Capítulo 4
Monitoramento e depuração de aplicações

Introdução

Como o Visual Studio .NET é a principal ferramenta para o desenvolvimento de aplicações .NET, não seria completa se não tivesse várias funcionalidades em sua IDE que permite a fácil depuração e monitoramento das aplicações. A utilização do depurador do Visual Studio .NET tem duas vantagens:

1. fornece uma interface familiar que permite o acompanhamento passo-à-passo do código/processo.
2. tem um forte integração com o Visual Studio .NET.

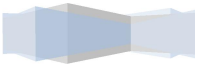
O Visual Studio .NET em conjunto com o seu depurador fornecem uma gama de ferramentas que dão uma grande flexibilidade para a depuração completa de uma aplicação, seja ela Windows, Web ou qualquer tipo de serviço. Além das funcionalidades que já estão embutidas, elas também são extensíveis, o que permite os desenvolvedores customizarem, extendendo alguns aspectos e funcionalidades para ter uma melhor visualização de problemas mais específicos.

A primeira parte do capítulo trata de como manipular o Event Log do Windows, que é um repositório de informações referentes as mais diversas aplicações que são executadas em cima do sistema operacional. Veremos como criar repositórios específicos para a nossa aplicação dentro dele, bem como logar informações dentro dela. Logo em seguida, analisaremos como manipular os processos que estão sendo executados na máquina em que a aplicação é executada.

Já na segunda parte, analisaremos grande parte das funcionalidades que a IDE do Visual Studio .NET fornece para ajudar na depuração da aplicação e também como extendê-la com um exemplo simples. Finalmente, veremos como habilitar o *Tracing* dentro da aplicação, que permite acompanhar o comportamento da mesma durante os testes, bem como quando estiver em produção, para detectar possíveis falhas.

Windows Event Log

Quando um problema ocorre o administrador do sistema ou técnicos que fornecem suporte devem ter informações a respeito da falha, quem gerou, quando, etc.. Muitas aplicações persistem erros e eventos de diversas formas, mantendo cada uma, um padrão proprietário que satisfaz a necessidade dela. Esses padrões proprietários possuem diferentes informações, formatos e disponibilizam, também de forma diferente, para o usuário. O *Event Log* do Windows fornece uma forma centralizada e padronizada para que as aplicações (e também para o sistema operacional) possa salvar os eventos e erros que desejarem.



Felizmente o próprio Windows fornece uma interface que permite aos usuários visualizarem os possíveis problemas e eventos que ocorreram em uma determinada aplicação. Essa interface chama-se **Event Viewer** e pode ser visualizada através da imagem abaixo:

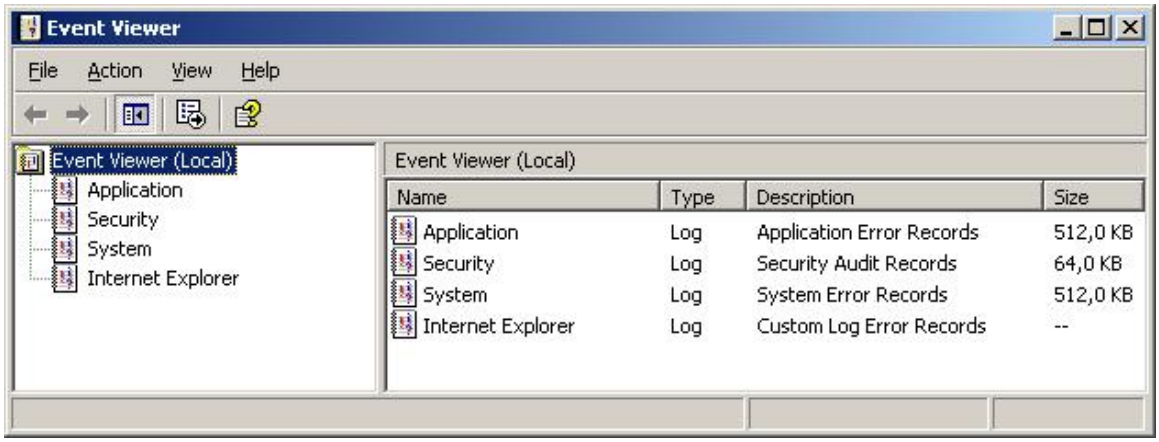


Imagem 4.1 – Interface do Event Viewer do Windows

Tipos de Logs

O .NET Framework já traz embutido diversas classes que auxiliam a manipulação total do *Event Log* do Windows. Essas classes fornecem toda a infraestrutura para a criação de novos repositórios como a gravação de eventos dentro deles. Atualmente existem cinco tipos de eventos que podem ser logados. A tabela abaixo descreve cada um desses tipos:

Tipo de Evento	Descrição
Error	Um evento que indica um problema grave, como por exemplo, uma falha ocorreu quando um serviço tentou ser carregado.
Warning	Um evento que não é uma falha, mas pode representar um futuro problema.
Information	Um evento que tem a finalidade de logar informações a nível de sucesso, como por exemplo, *serviço foi iniciado*, *serviço foi parado*.
Success Audit	Um evento que indica que a auditoria de segurança foi efetuada com sucesso.
Failure Audit	Um evento que indica que a auditoria de segurança falhou.

Quando você não informa nenhum dos tipos de eventos que vimos acima, por padrão, ele assumo o tipo *Information*. Computadores que rodam sistema operacional Windows 2000 ou superior possuem 3 tipos de *event logs*:

1. **System Log:** armazena eventos que ocorrem nos componentes do sistema, como por exemplo um problema com um *driver* qualquer.
2. **Security Log:** armazena eventos com relação à segurança.
3. **Application Log:** armazena eventos que ocorrem em uma aplicação

Arquitetura e manipulação fornecida pelo .NET

O .NET Framework fornece um *namespace* chamado **System.Diagnostics**. É dentro dele que temos todas as classes necessárias para manipular o *Event Log* do Windows. Este *namespace* vai muito além disso, fornecendo classes para *tracing* e monitoramento das aplicações que veremos mais tarde, ainda neste capítulo.

Basicamente temos três tipos principais que são utilizados para a manipulação do *Event Log*: **EventLog**, **EventLogEntryType** e **EventLogEntry**.

A classe **EventLog** é responsável por interagir entre a aplicação e o *Event Log* do Windows, customizando para que a sua aplicação possa gravar os eventos relevantes dentro dele. A partir dele também podemos, além de criar logs e entradas, ler todas as entradas de um log, bem como excluir os logs. Entendam por log o repositório que armazena todas as entradas feitas pelas aplicações. Essa é uma classe que fornece uma porção de métodos estáticos que permitem interagir com o *Event Log*. Para exemplificar a criação de um novo log, vamos analisar o código abaixo:

```
VB.NET
Imports System.Diagnostics

If Not EventLog.SourceExists("AplicacaoWeb") Then
    EventLog.CreateEventSource("AplicacaoWeb", "People")
End If

If Not EventLog.SourceExists("AplicacaoWin") Then
    EventLog.CreateEventSource("AplicacaoWin", "People")
End If

Dim logWeb As New EventLog()
logWeb.Source = "AplicacaoWeb"
logWeb.WriteEntry("Log - Web App.")

Dim logWin As New EventLog()
logWin.Source = "AplicacaoWin"
logWin.WriteEntry("Log - Win App.")

C#
using System.Diagnostics;

if(!EventLog.SourceExists("AplicacaoWeb"))
    EventLog.CreateEventSource("AplicacaoWeb", "People");

if(!EventLog.SourceExists("AplicacaoWin"))
    EventLog.CreateEventSource("AplicacaoWin", "People");
```

```
EventLog logWeb = new EventLog();
logWeb.Source = "AplicacaoWeb";
logWeb.WriteEntry("Log - Web App.");

EventLog logWin = new EventLog();
logWin.Source = "AplicacaoWin";
logWin.WriteEntry("Log - Win App.");
```

Antes de analisar o código, é necessário termos em mente dois termos importantes: *Log* e *Source*. O primeiro deles, *Log*, é o local onde armazenaremos os eventos, ou seja, o repositório dos eventos. Já o segundo, *Source*, é quem (aplicação/serviço) gerou o evento.

Neste momento nos deparamos com um método chamado *WriteEntry* que é responsável por encapsular a escrita de uma entrada dentro do *Event Log*. No exemplo acima, utilizamos um dos *overloads* do método que é disponibilizado, ou seja, somente passando a mensagem que será gravada no *Log*. Dentre vários *overloads* que este método fornece, temos ainda um que recebe como parâmetro, além da mensagem, um enumerador do tipo **EventLogEntryType** que, nada mais é que o tipo de evento (*Error*, *Warning*, *Information*, *Success Audit* e *Failure Audit*) e, como não informamos nenhum tipo no exemplo acima, por padrão, ele assume *Information*.

