* podem repetir de uma empresa para outra, a Microsoft decidiu utilizar tecnologias de criptografias baseadas em chave pública/privada para garantir a unicidade do *Assembly*. Sendo assim, um *Assembly* assinado com uma *strong name* possui o nome do arquivo, a cultura, a versão e uma chave privada do publicador do mesmo.

A Microsoft disponibilizou um utilitário de linha de comando que permite-nos gerar uma *strong name* para assinarmos os *Assemblys* que desejarmos. Esse utilitário chama-se sn.exe (sn = strong name) e, assim como os compiladores, encontra-se no seguinte local do disco: **C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727**. Basicamente, para gerar um par de chave pública/privada, basta simplesmente fazer:

```
C:\>sn -k C:\Temp\StrongName.snk
```

O parâmetro –k indica ao utilitário para gerar a chave. Lembrando que, se quiser, poderá abrir o *Visual Studio 2005 Command Prompt* para poupar a digitação do caminho completo até o utilitário. O arquivo gerado através do comando acima conterá as chaves pública e privada.

Assinando o Assembly

Agora que já sabe como criar uma strong name, você precisa referênciá-la no seu projeto. Para isso, você deverá utilizar o atributo chamado *AssemblyKeyFileAttribute* que é fornecido pelo .NET Framework e configurá-lo dentro do arquivo *AssemblyInfo*. O código abaixo exemplifica o uso deste atributo:

```
VB.NET
<Assembly: AssemblyKeyFile("C:\Temp\StrongName.snk")>

C#
[assembly: AssemblyKeyFile(@"C:\Temp\StrongName.snk")]
```

Quando o compilador encontra esse atributo, ele assinará o *Assembly* com a chave privada e irá incorporar a chave pública. Isso, além de garantir a unicidade do *Assembly*, garantirá também a chacagem de integridade, ou seja, passando pelas checagens de segurança do .NET Framework, irá garantir que o conteúdo do *Assembly* não foi alterado desde a sua última compilação.

Formas de distribuição de Assemblies

Assemblies privados

Quando referenciamos um determinado *Assembly* no projeto que estamos desenvolvendo, esse componente geralmente será copiado para a pasta da aplicação. Outra possibilidade é quando fazemos uma referência direta para o mesmo, e o manipulamos via **Reflection**. Esses tipos de *Assemblies* são considerados *Assemblies* privados, pois são utilizados por uma ou mais aplicações específicas, mas sempre tendo uma cópia local para mesmo.

Esse tipo de *Assemblies* apesar de funcionar bem em alguns cenários, dificulta o processo de deployment quando se trata de um cenário mais complexo, onde a quantidade de aplicações/usuários que utilizam esses *Assemblies* é muito grande. A alternativa a este método é a depositar esse *Assembly* no **Global Assembly Cache**, tema da próxima seção.

Global Assembly Cache – GAC

Como vimos acima, uma das maiores dificuldades quando trabalhamos com *Assemblies* privados é a questão do deployment. Mesmo um *Assembly* assinado com uma *strong name* permitiria, por exemplo, a execução lado-a-lado. É neste momento que entra em ação o **GAC – Global Assembly Cache**.

Se um *Assembly* poderá ser carregado por múltiplas aplicações, esse *Assembly* deverá ser colocado em um local de conhecimento de todos. Esse local conhecido trata-se do GAC,

que é um centralizador dos *Assemblies* disponíveis para consumo. Quando um *Assembly* é adicionado ali, ele terá uma séria de benefícios, tais como:

- Facilidade de deploy, já que está tudo centralizado
- Melhoria na performance
- Possibilita a execução lado-a-lado de um mesmo *Assembly* com versões diferentes, já que um diretório físico não resolve o problema, pois podemos ter dois componentes com o mesmo nome de arquivo.

Há três formas de inserirmos um determinado *Assembly* no GAC:

- **Windows Explorer:** Se navegar via Windows Explorer até o diretório **C:\WINDOWS\ASSEMBLY** você verá todos os *Assemblies* que estão no GAC daquela maquina específica. Você pode, via *drag-and-drop*, adicionar *Assemblies* ali, mas isso somente é interessante em ambiente de desenvolvimento.
- **Utilitário Gacutil.exe:** Este utilitário, fornecido também pelo SDK do .NET Framework, permite via linha de comando, interagir com o GAC, adicionando, removendo, listando, etc., *Assemblies*. Este utilitário é utlizando em ambientes de testes e desenvolvimento, nunca em produção.
- **Installers:** Permite adicionarmos a instalação dos componentes dentro do GAC via *Windows Installer*. Isso é perfeitamente útil quando desejamos empacotar o sistema em um projeto de setup para que o usuário final, muitas vezes sem muito conhecimento, possa instalar o sistema sem maiores dificuldades. Este é a forma que se utiliza para a instalação de um componente em uma máquina cliente, já que o .NET Framework redistribuível não possui o utilitário **GacUtil.exe**.