Como podemos visualizar através do código acima, criamos um *Log* para a empresa People. Todas as aplicações da People vão logar as informações dentro deste repositório. Depois disso, cada aplicações tem o seu identificador, o *Source*, que identificará dentro do Event Log quem foi o responsável pela geração do evento. Através da imagem abaixo você pode analisar os eventos recém criados e com o *Source* já especificado (em vermelho):

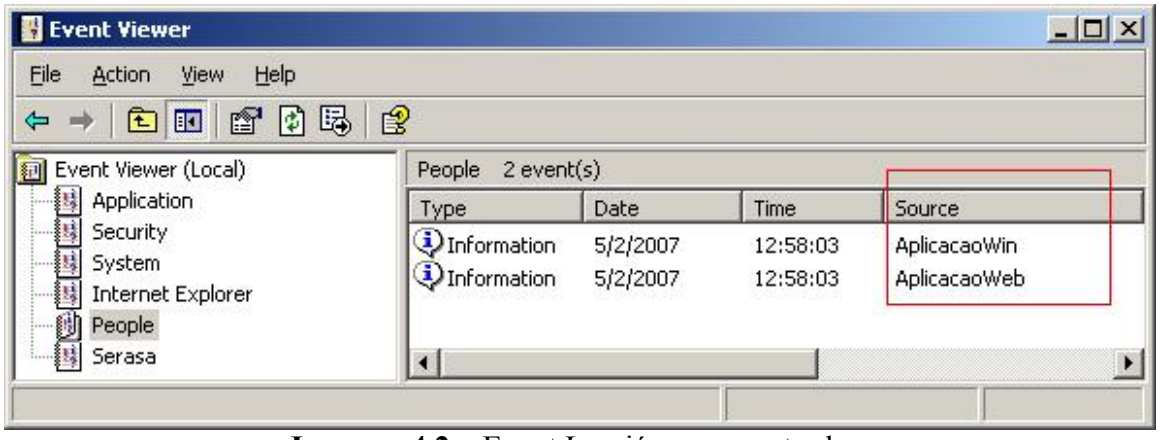


Imagem 4.2 – Event Log já com as entradas.

Se quisermos apagar via aplicação tanto o *Log* quanto o *Source* também é possível através da classe **EventLog**, como é mostrado abaixo:

```
VB.NET
Imports System.Diagnostics

EventLog.DeleteEventSource("AplicacaoWeb")
EventLog.DeleteEventSource("AplicacaoWin")
EventLog.Delete("People")

C#
using System.Diagnostics;

EventLog.DeleteEventSource("AplicacaoWeb");
EventLog.DeleteEventSource("AplicacaoWin");
EventLog.Delete("People");
```

Cada entrada que adicionamos vão sendo colocadas dentro do *Event Log*. Como vimos um pouco mais acima, além dos métodos estáticos que a classe **EventLog** disponibiliza, ela fornece outras funcionalidades, pois a partir de uma instância dela que temos acesso ao *Log* como um todo, permitindo interagir com este *Log*, gravando entradas, lendo essas entradas, etc.. O exemplo abaixo mostra como devemos fazer para exibir os eventos que foram previamente depositados dentro do *Event Log* do Windows:

```
VB.NET
Imports System.Diagnostics

Dim logs As New EventLog("People")

For Each log As EventLogEntry In logs.Entries
    Console.WriteLine(log.Message)
Next

C#
using System.Diagnostics;

EventLog logs = new EventLog("People");

foreach(EventLogEntry log in logs.Entries)
    Console.WriteLine(log.Message);
```

Neste momento temos nos deparamos com uma nova classe do tipo **EventLogEntry**. Esta classe representa uma entrada individual que está dentro do *Event Log*. A única finalidade desta classe é mesmo encapsular todo o acesso à um determinado registro que está dentro do Event Log e, sendo assim, não é permitido criar instâncias da mesma. A forma de trabalhar com ela é mostrado acima, ou seja, quando iteramos através da

propriedade *Entries* do objeto **EventLog**, cada elemento que a coleção retorna é do tipo **EventLogEntry** e, entre as principais propriedades que ela expõe, temos:

Propriedade	Descrição
EntryType	Retorna um enumerador do tipo **EventLogEntryType** que indica qual o tipo da entrada.
Index	Retorna um número inteiro indicando qual o índice que a entrada ocupa dentro do *Log*.
InstanceId	Retorna um número que identifica unicamente a entrada dentro do *Source*.
Message	Recupera a mensagem.
Source	Recupera o nome da aplicação que gerou o evento.
UserName	Recupera o nome do usuário que é responsável pelo evento.

Nota: As aplicações ASP.NET também suportam o acesso a *Event Logs* e fornecem uma arquitetura flexível para escolhermos onde desejamos persistir os eventos. Esse recurso chama-se *Health Monitoring* que está fora do escopo desta curso.

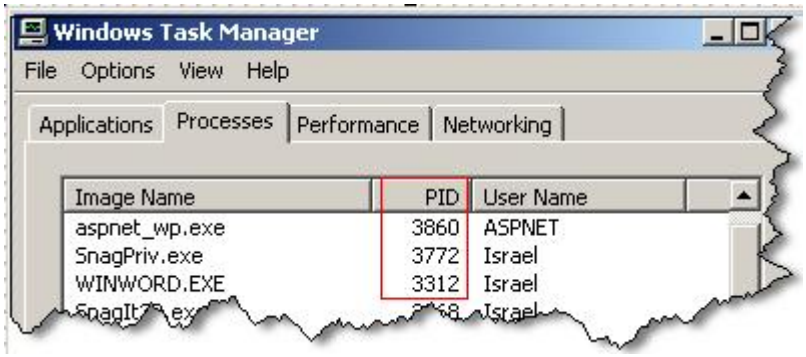
Processos

Imagine que a sua aplicação gera arquivos em formato texto que salvam diversos tipos de informações. Em algum local, dentro desta mesma aplicação, você lista todos os arquivos gerados e disponíveis para que o usuário possa visualizar. Para poupar trabalho do usuário que precisaria conhecer o caminho até o arquivo para abrí-lo no bloco de notas, automatizamos este passo, ou seja, quando o usuário dar um duplo clique em cima do arquivo, o notepad é aberto automaticamente, passando para o mesmo o arquivo texto a ser exibido. É neste momento que um novo processo é criado.

O .NET Framework fornece, também dentro do *namespace* **System.Diagnostics**, uma classe chamada **Process** que permite extrair informações dos processos que estão sendo executados dentro de um determinado computador. Dentro do Windows, você pode ver todos os processos que estão sendo executados no momento através da *Windows Task Manager*, quais podemos também acessá-los dentro de uma aplicação .NET. Para fins de exemplo, vamos utilizar o bloco de notas, mas nada impede de inicializarmos aplicações mais complexas, como por exemplo, o Word, Excel, etc..

A classe **Process** fornece acesso para os processos que estão sendo executados no computador. Essa classe permite-nos controlar um determinado processo, iniciando, parando e para fins de monitoramento da aplicação. A classe **Process** também fornece vários métodos estáticos para a manipulação do de um determinado processo, que permite inicializar diretamente, sem a necessidade de uma instância da classe **Process**. Entre os principais métodos, temos:

Métodos Estáticos	Descrição
GetCurrentProcess	Retorna um objeto do tipo **Process** contendo todas as informações

	do projeto corrente.
GetProcessById	Dado um número inteiro que representa o identificar de um processo, retorna um objeto do tipo **Process** que representa o processo. O Id que ele espera como parâmetro é o PID (*Process Identifier*), qual pode ser visualizado através da barra de tarefas do Windows, como é mostrado através da imagem abaixo:  **Imagem 4.3** – Barra de Tarefas do Windows
GetProcesses	Retorna um *array* de objetos do tipo **Process** que representam todos os processos que estão rodando no computador.
Start	Permite inicializar diretamente um processo, sem a necessidade de criar um objeto do tipo **Process**.

Logo, para inicializarmos o bloco de notas, podemos optar por duas formas: a primeira é utilizar o método estático *Start*. Já a segunda, é através de uma instância de uma classe do tipo **Process**. A segunda te possibilita uma maior controle sobre a mesma se precisar mais tarde recuperar informações a respeito do processo. O código abaixo exibe duas formas equivalentes de inicializar o bloco de notas:

```
VB.NET
Imports System.Diagnostics

'Primeira forma:
Process.Start("notepad.exe", "Arquivo.txt")

'Segunda forma:
Dim p As New Process()
p.StartInfo.FileName = "notepad.exe"
p.StartInfo.Arguments = "Arquivo.txt"
p.Start()

C#
using System.Diagnostics;

//Primeira forma:
Process.Start("notepad.exe", "Arquivo.txt");
```

```
//Segunda forma:
Process p = new Process();
p.StartInfo.FileName = "notepad.exe";
p.StartInfo.Arguments = "Arquivo.txt";
p.Start();
```

Depois de inicializado o processo, se em algum momento quisermos finalizar o mesmo, utilizamos o método *Kill*, que força a finalização do processo. Como esse método é executado assincronamente, em seguida invoque o método *WaitForExit* para aguardar o processo ser finalizado, se assim desejar.

Agora, se desejar listar todos os processos que estão sendo executados atualmente na máquina, então pode optar pela utilização do método *GetProcesses*, assim como é mostrado abaixo:

VB.NET
```
Imports System.Diagnostics

For Each p As Process In Process.GetProcesses()
    Console.WriteLine(String.Format("Processo: {0} - Id: {1}", _
        p.ProcessName, _
        p.Id))
Next
```

C#
```
using System.Diagnostics;

foreach(Process p in Process.GetProcesses())
{
    Console.WriteLine(String.Format("Processo: {0} - Id: {1}",
        p.ProcessName,
        p.Id));

}
```

Depurando uma aplicação

Sempre que estamos desenvolvendo aplicações, independente de qual linguagem ou plataforma estamos utilizando, há sempre uma necessidade muito grande de podemos depurá-la enquanto o processo de desenvolvimento acontece. A depuração consiste em conseguirmos encontrar possíveis falhas dentro da nossa aplicação e, entre essas falhas, temos basicamente três tipos: sintaxe, runtime e lógica.

A primeira falha, a de sintaxe, que a mais fácil de ser identificada, acontece quando escrevemos algum tipo de código que o compilador não consegue entender e já acusa o problema antes mesmo da aplicação ser executada. Em segundo lugar, temos as falhas que ocorrem em tempo de execução que, se não for tratada, muitas vezes força a aplicação a ser fechada e, para citar alguns exemplos temos: acesso a algum componente, servidor de banco de dados inexistente, serviço indisponível, etc.. Finalmente, temos as falhas de lógica que são um pouco mais difícies de encontrar, já que o compilador não consegue antecipar a falha e ela provavelmente irá explodir quando tal código for executado. Alguns exemplos desse tipo de falha são: incompatibilidade de tipos, divisão por zero, objeto inexistente, dados inválidos, etc..