A imagem abaixo mostra o Global Assembly Cache – GAC, já com uma porção de Assemblies adicionadas pela instalação do .NET Framework. Cada linha contém o nome do *Assembly*, a versão, a cultura e a chave pública:

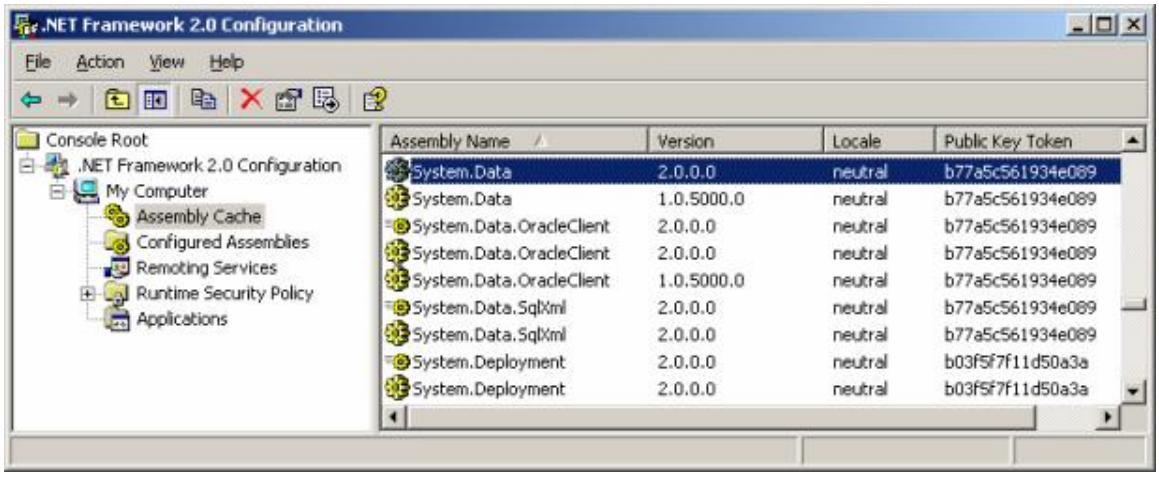


Imagem 3.2 – Global Assembly Cache – GAC.

Instalando um Assembly no GAC

Para instalar um *Assembly* no GAC, somente é permitido se o usuário que estiver fazendo isso tiver privilégios administrativos e, o mais importante, o *Assembly* deverá ter uma *strong name* definida. O trecho de código abaixo exibe o resultado da adição de um componente dentro do GAC e, a imagem a seguir, exibe o GAC já com o componente devidamente adicionado:

```
C:\>gacutil -i C:\Temp\ComponenteComum.dll
Microsoft (R) .NET Global Assembly Cache Utility. Version
2.0.50727.42
Copyright (c) Microsoft Corporation. All rights reserved.

Assembly successfully added to the cache
```

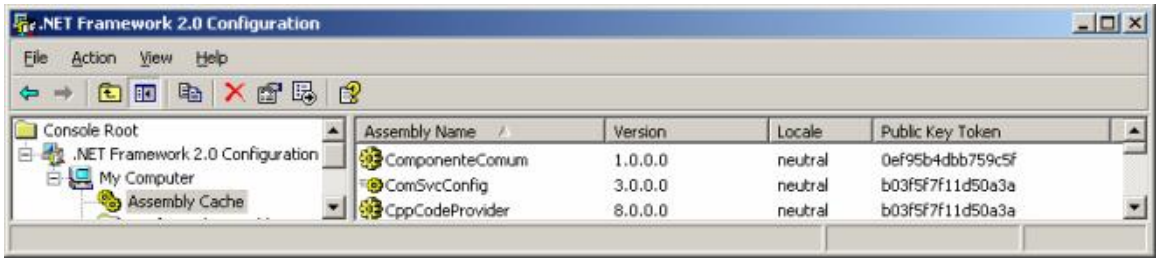


Imagem 3.3 – Componete já adicionado ao GAC.

Nota: Somente adicione no GAC os *Assemblies* que realmente serão compartilhados entre várias aplicações, caso contrário, o *Assembly* deverá ser implantando privadamente, ou seja, junto ao diretório da aplicação.

Embutindo arquivos dentro de um Assembly

Como vimos acima, um contém módulos gerenciados, metadados, manifesto, código IL e os recursos. Recursos ou arquivos de recursos são informações que também são embutidas dentro de um determinado *Assembly* que são informações necessárias para o funcionamento do *Assembly*.

Geralmente essas informações são imagens, arquivos xml, arquivos texto, mensagens, etc.. Se analisarmos o próprio .NET Framework, ele possui uma porção de recursos e, como exemplo, podemos citar o *Assembly* **System.Windows.Forms.dll** que embuti várias imagens que são utilizadas pelos controles que lá estão contidos quando são colocados na barras de ferramentas do Visual Studio .NET. Um outro exemplo é o caso do *Assembly* **System.Web.dll** que, na seção de recursos, além de imagens, possui também arquivos *javascript* que são utilizados pelos controles do ASP.NET.

Quando estamos em um mundo mais próximo da nossa realidade, ícones ou imagens que nossa aplicação depende para ser exibido nos formulários da aplicação, poderiam ser também embutidos dentro do *Assembly*. Sendo assim, podemos então adicionar uma imagem qualquer dentro do projeto e, clicando com o botão direito do mouse e indo até a opção "Propriedades", a janela de Propriedades é exibida. Encontre a propriedade *Build Action* e a defina como *Embedded Resource*. Isso fará com que o arquivo (seja texto, imagem, ícone, etc.) seja embutido dentro do Assembly. Supondo que a imagem adicionada foi **Background.jpg**, a chave para acessá-la será: **NomeProjeto.Background.jpg**. O trecho de código abaixo exibe a forma de recuperar a imagem via código e definí-la como sendo o *Background* de um formulário *Windows Forms*.

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

Dim asb As [Assembly] = Assembly.GetExecutingAssembly()
Me.BackgroundImage = _
    New
Bitmap(asb.GetManifestResourceStream("VB.Background.jpg"))

C#
using System.IO;
using System.Reflection;

Assembly asb = Assembly.GetExecutingAssembly();
this.BackgroundImage =
    new
Bitmap(asb.GetManifestResourceStream("CS.Background.jpg"));
```

A possibilidade de incluir arquivos dentro de um *Assembly* facilita a distribuição, já que você não precisa se preocupar em manter os caminhos dos arquivos, já que eles estão embutidos e a forma de acessá-los é diferente. Além disso, temos uma maior segurança, pois depois de compilado, não é mais possível alterá-lo a menos que, abra o projeto e edite o arquivo e, finalmente, gere um novo *Assembly* contendo as novas informações.

Instalação de Assemblies

Instaladores padrões

Desde as primeiras versões do .NET Framework temos uma nova categoria de tipos de projetos que estão embutidos dentro das *templates* do Visual Studio .NET. Essa categoria trata-se dos projetos de Setup.

Apesar de ser possível utilizarmos a técnica de XCOPY para a instalação de projetos no cliente. O XCOPY trata-se de simplesmente copiar o(s) arquivo(s) do projeto em um dispositivo qualquer, como por exemplo, CD, Pen-drive, etc., e levar até o cliente e lá copiar todos os arquivos para o disco da máquina onde o sistema irá rodar. A desinstalação é tão simples quanto a instalação, bastante apenas localizar a pasta onde o projeto está instalado e excluí-la do disco. Apesar de simples, isso nem sempre é o ideal por muitas razões:

1. Não existe segurança.
2. Exige que a pessoa que irá instalar, tenha um conhecimento razoável.
3. Não conseguimos automatizar as configurações na máquina durante a instalação.

Visando todos esses problemas, a Microsoft decidiu criar os projetos de Setup para auxiliar essa tarefa bastante comum. Os projetos são bastante interessantes e permitem uma construção muito rápido, sem a necessidade de adquirir componentes de terceiros. Um dos grandes benefícios é a possibilidade de gerar uma instalação "transacionada", ou seja, se durante a instalação algo falhar, seguramente ele irá desfazer todas as mudanças que o instalador fez. Atualmente temos os seguintes projetos disponíveis:

Tipo de Projeto	**Descrição**
Setup Project	Utilizado para a criação de projetos de instalação para aplicações que rodam em ambiente Windows (Windows Forms).
Web Setup Project	Utilizado para criação de projetos de instalação para aplicações ASP.NET.
Merge Module Project	Utilizado para criar um instalador para componentes compartilhados.
CAB Project	Utilizado para gerar e comprimir arquivos CAB para disponibilizá-los para download.

Customizando da instalação e desinstalação

Os projetos de Setup atendem perfeitamente para a instalação básica de um software: cópia de arquivos, criação de pastas, criação de items no menu Iniciar do Windows e atalho na área de trabalho do usuário. Mas há cenários onde você quer customizar a instalação de um projeto. Geralmente esse projeto requer diversas configurações que você precisará informar ou que ele mesmo possa criar durante o processo de instalação.

Essas configurações são, por exemplo, criação de base de dados e seus objetos dentro do *SQL Server*, criação de arquivos no disco, criação de *Message Queue*, entre outras várias possibilidades. Para resolver esse problema, podemos utilizar o que chamamos de classes de instalação ou **Installer Classes**.

Fornecidas pelo namespace **System.Configuration.Install** (para ter acesso as classes, é necessário fazer a referência à **System.Configuration.Install.dll**), essas classes disponibilizam uma gama de funcionalidades que podemos customizar a instalação do nosso projeto. Entre as classes disponíveis, temos a classe **Installer**, **AssemblyInstaller**, **ComponentInstaller**, **InstallContext** e a classe **TransactedInstaller**. Como a idéia aqui é customizar toda a instalação de um determinado software, essas classes vão nos auxiliar durante toda a criação desta forma customizada de instalar uma aplicação.

Installer

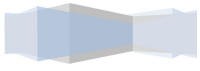
A classe **Installer** é responsável por fornecer todas as funcionalidades necessárias para a customização da instalação, sendo a classe base para todos os outros *Installers* dentro do .NET Framework. O passos a seguir definem o processo que deve ser realizado para que o instalador seja construído e executado corretamente:

1. Criar um instalador, herdando da classe Installer.
2. Sobrescreva os métodos *Install*, *Commit*, *Rollback* e *Uninstall*. Esses métodos não em visibilidade pública, pois são marcados como *protected*. Serão invocados automaticamente a partir da instalação do software.
3. Adicione o atributo **RunInstallerAttribute** a classe instaladora que está criado.
4. Invoque o instalador. Pode ser através do utilitário de linha de comando **installutil.exe** ou através da classe **AssemblyInstaller**, qual veremos mais detalhadamente logo abaixo.

Esta classe possui uma propriedade denominada *Installers*, que recebe como parâmetro uma coleção de *Installers* que, por sua vez, cada elemento é um tipo **Installer**. Esses *Installers* farão parte de uma mesma instalação e nos permite ter uma hierarquia de instaladores.

Cada um dos métodos *Install*, *Commit*, *Rollback* e *Uninstall* recebem como parâmetro uma coleção do tipo chave-valor (**IDictionary**) para que você possa receber informações entre os métodos da instalação corrente e, quando o mesmo é finalizado, eles valores são persistidos para mais tarde, quando a desinstalação acontecer, você consiga desfazer o que tinha feito. Para exemplificar, imagine que durante a instalação, eu preciso criar um arquivo no disco com um nome que é gerado dinamicamente. Esse nome é gerado e então, necessitamos guardá-lo para que no momento da desinstalação da aplicação (ou no *Rollback* da instalação), você consiga excluir o arquivo gerada pelo instalador.

O atributo **RunInstallerAttribute** especifica que quando o instalador customizado for executado para instalar um determinado *Assembly*, ele deverá invocar todas as classes que estão decoradas com este atributo e que em seu construtor, o valor definido esteja como *True*.



Para transformarmos isso em código, abaixo é mostrado um exemplo onde temos uma aplicação console que é o sistema que desenvolvemos e que o cliente precisará utilizar e,

em seguida, a classe instaladora do nosso sistema. Ainda na aplicação que o usuário terá acesso, além de termos os códigos normais, teremos que ter também o instalador que é a classe que herda da classe **Installer** e aplica o atributo **RunInstallerAttribute**.

VB.NET