Com todos esses possíveis problemas que todos nós desenvolvedores estamos acostumados a enfrentar, seria terrível se não tivéssemos uma ferramenta eficaz para nos ajudar a encontrar o problema que está acontecendo. Felizmente o Microsoft Visual Studio .NET 2005 fornece uma grande ferramenta de *debugger* que auxilia na depuração de qualquer projeto que está sendo nele construído, permitindo diagnosticar e consertar rapidamente o problema que está ocorrendo. O debugger permite você analisar e modificar valores de variáveis, visualizar a *StackTrace*, threads, *dumps* de memória e muito mais. Além disso, possui *Intellisense* em algumas janelas usadas exclusivamente para depuração e, apesar de fortemente integrado com o Visual Studio 2005, não deixa de ser flexível e permite que nós desenvolvedores estendam as funcionalidades e customizamos alguns controles para facilitar ainda mais o processo de depuração de código.

Quando você desenvolve uma aplicação, no Visual Studio .NET você trabalha em dois modos: *modo de desenvolvimento* e *modo de execução*. Quando estamos em modo de desenvolvimento, você pode criar, editar o código e também detectar possíveis problemas de sintaxe, que a própria IDE lhe ajudará a resolver, como por exemplo é o caso do Visual Basic .NET:

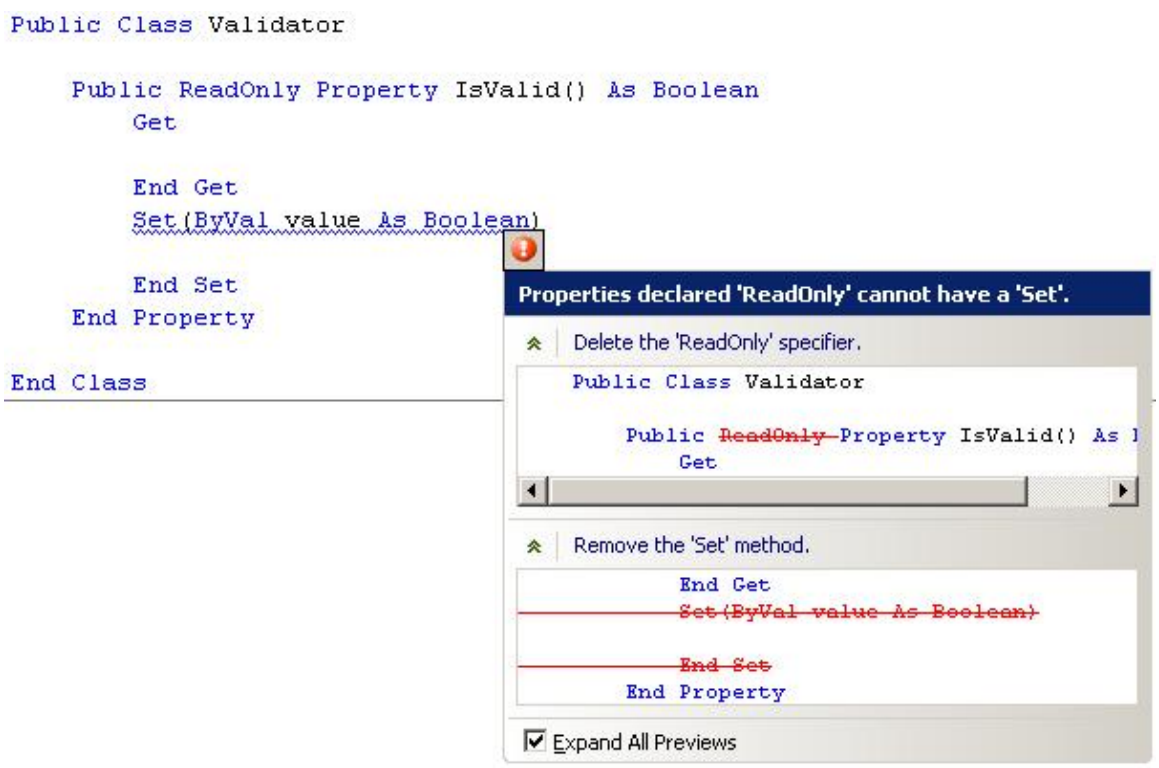


Imagem 4.4 – Auxiliar que nos sugere as possibilidades que existem para ajustar o código

Agora, problemas de lógica dificilmente você detecta durante esse modo. Somente quando a aplicação for executada e você ver que o resultado não é o esperado é que você detecta um possível problema mas, não pode mudar nenhuma linha de código neste momento. Neste momento é que aparece um novo modo, que chamamos de *break mode*. Esse modo permite-nos para a aplicação em um determinado ponto para que seja possível acompanhar a executação via linha de código e, conseqüentemente, analisar tudo o que acontece nos "bastidores" da aplicação.

Para entrar neste modo, você precisa colocar um breakpoint na linha que deseja que o depurador para que assim você possa acompanhar. Através da tecla F5 ou mesmo via menu *Debug*, *Start Debbuing* você também pode iniciar a aplicação anexando a mesma, o depurador. Com a aplicação neste modo, você já conseguirá averiguar o código que escreveu e acompanhar a execução passo à passo do mesmo.

Nas versões anteriores do Visual Studio .NET tínhamos os *data tips*. Os data tips serviam para quando, em *break mode*, passávamos o mouse por cima de uma tipo simples (inteiro, string, double, etc.) ele já exibia o valor da variável. Já o Visual Studio .NET 2005, os *data tips* ainda existem, mas agora muito mais poderosos, pois permitem também a visualização de objetos mais complexos, podendo visualizar todas as suas propriedades. Já no caso das coleções, os elementos também podem ser visualizados simplesmente passando o cursor do mouse em cima, assim como é mostrado na imagem abaixo:

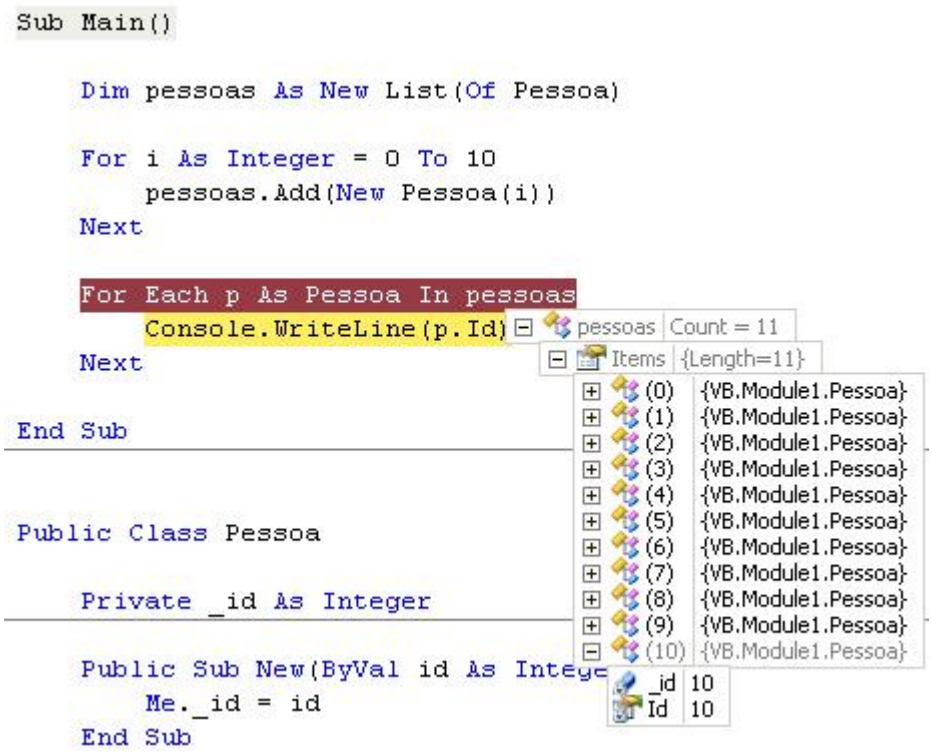


Imagem 4.5 – Data tips para objetos complexos

Criando um Debug Visualizer

A Microsoft incluiu na versão 2.0 do .NET Framework uma possibilidade de criar um *debugger* customizado para um determinado objeto que temos. Isso permite-nos criar uma interface mais amigável em relação à qual é fornecida quando utilizamos o debug do Visual Studio .NET 2005. Essa técnica é chamada de *Debugger Visualizer*. Você pode escrever esse visualizador para qualquer objeto da sua aplicação, com excessão de um *Object* ou *Array*.

A arquitetura do *debugger* está dividida em duas partes:

- O *debugger side* corre dentro do debugger do Visual Studio .NET, podendo ser criado e exibido uma interface para seu visualizador.
- O *debuggee side* corre dentro do processo que o Visual Studio .NET está depurando.

O objeto que você está querendo visualizar (uma *string* ou *Image*, por exemplo) existe dentro do processo *debuggee*. Logo, o *debuggee* side deve mandar os dados para o debugger side que, por sua vez, exibe o objeto para que o desenvolvedor possa depurar. Essa visualização é criada por nós que, ainda nesse artigo, veremos como criá-la.

O *debugger* side recebe o objeto que será visualizado através de um object provider, o qual implementa uma *Interface* chamada **IVisualizerObjectProvider**. O debuggee side

envia o objeto através de um *object source*, o qual deriva de uma classe chamada **VisualizerObjectSource**. O *object provider* também devolve o objeto para o *object source*, permitindo assim, além de exibir o objeto, editá-lo no visualizador (se assim desejar) e devolvê-lo para a aplicação. Essa comunicação é efetuada através de um objeto do tipo **Stream**.