```
Console.WriteLine("Esta é aplicação que o cliente solicitou...")
Console.ReadLine()
```

C#

```
Console.WriteLine("Esta é aplicação que o cliente solicitou...");
Console.ReadLine();
```

Instalador (dentro da mesma aplicação)

VB.NET

```
Imports System
Imports System.IO
Imports System.Collections
Imports System.ComponentModel
Imports System.Configuration.Install

Namespace Aplicacao
    <RunInstaller(true)> _
    Public Class Instalador
        Inherits Installer

        Public Overrides Sub Install(ByVal stateSaver As
IDictionary)
            Dim file As String = "c:\ViveraDuranteAplicacao.txt"
            Using sw As New StreamWriter(file)
                sw.Write("Conteudo!")
            End Using
            stateSaver("Arquivo") = file
            MyBase.Install(stateSaver)
        End Sub

        Public Overrides Sub Uninstall(ByVal savedState As
IDictionary)
            Dim file As String = savedState("Arquivo").ToString()
            If File.Exists(file) Then
                File.Delete(file)
            End If
            MyBase.Uninstall(savedState)
        End Sub
    End Class
End Namespace
```

C#

```
using System;
using System.IO;
using System.Collections;
using System.ComponentModel;
using System.Configuration.Install;

namespace Aplicacao
{
    [RunInstaller(true)]
    public class Instalador : Installer
    {
        public override void Install(IDictionary stateSaver)
        {
            string file = @"c:\ViveraDuranteAplicacao.txt";

            using (StreamWriter sw = new StreamWriter(file))
            {
                sw.Write("Conteudo!");
            }

            stateSaver["Arquivo"] = file;
            base.Install(stateSaver);
        }

        public override void Uninstall(IDictionary savedState)
        {
            string file = savedState["Arquivo"].ToString();

            if (File.Exists(file))
                File.Delete(file);

            base.Uninstall(savedState);
        }
    }
}
```

Como podemos reparar, somente optamos por sobrescrever os métodos *Install* e *Uninstall*. Como já era de se esperar, o primeiro deles ocorre apenas quando a aplicação for instalada. Como estamos colocando o nome do arquivo gerado dentro da coleção que vem como parâmetro (*stateSaver*), ele valor será mantido e mais tarde, quando o método Uninstall acontecer, ele conseguirá recuperar o nome do arquivo para que ele possa excluí-lo.

A classe **Installer** ainda contém uma propriedade bastante útil chamada *Context* do tipo **InstallContext** que, como o próprio nome diz, traz informações a respeito do contexto de instalação do *Assembly*, como por exemplo, o local do arquivo de log da instalação, o local do arquivo para salvar informações requisitadas pelo método *Uninstall* e, os argumentos que são passados através do utilitário **installutil.exe**, quais são mapeados

para entradas dentro da propriedade *Parameters*, do tipo **StringDictionary**. Além disso, a classe **InstallContext** ainda fornece um método chamado *LogMessage* que permite escrevermos algo na console e no arquivo de log da instalação.

Eventos

A classe Installer também fornece uma porção de eventos interessantes para comunicar-se com o cliente e notificá-lo de acordo com o progresso da instalação. A tabela abaixo descreve cada um destes eventos disponíveis:

Evento	Descrição
AfterInstall	Ocorre depois que todos os métodos *Install* de todos os instaladores contidos na propriedade *Installers* foram executados.
AfterRollback	Ocorre depois que todos os métodos *Rollback* de todos os instaladores contidos na propriedade *Installers* foram executados.
AfterUninstall	Ocorre depois que todos os métodos *Uninstall* de todos os instaladores contidos na propriedade *Installers* foram executados.
BeforeInstall	Ocorre antes que o método *Install* de cada instalador contido na propriedade *Installers* seja executado.
BeforeRollback	Ocorre antes que o método *Rollback* de cada instalador contido na propriedade *Installers* seja executado.
BeforeUninstall	Ocorre antes que o método *Uninstall* de cada instalador contido na propriedade *Installers* seja executado.
Committed	Ocorre depois de que todos os instaladores contidos na propriedade *Installers* concretizaram o trabalho com êxito.
Committing	Ocorre antes de todos os instaladores contidos na propriedade *Installers* concretizaram o trabalho.

AssemblyInstaller

Para fins de exemplo, também criaremos uma aplicação console para que sirva como instalador do sistema acima criado. Para conseguirmos instalar a aplicação dinamicamente, ou seja, via código, é será necessário utilizarmos uma nova classe, também exposta pelo *namespace* **System.Configuration.Install**. Neste momento entra em cena a classe **AssemblyInstaller**.

Também derivada da classe **Installer**, esta classe é responsável por instalar um *Assembly*. Dado o caminho físico completo até o *Assembly*, essa classe carrega-o e extrair de dentro dele todos os instaladores lá contidos e os executam. Para que isso seja possível, devemos seguir rigorosamente os passos que vimos acima para a criação do *Installer* e mantendo todos os instaladores com o modificador de acesso definido como *public*, caso contrário, não é possível ser acessado externamente. O trecho de código abaixo cria a instância da classe **AssemblyInstaller** apontando para o *Assembly* que deseja instalar.

Quando invocamos os métodos *Install*, *Commit*, *Rollback* e *Uninstall* da classe **AssemblyInstaller**, internamente ela invoca os respetivos métodos dos instaladores do *Assembly* informado. O trecho de código invoca o método *Install* e *Commit*:

VB.NET
```
Imports System
Imports System.Collections
Imports System.Configuration.Install