Para criarmos este visualizador, primeiramente precisamos marcar a classe como sendo uma classe de **DebuggerVisualizer** e, para isso, é fornecido um atributo chamado **DebuggerVisualizer** (usado a nível de *Assembly* ou classe) para definí-la como um visualizador. Além disso, a classe deverá obrigatoriamente herdar de uma classe abstrata chamada **DialogDebuggerVisualizer** e implementar o método chamado *Show* para customizar a visualização.

O atributo **DebuggerVisualizer**, contido dentro do *namespace* **System.Diagnostics**, fornece mais alguns parâmetros para a configuração do visualizador. O primeiro parâmetro é tipo, ou seja, a classe derivada **DialogDebuggerVisualizer** que é o nosso visualizador efetivamente; já o segundo especifica o tipo de objeto que fará a comunicação entre o *debugger side* e o *debuggee side* e, se não informado, um padrão será utilizado. O restante, chamado de "*Named Parameters*", você deve especificar o tipo de objeto que o visualizador irá trabalhar (uma *string* ou *Image*, por exemplo) e no outro, é o nome que aparecerá dentro do Visual Studio .NET 2005, para permitir visualização. Para ilustrar, veremos abaixo o código que cria o visualizador:

VB.NET

```
Imports System
Imports Microsoft.VisualStudio.DebuggerVisualizers
Imports System.Windows.Forms
Imports System.Drawing
Imports System.Diagnostics

<_
    Assembly:
DebuggerVisualizer(GetType(DebugTools.ImageVisualizer), _
    GetType(VisualizerObjectSource), _
    Target := GetType(System.Drawing.Image), _
    Description := "Image Visualizer") _
>
Namespace DebugTools
    Public Class ImageVisualizer
        Inherits DialogDebuggerVisualizer

        Protected Overrides Sub Show(IDialogVisualizerService
windowService, _
            IVisualizerObjectProvider objectProvider)

            Image data = DirectCast(objectProvider.GetObject(),
Image)
```

```
            Using(ImageVisualizerForm        f        =        New
ImageVisualizerForm())
                f.Image = data
                windowService.ShowDialog(f)
            End Using
        End Sub

    End Class
End Namespace
```

C#

```
using System;
using Microsoft.VisualStudio.DebuggerVisualizers;
using System.Windows.Forms;
using System.Drawing;
using System.Diagnostics;

[
    assembly:
DebuggerVisualizer(typeof(DebugTools.ImageVisualizer),
    typeof(VisualizerObjectSource),
    Target = typeof(System.Drawing.Image),
    Description = "Image Visualizer")
]
namespace DebugTools
{
    public class ImageVisualizer : DialogDebuggerVisualizer
    {
        protected   override   void   Show(IDialogVisualizerService
windowService,
            IVisualizerObjectProvider objectProvider)
        {
            Image data = (Image)objectProvider.GetObject();

            using(ImageVisualizerForm        f        =        new
ImageVisualizerForm())
            {
                f.Image = data;
                windowService.ShowDialog(f);
            }
        }
    }
}
```

13

Note que foi criado um formulário chamado *ImageVisualizerForm*. Este formulário é encarregado de receber a imagem e exibí-la. Não há segredos nele, ou seja, apenas foi criada uma propriedade chamada *Image* que recebe a imagem vinda do *object provider*. O primeiro parâmetro do método *Show*, chamado *windowService* do tipo

IDialogVisualizerService, fornece métodos para que o visualizador possa estar exibindo formulários, controles e janelas de diálogo.

Se você quiser que o seu visualizador edite o objeto e o devolva para a aplicação, terá que sobrescrever os métodos *TransferData* ou *CreateReplacementObject* da classe **VisualizerObjectSource**.

Finalmente, para que possamos utilizar o visualizador dentro do Visual Studio .NET 2005 teremos que compilar o projeto e, com a DLL gerada, colocá-la dentro do diretório *<Diretorio VS.NET>\Common7\Packages\Debugger\Visualizers*. Quando abrir o Visual Studio, já terá acesso ao visualizador, assim como é mostrado através da imagem abaixo:

Imagem 4.6 – Utilizando Debug Visualizer customizado

Ainda existem outros atributos como o **DebuggerVisualizer**. Da mesma forma que o visualizador,

Ainda dentro do *namespace* **System.Diagnostics** temos outras classes importantes que auxiliar no processo de depuração de código. Essas classes são: **Debugger**, **StackFrame** e **StackTrace**. A primeira delas, **Debugger**, trata-se de uma classe que permite, programaticamente, efetuar a comunicação com o debugger do Visual Studio .NET 2005. Essa classe é comumente utilizada quando você precisa depurar uma seção de código que já foi executada por algum outro processo, ou ainda, quando esse código é disparado sem antes mesmo da aplicação inicializar. Essa classe alguns membros estáticos que veremos a seguir as suas funcionalidades:

Membro	Descrição
IsAttached	Retorna um valor booleano indicando se *debugger* está ou não anexado ao processo.
Break	Quando chamado este método, ele gera um *breakpoint* e, conseqüentemente, a aplicação entra em *break mode* e assim você consegue depurar o código.

IsLogging	Indica se o *log* está ou não habilitado para o *debugger*.
Launch	Quando o *debugger* não estiver anexado ao processo, você pode invocar esse método para lançar o *debugger* e anexá-lo ao processo.
Log	Insere uma mensagem no *debugger* corrente.

A classe **StackFrame**, apesar de ser pública, é utilizada pela infraestrutura do .NET Framework e não é indicado utilizá-la diretamente no seu código. *Stack frame* trata-se da representação física de uma chamada à alguma função dentro da *thread* corrente. Esse objeto é criado e colocado dentro da pilha de chamadas de funções que são realizadas em toda a execução da *thread*.

A classe **StackTrace** é basicamente uma coleção de **StackFrames**. A classe **Exception** fornece uma propriedade chamada *StackTrace* do tipo *string*. Quando alguma exceção acontece no código, essa propriedade retornará todas as chamadas realizadas as funções até que onde o problema aconteceu. Isso ajudará imensamente você a detectar onde o problema realmente se encontra.

Janelas de depuração

Quando iniciamos o depurador do Visual Studio .NET, várias janelas ficam disponíveis para nos auxiliar durante o processo de depuração. Essas janelas ficam disponíveis no menu Debug, Windows. Cada uma delas fornece um funcionalidade diferente, a seguir:

Janela	**Descrição**
Breakpoint	Exibe uma janela onde você pode visualizar todos os *breakpoint* definidos na aplicação e também fornece recursos para excluí-los ou desabilitá-los.
Output	Exibe informações de status para várias features da IDE do Visual Studio .NET.
Script Explorer	Exibe uma lista de arquivos de scripts que estão atualmente carregados no programa que você está depurando.
Watch	Cria uma lista customizada de variáveis e expressões que deseja monitorar.
Autos	Visualiza todas as variáveis dentro bloco corrente e de três blocos antes e três blocos depois do bloco corrente.
Locals	Permite visualizar e modificar o valor das variáveis locais.
Immediate	Utilizado para depurar e avaliar expressões, executar código, imprimir valores de variáveis, entre outros, a partir de linha de comando.
Call Stack	Visualiza todo o histórico de chamadas das linhas de código que estão sendo depuradas.
Threads	Exibe e controla as *threads* do programa que está sendo depurado.
Modules	Exibe uma lista de módulos (DLL ou EXE) que estão sendo utilizadas pelo programa que está sendo depurado, mostrando informações relevantes sobre cada uma delas.

Processes	Exibe uma lista com todos os processos que você tem anexado ou lançado a partir do Visual Studio .NET.
Memory	Disponibiliza informações e permite a visualização do espaço de memória que está sendo utilizado pela sua aplicação.
Disassembly	Esta janela mostra o código assembly correspondente ao código, criado pelo compilador.

Tracing

Tracing é a forma que temos para monitorar a execução da aplicação enquanto ela está rodando. Em tempo de desenvolvimento, você pode adicionar instrumentações de *tracing* e *debugging* para a aplicação .NET e, poderá utilizá-la durante o processo de desenvolvimento e também depois de distribuído. Para aplicações Windows, o *namespace* **System.Diagnostics** fornece duas classes para esse tipo de monitoramento: **Trace** e **Debug**. Já quando estamos falando de aplicações Web, temos a classe **TraceContext** dentro do *namespace* **System.Web**, que nada mais é que, entre outras funcionalidades específicas para o contexto, um *wrapper* para a classe **Trace**.

Basicamente as classes **Trace** e **Debug** possuem as mesmas finalidades. A única diferença significativa entre as classes é que primeira delas, a **Trace**, é compilada por padrão para dentro do *Assembly* e a classe **Debug** não. Enquanto estamos desenvolvendo a aplicação, tanto as informações de **Trace** e **Debug** são exibidas na janela *Output* do Visual Studio .NET. Para habilitar essas funcionalidades para elas serem distribuídas juntamente com o *Assembly*, você deve compilar a aplicação com a diretiva *TRACE* ou *DEBUG*. Para habilitar ou desabilitar esses recursos você tem duas formas: ou via IDE ou via linha de comando:

1. Via IDE:
    a. **Visual Basic .NET:** Propriedades do Projeto → Aba Compile → Advanced Compile Options → Compilation Constants.
    b. **Visual C#:** Propriedades do Projeto → Aba Build → General.
2. Via linha de comando:
    a. **Visual Basic .NET:** vbc /r:App.dll /d:TRACE=TRUE /d:DEBUG=FALSE Main.vb
    b. **Visual C#:** csc /r:App.dll /d:TRACE /d:DEBUG=FALSE Main.cs

O uso destas classes é bastante simples, mas antes de visualizar como utilizá-las, vamos analisar os métodos estáticos que elas fornecem e para que serve cada um deles. Logo em seguida, temos um trecho de código que basicamente mostra os métodos sendo invocados a partir de suas respectivas classes.

Método	**Descrição**
Assert	Dado uma condição, ele analisa se a mesma é ou não falsa. Se for, a mensagem especificada é armazenada e, se estiver rodando uma aplicação que possui uma interface gráfica, uma caixa de diálogo é

	exibida com mensagem.
WriteIf	Dado uma condição, ele analisa se a mesma é ou não verdadeira. Se for, a mensagem especificada é armazenada.
Fail	Grava a mensagem especificada e exibe uma caixa de diálogo é exibida com a mesma mensagem.
Write	Simplesmente salva a mensagem.
WriteLine	Salva mensagem, adicionando uma quebra de linha.
WriteLineIf	Dado uma condição, ele analisa se a mesma é ou não verdadeira. Se for, a mensagem especificada é armazenada juntamente com uma quebra de linha.

VB.NET

```
Imports System.Diagnostics

Trace.Assert(1 > 2, "1 nunca será maior que 2")
Debug.Assert(1 > 2, "1 nunca será maior que 2")

Trace.WriteIf(1 = 1, "1 é igual a 1")
Debug.WriteIf(1 = 1, "1 é igual a 1")

Trace.Fail("Algo aconteceu.")
Debug.Fail("Algo aconteceu.")

Trace.Write("Um valor para logar.")
Debug.Write("Um valor para logar.")

Trace.WriteLine("Um valor para logar c/ quebra de linha.")
Debug.WriteLine("Um valor para logar c/ quebra de linha.")

Trace.WriteLineIf(1 = 1, "1 é igual a 1 c/ quebra de linha.")
Debug.WriteLineIf(1 = 1, "1 é igual a 1 c/ quebra de linha.")
```

C#

```
using System.Diagnostics;

Trace.Assert(1 > 2, "1 nunca será maior que 2");
Debug.Assert(1 > 2, "1 nunca será maior que 2");

Trace.WriteIf(1 = 1, "1 é igual a 1");
Debug.WriteIf(1 = 1, "1 é igual a 1");

Trace.Fail("Algo aconteceu.");
Debug.Fail("Algo aconteceu.");

Trace.Write("Um valor para logar.");
Debug.Write("Um valor para logar.");

Trace.WriteLine("Um valor para logar c/ quebra de linha.");
```

```
Debug.WriteLine("Um valor para logar c/ quebra de linha.");

Trace.WriteLineIf(1 = 1, "1 é igual a 1 c/ quebra de linha.");
Debug.WriteLineIf(1 = 1, "1 é igual a 1 c/ quebra de linha.");
```

Até o momento essas configurações nos atendem. Mas nem sempre temos o Visual Studio .NET e sua janela de *Output* nos clientes que iremos instalar a aplicação. Seria muito mais conveniente neste caso, podemos persistir tais informações em arquivos textos ou até mesmo no *Event Log* do Windows, se assim desejarmos. Além disso, pode surgir a necessidade de filtrar qual tipo de mensagem desejamos persistir e, nos exemplos acima tudo, sem excessão, é salvo. É neste momento que entra em ação outros objetos que são utilizados para fornecer uma maior flexibilidade para configurar e dar manutenção no *tracing* da aplicação. Nesta seção veremos os seguintes elementos: **TraceSwitch**, **TraceListener** e **TraceSource**.

TraceSwitch

Switches permitem você habilitar, desabilitar e filtrar as mensagens de *tracing*, determinando se elas devem ou não serem persistidas. São objetos que estão acessíveis via código, mas que também podem ser configurados via arquivos de configuração, o que dá uma maior flexibilidade, já que possíveis alterações não exigirá que se recompile o programa.

Atualmente existem três tipos de *switches* fornecidos pelo .NET Framework: **BooleanSwitch**, **TraceSwitch** e **SourceSwitch**. O primeiro deles possibilita ligar e desligar o *tracing*. Já os dois últimos, permitem você habilitar ou desabilitar o *switch* para um determinado nível. Esse *switch* baseia-se nos seguintes níveis, disponibilizados pelo enumerador **TraceLevel**: *Error*, *Warning*, *Info* e *Verbose*. São esses níveis que são utilizados como filtros para configurarmos dentro da aplicação. Podemos em algum momento, queremos filtrar somente as mensagens de nível *Error* e, mais tarde, somente o que for *Warning*. Através da tabela abaixo conseguimos visualizar de forma tabular o que cada um os níveis possibilita:

Item	Constante	Tipo de Mensagem
Off	0	Nenhuma.
Error	1	Somente mensagens de erro.
Warning	2	Mensagens de aviso e mensagens de erro.
Info	3	Mensagens informativas, de aviso e erro.
Verbose	4	Mensagens *verbose*, informativas, de aviso e erro. *Verbose*: exibe toda e qualquer mensagem de *tracing* e *debugging*.

Inicialmente vamos analisar a implementação de código para o *switch* **BooleanSwitch** onde, via arquivo de configuração, vamos ver como devemos proceder para habilitar e desabilitar o *switch*, sem a necessidade de recompilar o código. Inicialmente, analisaremos a configuração que deve ser realizada no arquivo de configuração da aplicação para suportar o *swtich*. O código a seguir é idêntico para qualquer uma das linguagens:

```
App.Config
<configuration>
  <system.diagnostics>
    <switches>
      <add name="ModuloDados" value="0"/>
    </switches>
  </system.diagnostics>
</configuration>
```

No atributo *name* definimos o nome do *switch*. Esse nome é o mesmo que deve ser referenciado dentro da aplicação, para vincular o *switch* ao arquivo de configuração. O atributo *value* indica se o *switch* está ou não habilitado, ou seja, 0 (zero) está desabilitado e 1 (um) está habilitado. O código abaixo como fazer o vinculo entre o arquivo de configuração e a aplicação e, reparem que o mesmo nome é passado para o construtor da classe **BooleanSwitch**:

```
VB.NET
Imports System.Diagnostics

Private dataSwitch As New _
    BooleanSwitch("ModuloDados", "Acesso a dados.")

Sub Main()
    If dataSwitch.Enabled Then
        Console.WriteLine("Habilitado.")
    Else
        Console.WriteLine("Não habilitado.")
    End If
End Sub

C#
using System.Diagnostics;

private static BooleanSwitch dataSwitch =
    new BooleanSwitch("ModuloDados", "Acesso a dados.");

static void Main(string[] args)
{
```

```
    if (dataSwitch.Enabled)
    {
        Console.WriteLine("Habilitado.");
    }
    else
    {
        Console.WriteLine("Não habilitado.");
    }
}
```

Já a utilização do **TraceSwitch** também é bem semelhante, pois também permite a configuração via *App.Config*. Neste caso, no arquivo de configuração vamos informar qual o tipo de mensagem de *tracing* que desejamos que seja persistida, filtrando a partir de nível de mensagem, mais precisamente, através do enumerador **TraceLevel**, qual já analisamos mais acima. Essa classe possui quatro propriedades, sendo uma para cada nível do enumerador **TraceLevel**: *TraceError*, *TraceInfo*, *TraceVerbose* e *TraceWarning*, onde cada uma retorna um valor booleano indicando se o *switch* pode ou não persistir informações de um determinado nível.

Portanto, no arquivo de configuração devemos informar um dos itens do enumerador, podendo ser a constante (número inteiro) ou o "alias". O código abaixo exemplifica o uso do *switch* **TraceSwitch**:

App.Config

```
<configuration>
  <system.diagnostics>
    <switches>
      <add name="ModuloDados" value="Warning"/>
    </switches>
  </system.diagnostics>
</configuration>
```

VB.NET

```
Imports System.Diagnostics

Private dataSwitch As New _
    TraceSwitch("ModuloDados", "Acesso a dados.")

Sub Main()
    Console.WriteLine("Error: {0}", dataSwitch.TraceError)
    Console.WriteLine("Warning: {0}", dataSwitch.TraceWarning)
End Sub
```

C#

```
using System.Diagnostics;

private static TraceSwitch dataSwitch =
    new TraceSwitch("ModuloDados", "Acesso a dados.");

static void Main(string[] args)
{
    Console.WriteLine("Error: {0}", dataSwitch.TraceError);
    Console.WriteLine("Warning: {0}", dataSwitch.TraceWarning);
}
```

Se repararem no atributo *value* do arquivo de configuração, temos ali definido *Warning* (podendo ser a constante 2 que teria o mesmo efeito). Como esses itens são acumulativos, o *Warning* irá gravar tanto informações de erros quanto de avisos.

O último, porém não menos importante, é o **SourceSwitch**. Trata-se de uma nova classe que foi incluída na versão 2.0 do .NET Framework. Esse *switch* é utilizado em conjunto com o objeto **TraceSource**, qual veremos mais abaixo.

A classe **SourceSwitch** possui uma propriedade importante chamada *Level*. Essa propriedade recebe um valor fornecido pelo enumerador **SourceLevels**. Os valores definidos por esse enumerador identificará quais eventos serão capturados pelo *tracing*. Veremos mais detalhes sobre esse *switch* mais abaixo, quando abordarmos a respeito da classe **TraceSource**.

TraceListener

Logo no começo desta seção, vimos que todas as informações depositadas através da classe **Trace** e também da classe **Debug** são enviadas e exibidas na janela *Output* do Visual Studio .NET. Isso ajuda quando estamos em tempo de desenvolvimento, mas não resolve quando a aplicação é instalada no cliente. É necessário definirmos algum lugar que permita persistir fisicamente os dados para uma posterior análise.