Dim savedState As IDictionary = New Hashtable()
Dim cmdLine() As String = New String() {"/LogFile=Log.txt"}

Using installer As New AssemblyInstaller
("c:\Temp\Aplicacao.exe", cmdLine)
    installer.Install(savedState)
    installer.Commit(savedState)
End Using
Console.WriteLine("Aplicao Instalada.")
Console.ReadLine()
```

C#
```
using System;
using System.Collections;
using System.Configuration.Install;

IDictionary savedState = new Hashtable();
string[] cmdLine = new string[] { "/LogFile=Log.txt" };

using (AssemblyInstaller installer = new
AssemblyInstaller("c:\\Temp\\Aplicacao.exe", cmdLine))
{
    installer.Install(savedState);
    installer.Commit(savedState);
}

Console.WriteLine("Aplicação Instalada.");
Console.ReadLine();
```

Mas e se desejarmos desinstalar a aplicação? Bem, é tão simples quanto a instalação. Cria-se o objeto **AssemblyInstaller** apontando para o *Assembly*, passando os argumentos se desejar e agora basta chamar o método *Uninstall* ou invés de *Install* e *Commit*.

ComponentInstaller

Mais um derivado da classe **Installer**, este componente permite a possibilidade de interargimos com o instalador, passando para ele uma instância de um objeto para que ele possa utilizar para a instalação correta da aplicação.

Assim como a classe **Installer**, você também deve criar um instalador, mas agora, utilizando como base o **ComponentInstaller**. Esta classe basicamente fornece um método abstrato chamado de *CopyFromComponent* que recebe em seu parâmetro uma instância de um objeto que implemente a *Interface* **IComponent**. Um exemplo típico é quando você precisa criar objetos dentro do *SQL Server* (Stored Procedures, Tabelas, etc.). É notável que para isso precisamos de uma conexão com o servidor de banco de dados. Como pode haver a possibilidade desta conexão já estar disponível, poderia simplesmente mandar essa conexão para que o instalador possa utilizar para a criação dos objetos que mencionamos acima.

TransactedInstaller

Finalmente temos o instalador **TransactedInstaller**, qual também é derivado da classe **Installer**. Esse instalador tem um funcionalida extremamente útil, envolve a instalação da aplicação em um ambiente transacionado. Isso quer dizer que, se alguma exceção ocorrer durante o processo de instalação, o **TransactedInstaller** deixará o computador em qual está sendo instalado em um estado consistente. Basicamente, ele garantirá que o método *Rollback* será executado e, sendo assim, não esqueça de sobrescrever o método em seu instalador para poder contemplar uma possível exceção que possa vir a acontecer. O código abaixo é uma versão transacionada do qual vimos um pouco mais acima, quando ainda falávamos da classe **AssemblyInstaller**:

```
VB.NET
Imports System
Imports System.Collections
Imports System.Configuration.Install

Dim savedState As IDictionary = New Hashtable()
Dim cmdLine() As String = New String() {"/LogFile=Log.txt"}

Using installer As New AssemblyInstaller
("c:\Temp\Aplicacao.exe", cmdLine)
    Using tran As New TransactedInstaller()
        tran.Context = installer.Context
        tran.Installers.Add(installer)

        tran.Install(savedState)
        tran.Commit(savedState)
    End Using
End Using

Console.WriteLine("Aplicao Instalada.")
Console.ReadLine

C#
using System;
```

```
using System.Collections;
using System.Configuration.Install;

IDictionary savedState = new Hashtable();
string[] cmdLine = new string[] { "/LogFile=Log.txt" };

using (AssemblyInstaller installer = new
AssemblyInstaller(@"c:\\Temp\\Aplicacao.exe", cmdLine))
{
    using (TransactedInstaller tran = new TransactedInstaller())
    {
        tran.Context = installer.Context;
        tran.Installers.Add(installer);

        tran.Install(savedState);
        tran.Commit(savedState);
    }
}

Console.WriteLine("Aplicação Instalada.");
Console.ReadLine();
```

Arquivos de configuração

O que são arquivos de configuração

Os arquivos de configuração são arquivos que colocamos no interior da aplicação que, permitem de uma forma bem flexível, customizar parâmetros para que a mesma possa trabalhar corretamente.

A flexibilidade é verdadeira porque alguns parâmetros variam de um ambiente para outro, como é o caso, por exemplo, de uma conexão com o banco de dados. Quando a aplicação está sendo desenvolvido, muito provavelmente, a conexão tem que apontar para o servidor de banco de dados de desenvolvimento. Agora, quando mandamos essa aplicação para o cliente, as configurações de servidor de base de dados são completamente diferentes. Como não é possível compilar a aplicação no cliente, seria interessante se, de alguma forma, conseguíssemos alterar essa configuração sem a necessidade de recompilar a aplicação.

Temos dois tipos de arquivo de configuração: **App.Config** e **Web.Config**. O primeiro deles é utilizado em aplicações executáveis, como por exemplo, aplicações Windows, Windows Services, etc; já o segundo, é utilizado por aplicações Web/ASP.NET, também com a finalidade de configuração de parâmetros que a aplicação necessita para trabalhar.

Além as configurações de conexões com a base de dados, ainda temos uma seção interessante, que é a **AppSettings**. Essa seção permite colocarmos qualquer tipo de

informação que necessitamos parametrizar na aplicação. Imagine que a sua aplicação manipule alguma fila de mensagens de *Message Queue*. Cada cliente tem a fila disponível para a sua aplicação utilizar mas, cada uma tem um nome diferente. Então para isso, podemos parametrizar o nome da fila no arquivo de configuração e, em cada cliente, você altera o nome, sem a necessidade de qualquer trabalho adicional. Para exemplificar a parametrização da conexão com o banco de dados e as informações de parâmetros via arquivo de configuração, o código dá uma idéia de como proceder:

```
<?xml version="1.0"?>
<configuration>
  <appSettings>
    <add
        key="MessageQueueName"
        value="FileDeMensagens" />
  </appSettings>
  <connectionStrings>
    <add
        name="SqlConnectionString"
        connectionString="Data                    Source=.;Initial
Catalog=MeuBanco;Integrated Security=SSPI;"
        providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Como podemos visualizar, através do elemento *add* podemos adicionar quantos parâmetros e conexões forem necessária para a aplicação poder trabalhar. O que precisa se atentar é com relação aos elementos *key* (em **AppSetings**) e em *name* (em **connectionStrings**), pois elas não podem repetirem dentro do mesmo arquivo. Agora, dentro da aplicação você pode acessar as informações que criamos no arquivo de configuração acima. Para isso, basta utilizar a classe **ConfigurationManager** que está contida dentro do *namespace* **System.Configuration**. Entre várias funcionalidades, essa classe expõe duas propriedades chamadas *AppSettings* e *ConnectionStrings* que retornam coleções de cada uma das seções. O trecho de código exibe a forma que utilizamos para poder recuperá-las:

```
VB.NET
Imports System.Configuration
'...
Console.WriteLine(ConfigurationManager.AppSettings("MessageQueueN
ame"))
Console.WriteLine(ConfigurationManager.ConnectionStrings("SqlConn
ectionString"))

C#
using System.Configuration;
```

```
//...
Console.WriteLine(ConfigurationManager.AppSettings["MessageQueueN
ame"]);
Console.WriteLine(ConfigurationManager.ConnectionStrings["SqlConn
ectionString"]);
```

Criando uma seção de configuração customizada

O que vimos até o momento satisfaz uma boa parte das necessidades que temos. Mas agora, gostaríamos de criar uma seção de configuração própria na aplicação que estamos desenvolvendo para ter um maior controle e uma tipagem mas eficiente. A idéia aqui é mostrar como criar uma seção de configuração específica para a aplicação que estamos desenvolvendo.

Para iniciar, mesmo tendo um namespace disponível na aplicação quando a criamos, é necessário adicionar a referência a uma DLL chamada **System.Configuration.dll**. Essa DLL disponibilizará para a aplicação várias classes que iremos utilizar para a construção dessa seção de configuração customizada.

ConfigurationElement

ConfigurationElement é uma classe abstrata que representa um determinado elemento dentro do arquivo de configuração. Como trata-se de uma classe abstrata, obrigatoriamente deve ser herdada em uma classe concreta que representará elementos de configuração XML, que tratam-se também dos parâmetros que precisamos para que nossa aplicação trabalhe/manipule.

A classe concreta que herda de **ConfigurationElement** basicamente conterá as propriedades que desejam disponibilizar no arquivo de configuration para ser definido e, conseqüentemente, utilizado pela aplicação. Essas propriedades são configuradas com alguns atributos que irão definir toda as características individuais de cada propriedade, como por exemplo, o tipo, nome, valor padrão, etc. Para exemplificar a criação da seção customizada, vamos inicialmente analisar a implementação de **ConfigurationElement** logo abaixo:

```
VB.NET
Imports System.Configuration

Public Class UrlConfigElement

    Inherits ConfigurationElement

    <ConfigurationProperty("name",      DefaultValue:="Microsoft",
IsRequired:=True, IsKey:=True)> _
    Public ReadOnly Property Name() As String
```

```
        Get
            Return MyBase.Item("name").ToString()
        End Get
    End Property

    <ConfigurationProperty("url",
DefaultValue:="http://www.microsoft.com", IsRequired:=True)> _
    <RegexStringValidator("\w+:\/\/[\w.]+\S*")> _
    Public ReadOnly Property Url() As String
        Get
            Return MyBase.Item("url").ToString()
        End Get
    End Property

    <ConfigurationProperty("port",               DefaultValue:=0,
IsRequired:=False)> _
    <IntegerValidator(MinValue:=0,                MaxValue:=8080,
ExcludeRange:=False)> _
    Public ReadOnly Property Port() As Integer
        Get
            Return MyBase.Item("port").ToString()
        End Get
    End Property

End Class
```

C#

```
using System.Configuration;

public class UrlConfigElement : ConfigurationElement
{
    [ConfigurationProperty("name",  DefaultValue  =  "Microsoft",
IsRequired = true, IsKey = true)]
    public string Name
    {
        get
        {
            return (string)this["name"];
        }
    }

    [ConfigurationProperty("url",          DefaultValue          =
"http://www.microsoft.com", IsRequired = true)]
    [RegexStringValidator(@"\w+:\/\/[\w.]+\S*")]
    public string Url
    {
        get
        {
            return (string)this["url"];
        }
```

```
    }

    [ConfigurationProperty("port",    DefaultValue    =    (int)0,
IsRequired = false)]
    [IntegerValidator(MinValue = 0, MaxValue = 8080, ExcludeRange
= false)]
    public int Port
    {
        get
        {
            return (int)this["port"];
        }
    }
}
```

Temos no código acima uma classe chamada *UrlConfigElement* que contém três propriedades: *Name*, *Url* e *Port*. Essa classe representa informações a respeito de um website específica com o seu nome e a porta. Cada uma dessas propriedades contém uma atributo chamado **ConfigurationProperty** que, como vimos acima, indica como ela será representada pelo XML dentro do arquivo de configuração.