Felizmente temos a nossa disposição os *listeners*. Os *listeners* são responsáveis por coletar, armazenar e direcionar as informações de *tracing*. Tanto a classe **Trace** quanto a classe **Debug** possuem uma propriedade chamada *Listeners* que expõem uma coleção do tipo **TraceListenerCollection**, que permite adicionarmos quantos *listeners* forem necessários. Sendo assim, podemos ter as mensagens de *tracing* sendo persistidas de diversas formas. Atualmente dentro do .NET Framework temos alguns *listeners* a nossa disposição. Todos os listeners herdam diretamente da classe abstrata **TraceListener**. Abaixo temos uma tabela que descreve todos os *listeners* que estão contidos dentro do .NET Framework:

Listener	**Descrição**
TextWriterTraceListener	Redireciona toda a saída para um *stream*, persistindo o

	conteúdo dentro de um arquivo texto convencional
EventLogTraceListener	Redireciona toda a saída para o *Event Log* do Windows.
DefaultTraceListener	Esse é o *listener* padrão que é adicionado a classe **Trace** e **Debug**. Os métodos *Write* e *WriteLine* enviam o conteúdo para a janela *Output* do Visual Studio .NET e também para o método *Log* do classe **Debugger** (discutida mais acima).
ConsoleTraceListener	Redireciona toda a saída para a *console*.
DelimitedListTraceListener	Redireciona toda a saída para um *stream*, persistindo o conteúdo dentro de um arquivo texto mas, neste caso, utilizando um delimitador. O delimitador pode ser definido através da propriedade *Delimiter*.
XmlWriterTraceListener	Redireciona toda a saída para um *stream*, persistindo o conteúdo dentro de um arquivo em formato XML.

Todos os listeners podem ser configurados via código (Visual Basic .NET/Visual C#) ou, como já era de se esperar, via arquivo de configuração. Como temos a propriedade *Listeners*, que expõe uma coleção, podemos adicionar quantos quisermos, ou seja, podemos ter as informações de *tracing* persistidas em arquivos XML e arquivos textos convencionais.

Nota: Todos os *listeners* possuem um método chamado *Flush* e outro chamado *Close*. Quando você invoca os métodos *WriteXXX* na verdade as informações estão sendo armazenadas na memória. Para você persistir os dados fisicamente, então terá que invocar o método *Flush* que tem a finalidade de gravar os dados em disco para todos os *listeners* ou o método *Close* que, além de persistir os dados para todos os *listeners*, também os fecha.

Finalmente, vamos analisar a implementação dos listeners em código e, em seguida, via arquivo de configuração. Reparem que o método *Clear* da classe **Trace** se encarrega de limpar todos os *listeners* existentes na coleção e, conseqüentemente, o *listener* padrão:

```
VB.NET
Imports System.Diagnostics

Trace.Listeners.Clear()
Trace.Listeners.Add(New XmlWriterTraceListener("Log.xml"))
Trace.Listeners.Add(New TextWriterTraceListener("Log.txt"))

Trace.WriteLine("Teste de Tracing", "Testes")
Trace.WriteLine("Adicionando uma nova entrada", "Testes")

Trace.Flush()
```

```
C#
using System.Diagnostics;

Trace.Listeners.Clear();
Trace.Listeners.Add(new XmlWriterTraceListener("Log.xml"));
Trace.Listeners.Add(new TextWriterTraceListener("Log.txt"));

Trace.WriteLine("Teste de Tracing", "Testes");
Trace.WriteLine("Adicionando uma nova entrada", "Testes");

Trace.Flush();
```

Já abaixo temos a configuração dos *listeners* via arquivo de configuração, o que da uma maior flexibilidade, já que podemos adicioná-los ou removê-los, *plug and play*. Lembrando que a configuração é válida para qualquer uma das linguagens (Visual Basic .NET ou Visual C#):

```
App.Config
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="4">
      <listeners>
        <remove name="Default" />
        <add name="txtListener"
             type="System.Diagnostics.TextWriterTraceListener"
             initializeData="Log.txt" />
        <add name="xmlListener"
             type="System.Diagnostics.XmlWriterTraceListener"
             initializeData="Log.xml" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

TraceSource

Esta classe foi incluída a partir da versão 2.0 do .NET Framework. Em projetos com muitos componentes, seria interessante termos um *tracing* diferenciado para cada um deles, o que ficaria muito mais simples de gerenciar e também uma melhor forma de organizar. Com essa necessidade, sugiu a classe **TraceSource**, que permite configurarmos as informações necessárias de *tracing*, como por exemplo os *listeners*, *switches*, etc..

O **TraceSource** permite montarmos toda a configuração necessária para cada seção ou componente do código que estamos desenvolvendo e, como já é previsto, além da

codificação via Visual Basic .NET ou Visual C#, permite também a configuração via *App.Config*, mas antes de entendermos a sua implementação, vamos analisar algums membros que essa classe fornece para a manipulação do *tracing*:

Membro	**Descrição**
Construtor	Essa classe possui dois construtores. O primeiro deles aceita somente uma *string* indicando o nome do **TraceSource**. Já o segundo construtor, além desta *string*, aceita também um parâmetro do tipo **SourceLevels**. **SourceLevels** é um enumerador que especifica o nível das mensagens que serão filtradas pelo *switch*. Entre os possíveis valores temos: • **ActivityTracing:** Permite efetuar o *tracing* dos seguintes eventos: *Stop*, *Start*, *Suspend*, *Transfer* e *Resume*. • All: Permite o *tracing* de todos os eventos. • **Critical:** Permite somente eventos críticos. • **Error:** Permite eventos críticos e eventos de erros. • **Information:** Permite efetuar o *tracing* de eventos críticos, de erros, avisos e informações. • **Off:** Não captura nenhum evento. • **Verbose:** Permite efetuar o *tracing* de eventos críticos, de erros, avisos, informações e também de *verbose*. • **Warning:** Permite efetuar o tracing de eventos críticos, de erros e avisos.
Listeners	Retorna a coleção de *listeners*.
Switch	Expõe ou recebe um objeto **SourceSwitch**.
TraceData	Permite escrevemos dados (de objetos e variáveis) dentro dos *listeners*. Esse método recebe três parâmetros: o primeiro deles do tipo **TraceEventType**, que trata-se de um enumerador que define os seguintes valores: • **Critical:** Erro fatal. • **Error:** Erro que é possível ser recuperado. • **Information:** Informações adicionais. • **Resume:** Reiniciando uma operação; • **Start:** Iniciando uma operação. • **Stop:** Parando uma operação. • **Suspend:** Suspendendo uma operação. • **Transfer:** Transferência de controle para uma outra operação. • **Verbose:** *Tracing* de debug. • **Warning:** Problema não crítico. Já o segundo parâmetro do método é do tipo inteiro, identifica o

	evento e, finalmente o último parâmetro, do tipo **Object** que é o valor que será salvo pela switch.
TraceEvent	Possui os mesmos parâmetros do método anterior, com excessão do último, que aqui é do tipo *string*, que permite escrever uma mensagem dentro da coleção de *listeners*.
TraceInformation	Permite escrever uma informação adicional dentro da coleção de *listeners*.
TraceTransfer	Escreve uma mensagem de transferência de controle de operação.

Quando utilizamos os métodos *TraceData* e *TraceEvent* que esperam em seu primeiro parâmetro um valor fornecido pelo enumerador **TraceEventType**, o valor vai ser comparado com o valor especificado na propriedade *Level* da classe **SourceSwitch** através do enumerador **SourceLevels**. Veremos a partir do código abaixo a razão disso:

VB.NET
```
Imports System.Diagnostics

Sub Main()
    Dim source As New TraceSource("Trace - Componente 1")
    Dim defaultSwitch As New SourceSwitch("switch")
    defaultSwitch.Level = SourceLevels.Critical
    source.Switch = defaultSwitch
    source.Listeners.Add(New TextWriterTraceListener("Log.txt"))

    source.TraceEvent(TraceEventType.Critical,    12,    "Mensagem
crítica")
    source.TraceEvent(TraceEventType.Information,  22,  "Mensagem
de informação")
    source.Close()
End Sub
```

C#
```
using System.Diagnostics;

static void Main(string[] args)
{
    TraceSource source = new TraceSource("Trace - Componente 1");
    SourceSwitch defaultSwitch = new SourceSwitch("switch");
    defaultSwitch.Level = SourceLevels.Critical;
    source.Switch = defaultSwitch;
    source.Listeners.Add(new TextWriterTraceListener("Log.txt"));

    source.TraceEvent(TraceEventType.Critical,    12,    "Mensagem
crítica");
    source.TraceEvent(TraceEventType.Information,  22,  "Mensagem
de informação");
    source.Close();
```

```
}
```

Como podemos ver, definimos **SourceLevels.Critical** na propriedade *Level* da classe **SourceSwitch**. Em seguida, adicionamos a instância da classe **SourceSwitch** na propreidade *Switch* da instância da classe **TraceSource** onde também definimos o *listener* **TextWriterTraceListener** para persistir os dados em um arquivo texto chamado "Log.txt". Finalmente, invocamos o método *TraceEvent* para inserirmos uma nova entrada no *tracing*. O primeiro método, informamos como **TraceEventType** o valor *Critical*. Já na segunda vez que invocamos o mesmo método, informamos como **TraceEventType** o valor *Information*. Sendo assim, o segundo não será persistido, justamente porque o *switch* somente aceita, no mínimo, eventos do tipo *Critical*. Isso irá garantir que, desenvolvedores que utilizam o componente, não grave informações desnecessárias.

Finalmente, se quisermos configurar o mesmo **TraceSource**, só que agora dentro do arquivo de configuração, podemos proceder da seguinte forma:

```
App.Config
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Trace - Componente 1" switchName="switch">
        <listeners>
          <add
            name="myLocalListener"
            type="System.Diagnostics.TextWriterTraceListener"
            initializeData="Log.txt" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="switch" value="Information" />
    </switches>
  </system.diagnostics>
</configuration>
```

E para utilizá-lo dentro da aplicação, faz:

```
VB.NET
Imports System.Diagnostics

Dim source As New TraceSource("Trace - Componente 1")
```

```
source.TraceEvent(TraceEventType.Critical,      12,      "Mensagem
crítica")
source.TraceEvent(TraceEventType.Information,  22,  "Mensagem  de
informação")
source.Close()

C#
using System.Diagnostics;

TraceSource source = new TraceSource("Trace - Componente 1");

source.TraceEvent(TraceEventType.Critical,      12,      "Mensagem
crítica");
source.TraceEvent(TraceEventType.Information,  22,  "Mensagem  de
informação");
source.Close();
```

WMI – Windows Management Instrumentation

Windows Management Instrumentation (WMI) é parte do sistema operacional que fornece uma infraestrutura para controle, gerenciamento e informações para o gerenciamento do sistema operacional. Ele pode fornecer informações relevantes para monitoramento de aplicações e, conseqüentemente, reportar de forma amigável ao usuário.

O .NET Framework fornece uma variedade de classes que nos auxiliam para resgatar informações referentes ao sistema operacional. Essas classes estão disponíveis dentro do *namespace* **System.Management** que, é necessário fazer a referência à *System.Management.dll* na aplicação se desejar utilizar o WMI. Através da imagem abaixo é possível visualizar a arquitetura do WMI dentro do mundo .NET:

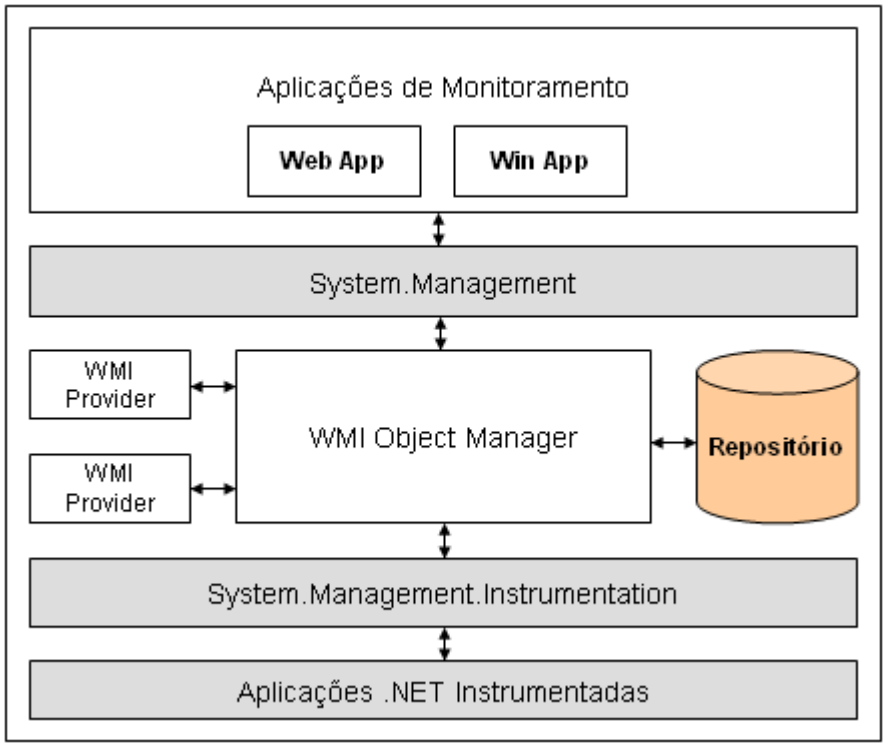


Imagem 4.7 – Arquitetura WMI dentro do .NET Framework.

O *namespace* **System.Management** fornece classes para manipularmos o WMI de uma forma simples e bastante produtiva, possibilitando escrever as queries que utilizamos para extrairmos informações do sistema operacional, dispositivos e outras aplicação, de uma forma semelhante ao que existe atualmente com a linguagem SQL. Dentro deste *namespace* temos a classe chamada **ManagementObjectSearcher**, que é uma das mais conhecidas e utilizadas. Essa classe é comumente utilizada para recuperar diversos tipos de informações, como por exemplo enumerar todos os drives de um determinado computador, os adaptadores de rede, processos que está sendo executados e muitos outros objetos dentro do sistema. Quando desejamos recuperar esse tipo de informação, utilizamos uma query, baseada em SQL. Conseqüentemente, isso irá possibilitar colocarmos critérios na query para extrairmos informações mais precisas, como por exemplo, exibir somente os processos que estão atualmente pausados, etc.. Essa query é representada por um objeto do tipo **ObjectQuery**. A imagem abaixo ilustra a hierarquia das queries disponíveis dentro deste *namespace*:

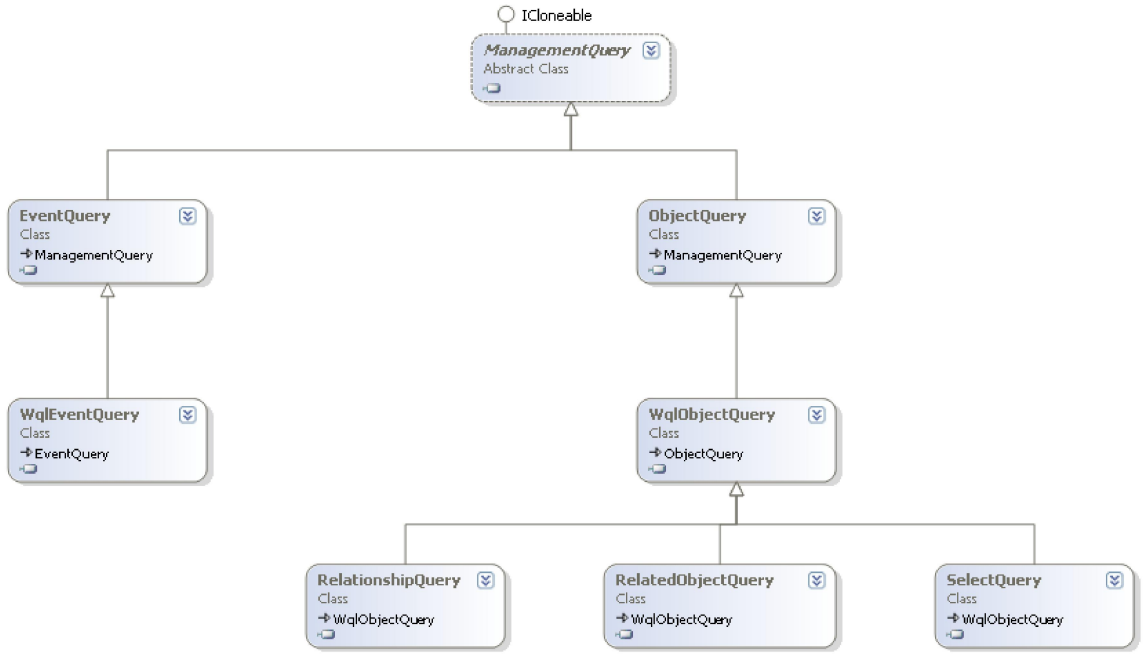


Imagem 4.8 – Estrutura das queries disponíveis para WMI

Como a classe **ManagementObjectSearcher** permite buscar objetos baseados em uma query, ela fornece um método chamado *Get* que retorna uma objeto (coleção) do tipo **ManagementObjectCollection**, contendo todos os objetos encontrados, sendo cada um deles representados por um objeto do tipo **ManagementObject**. Esse objeto representa os dados de um determinado objeto WMI que foi encontrado pela query. Para exemplificar a utilização do que vimos até o momento, o código abaixo exibe como recuperar todos os processos que estão sendo executados na máquina:

```
VB.NET
Imports System.Management

Dim searcher As New ManagementObjectSearcher("SELECT * FROM
Win32_Service WHERE Started = TRUE")

For Each service As ManagementObject In searcher.Get()
    Console.WriteLine("Serviço = " & service("Caption"))
Next

C#
using System.Management;

ManagementObjectSearcher searcher =
    new ManagementObjectSearcher("SELECT * FROM Win32_Service
WHERE Started = TRUE");

foreach (ManagementObject service in searcher.Get())
```

```
    Console.WriteLine("Serviço = " + service["Caption"]);
```

Um detalhe no código acima é com relação ao objeto **ManagementObject**. Ele possui uma propriedade *default* (*indexer* em Visual C#) que expõe uma coleção do tipo **PropertyDataCollection** contendo todas as propriedades de um determinado objeto. O único problema aqui é que, por questões de genericidade, você deverá passar uma *string* contendo o nome da propriedade do objeto que deseja recuperar. Isso vai implicar em saber as propriedades que o objeto possui pois, se passar um nome que não existe, uma excessão do tipo **ManagementException** será atirada.

Ainda dentro deste mesmo *namespace* temos a classe **ManagementEventWatcher**. Essa classe disponibiliza um evento chamdo *EventArrived* que é disparado quando um novo evento acontece. Esse evento, quando assinado, servem para notificar a aplicação que o utiliza para efetuar alguma ação customizada. Esse evento recebe em seu parâmetro um objeto do tipo **EventArrivedEventArgs**, contendo informações relacionadas ao evento que ocorreu. O código abaixo exibe como implementar essa classe para monitorar os eventos:

VB.NET

```
Imports System.Management

Sub Main()
    Dim watcher As New ManagementEventWatcher( _
        New WqlEventQuery( _
        "SELECT * FROM __InstanceCreationEvent " & _
        "WITHIN 1 WHERE TargetInstance ISA ""Win32_Process"""))

    AddHandler watcher.EventArrived, AddressOf EventArrived
    watcher.Start()
    System.Threading.Thread.Sleep(180000)
    watcher.Stop()
End Sub

Private Sub EventArrived(ByVal sender As Object, _
    ByVal e As EventArrivedEventArgs)

    Console.WriteLine("Processo criado  = " & _
        DirectCast(e.NewEvent("TargetInstance"),
ManagementBaseObject)("Caption"))
End Sub
```

C#

```
using System.Management;

static void Main(string[] args)
{
```

```
    ManagementEventWatcher watcher =
        new ManagementEventWatcher(new WqlEventQuery(
            @"SELECT * FROM __InstanceCreationEvent WITHIN 1 " +
            @"WHERE TargetInstance ISA ""Win32_Process"""));

    watcher.EventArrived +=
        new EventArrivedEventHandler(EventArrived);
    watcher.Start();
    System.Threading.Thread.Sleep(180000);
    watcher.Stop();
}

static void EventArrived(object sender, EventArrivedEventArgs e)
{
    Console.WriteLine("Processo     criado          =     "     +
((ManagementBaseObject)e.NewEvent["TargetInstance"])["Caption"]);
}
```

A partir do código acima, instanciamos a classe **ManagementEventWatcher** para monitorar os processos que estão sendo criados. Neste momento, temos uma nova classe chamada **WqlEventQuery**, que é uma classe que trabalha exclusivamente com eventos do WMI. A query informada no exemplo, monitora os processos que estão sendo criados. Em seguida, assinamos o evento *EventArrived* e, especificamos qual o procedimento que será disparado quando o evento ocorrer. Finalmente, chamamos o método *Start* do objeto **ManagementEventWatcher** para iniciar o monitoramento e, para fins de exemplo, suspendemos a execução da *thread* através do método *Sleep*, mantendo-a parada por 3 minutos. Nesse momento, todo e qualquer processo que for iniciado, como por exemplo, o bloco de notas, a calculadora do Windows, o evento *EventArrived* será disparado e, no caso acima, uma mensagem contendo o nome do processo recém iniciado, será escrita na tela.

Ainda temos um outro *namespace* chamado **System.Management.Instrumentation** que disponibiliza um conjunto de classes para que as aplicações, também construídas em .NET, possam passar para o *Object Manager* informações a seu respeito, informações quais, mais tarde serão acessadas pelas aplicações de monitoramento, via classes disponíveis dentro do *namespace* **System.Management**.

Basicamente para expor informações para o WMI para que aplicações de monitoramento consumam, é necessário criarmos uma classe que herde de uma classe abstrata chamada **BaseEvent**. Essa classe abstrata implementa a *Interface* **IEvent**, qual possui um método chamado *Fire*. Esse método delega a chamada para a classe **Instrumentation** que, efetivamente invocará o evento e, conseqüentemente, mandará para o WMI. Para já ilustrarmos, abaixo está somente a classe, que herda diretamente da classe **BaseEvent** e adiciona os propriedades necessárias que armazenarão as informações que devem ser expostas ao WMI e também as aplicações de monitoramento.

VB.NET

```
Imports System.Management.Instrumentation

Public Class EventoAppTeste
    Inherits BaseEvent

    Private _codigo As Integer
    Private _nome As String

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

    Public Property Codigo() As Integer
        Get
            Return Me._codigo
        End Get
        Set(ByVal value As Integer)
            Me._codigo = value
        End Set
    End Property

    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(ByVal value As String)
            Me._nome = value
        End Set
    End Property
End Class
```

C#

```
using System.Management.Instrumentation;

public class EventoAppTeste : BaseEvent
{
    private int _codigo;
    private string _nome;

    public EventoAppTeste(int codigo, string nome)
    {
        this._codigo = codigo;
        this._nome = nome;
    }

    public int Codigo
    {
```

```
        get
        {
            return this._codigo;
        }
        set
        {
            this._codigo = value;
        }
    }

    public string Nome
    {
        get
        {
            return this._codigo;
        }
        set
        {
            this._codigo = value;
        }
    }
}
```

A seguir, é necessário incluirmos dentro do *Assembly* que terá esse evento, um instalador para que o mesmo seja capaz de inserir dentro do WMI o tal evento. Para a criação deste instador, iremos criar uma classe que herda diretamente da classe **DefaultManagementProjectInstaller**. Essa classe herda da classe **Installer** e possui a implementação necessária para a instalação de um *Assembly* que possui informações de instrumentação – WMI. Somente a herança é necessária e nenhum método precisa ser sobrescrito. O código abaixo mostra como criá-la:

VB.NET

```
Imports System.ComponentModel
Imports System.Management.Instrumentation

<RunInstaller(True)> Public Class Instalador
    Inherits DefaultManagementProjectInstaller
End Class
```

C#

```
using System.ComponentModel;
using System.Management.Instrumentation;

[RunInstaller(true)]
public class Instalador : DefaultManagementProjectInstaller { }
```

Nota: Para saber mais sobre instaladores e sobre o atributo **RunInstallerAttribute**, consulte o *Capítulo 3*.

Ainda, neste mesmo *Assembly* é necessário adicionarmos um atributo chamado **InstrumentedAttribute** no arquivo *AssemblyInfo*. Esse atributo também está contido dentro do *namespace* **System.Management.Instrumentation** e indica que o *Assembly* possui informações de instrumentação. O trecho de código abaixo exibe a configuração deste atributo dentro do projeto onde o evento foi criado, informando em seu construtor o *namespace* que será utilizado pela instrumentação:

```
VB.NET
Imports System.Management.Instrumentation

<Assembly: Instrumented("Root/Default")>

C#
using System.Management.Instrumentation;

[assembly: Instrumented("Root/Default")]
```

Finalmente, para adicionarmos o evento dentro do WMI, é necessário instalarmos o *Assembly*, mais precisamente o instalador que está contido dentro dele. Para isso, temos duas possibilidades para instalá-lo:

1. Empacotar em um arquivo MSI
2. Rodar o utilitário installutil.exe

Para fins de exemplo, iremos optar pela segunda opção:

```
C:\>installutil C:\App.exe
...
...
...
The transacted install has completed.
```

Nota: Atente-se para abrir o Prompt do Visual Studio .NET ao invés do Prompt fornecido pelo Windows, para evitar escrever todo o caminho até o utilitário.

Quando o utilitário *installutil.exe* encontrar o instalador dos eventos de WMI dentro do *Assembly*, ele fará todo o trabalho necessário para adicioná-lo dentro da arquitetura do WMI e possibilitará outras aplicações a consumí-los. Uma vez criado os eventos dentro do WMI, o mesmo *Assembly* que o cria, geralmente é o mesmo que o invoca (via método *Fire*) para notificar o WMI. Sendo assim, para fazermos um teste, temos que criar

instâncias da classe *EventoAppTeste*, definirmos os valores das propriedades e invocar o método *Fire*. O código a seguir mostra um exemplo dentro de um laço *For* de 10 iterações, suspendendo a *thread* por 5 segundos:

```
VB.NET
For i As Integer = 0 To 10
    Dim ev As New EventoAppTeste(I, "Nome " + i.ToString())
    ev.Fire()
    System.Threading.Thread.Sleep(5000);
Next

C#
for (int i = 0; i < 10; i++)
{
    new EventoAppTeste(i, "Nome " + i.ToString()).Fire();
    System.Threading.Thread.Sleep(5000);
}
```

Esse laço fará com que ele mande 10 notificações do evento para o WMI e agora, compete a cada aplicação que quiser monitorar esses eventos, apanhá-los e exibir da forma que achar mais conveniente. No nosso caso, utilizaremos o mesmo exemplo que analisamos um pouco mais acima, quando utilizamos a classe **ManagementEventWatcher**, mudando ligeiramente a query para apenas recuperarmos os eventos do tipo *EventoAppTeste*. O códig abaixo mostra um exemplo na íntegra de um cliente que deseja recuperar um determinado tipo de evento:

```
VB.NET
Imports System.Management

Sub Main()
    Dim watcher As New ManagementEventWatcher("root/default", _
        "SELECT * FROM EventoAppTeste")

    AddHandler watcher.EventArrived, AddressOf EventArrived
    watcher.Start()
    System.Threading.Thread.Sleep(180000)
    watcher.Stop()
End Sub

Private Sub EventArrived(ByVal sender As Object, _
    ByVal e As EventArrivedEventArgs)

    Console.WriteLine(e.NewEvent.Properties("Codigo").Value)
    Console.WriteLine(e.NewEvent.Properties("Nome").Value)
End Sub
```

C#
```
using System.Management;

static void Main(string[] args)
{
    ManagementEventWatcher watcher =
        new ManagementEventWatcher("root/default",
            "SELECT * FROM EventoAppTeste");

    watcher.EventArrived +=
        new EventArrivedEventHandler(EventArrived);
    watcher.Start();
    System.Threading.Thread.Sleep(180000);
    watcher.Stop();
}

static void EventArrived(object sender, EventArrivedEventArgs e)
{
    Console.WriteLine(e.NewEvent.Properties["Codigo"].Value);
    Console.WriteLine(e.NewEvent.Properties["Nome"].Value);
}
```

A única mudança considerável neste código em relação ao que vimos um pouco mais acima, é com relação ao construtor da classe **ManagementEventWatcher**. No primeiro parâmetro, ele recebe uma string que identifica o escopo (*namespace*) que o watcher deverá monitorar; já o segundo parâmetro, é a query que vai definir qual evento deverá ser monitorado e, se repararem, a cláusula *FROM* recebe o nome do evento que criamos anteriormente, ou seja, *EventoAppTeste*.