Esse atributo permite-nos informações algunas informações importantes a respeito da propriedades, uma forma de vincular as propriedades do código à configurações do arquivo de configuração. A tabela abaixo descreve as principais configurações que podemos fazer para cada propriedade individualmente:

Propriedade	**Descrição**
DefaultValue	Define um valor padrão para a propriedade que é utilizada quando não é informada pelo arquivo de configuração. Geralmente é utilizada quando a propriedade *IsRequired* é definida como *False*.
Description	Utilizado apenas para definir uma descrição. Ela não é utilizada pela aplicação.
IsKey	Indica se a propriedade é a chave para o objeto, qual também será utilizada dentro de uma possível coleção dentro do arquivo de configuração.
IsRequired	Indica se a propriedade é ou não obrigatória.
Name	Especifica o nome da propriedade que será utilizado no arquivo de configuração.
Type	Define o tipo que a propriedade deverá ter no arquivo de configuração.

Validators

Os validadores disponibilizam uma forma interessante para validarmos as informações que são colocadas no arquivo de configuração. Assim como o atributo **ConfigurationProperty**, os validadores também são definidos via atributos. Todos os

validadores para esta finalidade herdam diretamente de uma classe abstrata chamada **ConfigurationValidatorBase** que define dois métodos que devem ser implementados: *CanValidate* e *Validate*. O primeiro deles, *CanValidate* define se um objeto pode ser validado baseando-se em um determinado tipo; o segundo e último método, *Validate*, determina a validação efetiva do objeto, indicando se ele está ou não de acordo com o que a aplicação espera para trabalhar.

Existem vários validadores disponíveis dentro do .NET Framework, mais precisamente, dentro do *namespace* **System.Configuration** e, como já era de se esperar, podemos implementar o nosso próprio validador herdando de **ConfigurationValidatorBase**. Se analisarmos o trecho de código acima, veremos o uso de dois validadores diferentes: **RegexStringValidator** e **IntegerValidator**. O primeiro deles permite validar uma determinada *string* de acordo com uma *regular expressions*; já o **IntegerValidator** faz a verificação para saber se o valor informado é ou não do tipo *Int32*. Ambos atributos fornecem propriedades diferentes, que são necessárias de acordo com o tipo de validação.

ConfigurationElementCollection

Como o próprio nome diz, **ConfigurationElementCollection**, trata-se de uma coleção de elementos dentro do arquivo de configuração. A idéia por trás desta objeto/coleção está em permitir a declaração no arquivo de configuração de múltiplos elementos de um mesmo tipo que, no nosso caso, é o *UrlConfigElement*.

A sua implementação não é muito complexa. Trata-se também de uma classe abstrata que fornece dois principais métodos que devem ser implementados na classe derivada: *CreateNewElement* e *GetElementKey*. O método *CreateNewElement* permite criarmos elementos de um determinado tipo que deve sempre retornar uma instância do objeto que estamos querendo armazenar dentro da nossa coleção. Já o método *GetElementKey* deve retornar a chave para o objeto, que é sempre a propriedade que está marcada com o atributo *IsKey* do atributo **ConfigurationProperty** dentro do objeto que herda de **ConfigurationElement**. O código abaixo demonstra como devemos proceder para utilizar a classe abstrata **ConfigurationElementCollection**:

```
VB.NET
Imports System.Configuration
'...
Public Class UrlColl

    Inherits ConfigurationElementCollection

    Protected Overloads Overrides Function CreateNewElement() As
ConfigurationElement
        Return New UrlConfigElement()
    End Function

    Protected Overrides Function GetElementKey(ByVal element As
```

```
ConfigurationElement) As Object
        Return DirectCast(element, UrlConfigElement).Name
    End Function

End Class
```

C#

```
using System.Configuration;
//...
public class UrlsCollection : ConfigurationElementCollection
{
    protected override ConfigurationElement CreateNewElement()
    {
        return new UrlConfigElement();
    }

    protected override Object GetElementKey(ConfigurationElement
element)
    {
        return ((UrlConfigElement)element).Name;
    }
}
```

ConfigurationSection

Finalmente, temos a classe (também abstrata) **ConfigurationSection**. Ela representa uma seção dentro do arquivo de configuração, que permite-nos customizar uma seção para um determinado tipo que criamos dentro da aplicação. Você deverá herdá-la para fornecer ao sistema uma forma programática de fortemente tipada de acessar as configurações que foram definidas no arquivo de configuração.

Assim como aconteceu com a classe que herda de **ConfigurationElement**, a seção que vamos customizar a partir de **ConfigurationSection** precisa apenas das propriedades que a mesma irá expor para serem configuradas a partir do arquivo de configuração. Como a idéia aqui é também mostrar o uso da coleção que criamos um pouco mais acima então, uma das propriedades, irá retornar uma coleção com todos os elementos já configurados. Através do código abaixo podemos analisar a implementação da seção customizada, já com a propriedade que expõe a coleção de elementos do tipo *UrlConfigElement* e também um único elemento que irá expor apenas um objeto, também do tipo *UrlConfigElement:*

VB.NET

```
Imports System.Configuration
'...
Public Class UrlsSection
```

```
    Inherits ConfigurationSection

    <ConfigurationProperty("name",      DefaultValue:="Favoritos",
IsRequired:=True, IsKey:=False)> _
    <StringValidator(InvalidCharacters:="
~!@#$%^&*()[]{}/;'\""|\\", MinLength:=1, MaxLength:=60)> _
    Public ReadOnly Property Name() As String
        Get
            Return MyBase.Item("name").ToString()
        End Get
    End Property

    <ConfigurationProperty("simple")> _
    Public ReadOnly Property Simple() As UrlConfigElement
        Get
            Return              DirectCast(MyBase.Item("simple"),
UrlConfigElement)
        End Get
    End Property

    <ConfigurationProperty("urls", IsDefaultCollection:=False)> _
    Public ReadOnly Property Urls() As UrlColl
        Get
            Return DirectCast(MyBase.Item("urls"), UrlColl)
        End Get
    End Property

End Class
```

C#

```
using System.Configuration;
//...
public class UrlsSection : ConfigurationSection
{
    [ConfigurationProperty("name",  DefaultValue  =  "Favoritos",
IsRequired = true, IsKey = false)]
    [StringValidator(InvalidCharacters            =             "
~!@#$%^&*()[]{}/;'\"|\\", MinLength = 1, MaxLength = 60)]
    public string Name
    {
        get
        {
            return (string)this["name"];
        }
    }

    [ConfigurationProperty("simple")]
    public UrlConfigElement Simple
    {
        get
```

```
        {
            return (UrlConfigElement)base["simple"];
        }
    }

    [ConfigurationProperty("urls", IsDefaultCollection = false)]
    public UrlsCollection Urls
    {
        get
        {
            return (UrlsCollection)base["urls"];
        }
    }
}
```

Temos então a propriedade *Name* que expõe uma *string*, a propriedade *Simple* que retorna uma instância (individual e isolada) do elemento *UrlConfigElement* e a propriedade *UrlsCollections* que, como o próprio nome diz, retorna uma coleção de objetos do tipo *UrlConfigElement*, conforme criamos um pouco mais acima.

Registrando a seção customizada

Depois de todas as classes criadas, chega o momento que precisamos configurar o arquivo de configuração para definir como estruturar o arquivo e também colocar o valor correspondente a cada uma das propriedades.

Para registrarmos essa seção dentro do arquivo de configuração da aplicação é necessário utilizarmos o elemento *configSections*. Através dele, especificamos o nome da seção que será definido para estruturarmos o XML. Além disso, é necessário informar o tipo que será utilizado (que na verdade é a classe *UrlsSection* que criamos acima) e também o *Assembly* em qual ela se encontra. Para exemplificar a configuração, vamos analisar o código abaixo:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section
      name="urlsSection"
      type="Aplicacao.UrlsSection, Aplicacao" />
  </configSections>
</configuration>
```


Como podemos ver, dentro do elemento *configSections* criamos uma nova seção através do elemento *section*. Primeiramente precisamos informar o nome que essa seção terá dentro do arquivo de configuração. Aqui você pode escolher o que achar mais viável para

a sua aplicação mas, lembre-se que é a partir dela que irá recuperar as informações no seu código. Em seguida, utilizamos o atributo type para especificar que essa seção irá ser baseada em um tipo que criamos no código. Esse tipo deve ser informado com o seu full-name, ou seja, desde o *namespace* raiz até a classe que será utilizada. Além disso, ainda é necessário dizer qual é o *Assembly* em que o mesmo está contido que, no nosso caso, chama-se *Aplicacao*.

Para complementar o arquivo de configuração, necessitamos agora incluir a seção customizada que criamos anteriormente. Como nomeamos essa seção como *urlsSection*, é ela que será utilizada para configurarmos todas as propriedades que criamos via código. A começar pela propriedade *Name* e também uma propriedade *Simple*, que armazena um elemento individual do tipo *UrlConfigElement*. Para finalizar, temos a propriedade *Urls* que é uma coleção e a sua estrutura é um pouco diferenciada, justamente porque armazena vários elementos que, no nosso caso, é do tipo *UrlConfigElement*. Abaixo temos o arquivo de configuração na íntegra:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section
      name="urlsSection"
      type="Aplicacao.UrlsSection, Aplicacao" />
  </configSections>
  <urlsSection name="Favoritos">
    <simple
        name="Microsoft"
        url="http://www.microsoft.com"
        port="0" />
    <urls>
      <clear />
      <add
          name="People"
          url="http://www.people.com.br"
          port="0" />
      <add
          name="Projetando.NET"
          url="http://www.projetando.net/"
          port="8080" />
    </urls>
  </urlsSection>
</configuration>
```

Como podemos notar, temos o elementos *urls* qual podemos adicionar quantos elementos desejarmos, pois trata-se de uma coleção. Agora, para acessarmos essas informações dentro da aplicação, é necessário utilizarmos a classe **ConfigurationManager** que

fornece vários métodos estáticos, que permitem manipular o arquivo de configuração da aplicação corrente.

Um método importante que a classe **ConfigurationManager** disponibiliza é o método *OpenExeConfiguration* que retorna um objeto do tipo **Configuration** que representa as configurações que estão no arquivo de configuração. Mas no nosso caso, utilizaremos uma espécie de atalho. Trata-se de um outro método, chamado *GetSection* que, dado uma *string* com o nome da seção, ele retorna a instância da mesma. O código abaixo mostra como resgatar as informações do arquivo de configuração e, para fins de exemplo, vamos apenas mostrá-los na tela:

```
VB.NET
Imports System.Configuration

Dim section As UrlsSection =
DirectCast(ConfigurationManager.GetSection("urlsSection"),
UrlsSection)
Console.WriteLine(section.Name)

For Each element As UrlConfigElement In section.Urls
    Console.WriteLine(element.Name & " - " & element.Url)
Next

C#
using System.Configuration;

UrlsSection sec = ConfigurationManager.GetSection("urlsSection")
as UrlsSection;
Console.WriteLine(sec.Name);

foreach (UrlConfigElement element in sec.Urls)
    Console.WriteLine(element.Name + " - " + element.Url);
```

O quadro abaixo exibe o *output* deste código:

```
Favoritos
People - http://www.people.com.br
Projetando.NET - http://www.projetando.net/
```