# Arquitetura Plugável em .Net

## Construa aplicações em módulos com .NET


**Felipe Cembranelli**

*(felipe.cembranelli@sanmina-sci.com)*

é graduado em Ciências da Computação pela Unesp e pós-graduado em Telecomunicações. Atua há 11 anos na área de desenvolvimento e é certificado na plataforma Microsoft desde 2001. Atualmente trabalha na equipe .NET da Sanmina-SCI na região de Campinas.

**Gustavo Velasquez**

*(velasquez@email.com)*

é analista de sistemas graduado pela Universidade Federal de Uberlândia (UFU). 7 anos de experiência com tecnologias Borland/Microsoft e é certificado na plataforma Microsoft desde 2005. Atua como Software Analyst no BRDC (Brazil Research & Development Center) na multinacional Sanmina-SCI Corp/Campinas SP


Nos dias de hoje, existem diversos tipos de aplicações que sofrem uma quantidade razoável de mudanças em suas regras de negócio. Além disso, a adição de novas regras também é muito freqüente. Por este fato, torna-se cada vez mais necessária a construção de aplicações que possam ser atualizadas ou estendidas com facilidade e rapidez. Um desejo comum de muitos desenvolvedores é simplesmente criar novas partes de um sistema e fazer a publicação dessas partes sem a necessidade de recompilação do "core" da aplicação. Por exemplo, imagine que você simplesmente "copiasse" um *assembly* contendo as novas regras de negócio para uma determinada pasta, e a aplicação *host* conseguisse utilizar essa nova "peça" do sistema e aplicasse essas novas regras. Ou seja, desenhar uma aplicação de forma que as peças que a componha possam ser plugadas em tempo de execução.

Não estamos falando simplesmente em "quebrar a aplicação em camadas". Estamos falando em utilizar um componente sem a necessidade de uma "referência física" dentro do projeto que irá consumir este componente.

Ainda confuso? Vamos ver se com um exemplo as coisas ficam mais claras...

Neste artigo iremos construir uma pequena aplicação, utilizando *reflection* e alguns conceitos de *POO* (Programação orientada a objetos) para mostrar como é possível construir aplicações com uma "Arquitetura plugável" de maneira bem simples e com toda a potência, flexibilidade e desacoplamento que esta arquitetura proporciona.

## O Problema

A necessidade de uma arquitetura plugável pode ser identificada em diversos cenários e não é muito difícil de ser encontrada em um ambiente real dentro de uma empresa. Aqui neste artigo, para efeito de ilustração da técnica, iremos imaginar a seguinte situação: você é o arquiteto responsável por desenhar uma

solução para realizar a "integração" entre alguns sistemas de sua empresa.

A aplicação deverá, a princípio:

• Receber os dados de entrada a partir de um banco de dados ou de arquivos no formato "texto". Porém, diversos outros tipos de fontes de dados poderão "aparecer" com o decorrer do tempo, como por exemplo, receber os dados através de um *web services* ou através de *FTP* (*File Transfer Protocol*).

• Processar os dados de entrada, aplicar alguma lógica de negócio, que pode variar bastante, e gerar uma saída que seja entendida pelo segundo sistema.

• Enviar esses dados para o segundo sistema, gerando um novo arquivo ou mesmo enviando o resultado via e-mail para os envolvidos no processo. Mais uma vez, diversos outros tipos de saída poderão também surgir, como por exemplo, salvar os dados em outra base de dados ou enviá-los via FTP.

Como nós podemos ver, existe uma série de possibilidades relacionadas tanto à entrada de dados quanto à saída, dependendo dos sistemas que nós desejamos integrar. Além disso, seria interessante pensar em uma maneira de atualizar ou acrescentar uma nova "integração" de sistemas, sem a necessidade de deixar as demais "fora do ar".

## Desenhando a solução

A solução de exemplo será composta de uma parte "fixa" (*host*) e de pequenas peças plugáveis (*plug-ins*). Essas "peças" deveriam ser encaixadas dentro do *host* sem a necessidade de recompilação do mesmo e sem a necessidade de "referenciar" esse novo componente em "tempo de design". O ato de "encaixar" um *plug-in* significa somente copiar uma *DLL* (*dynamic link library*) para uma pasta específica e o host deverá ser capaz de reconhecer aquele novo *plug-in* e passar a utilizá-lo. Pode parecer meio complicado, mas tanto a idéia quanto a implementação são bastante simples.

Com a finalidade de simplificar, o *host* será implementado como uma aplicação "Console", mas em um ambiente real poderia ser implementado como um "*Windows Service*". Já os *plug-ins* serão construídos como projetos "*Class Library*".

**Listagem 1.** Interface

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Pluggable
{
    public interface IPluggable
    {
        /// <summary>
        /// Nome do plug-in
        /// </summary>
        string Name { get;}
        #region Estes métodos serão chamados pelo host e devem ser implementados pelo plug-in
        /// <summary>
        /// Obtém os dados da fonte e deixa disponível para processamento/transformação
        /// </summary>
        /// <returns></returns>
        string Import();
        /// <summary>
        /// Processa os dados e deixa disponível para envio/exportação
        /// </summary>
        /// <returns></returns>
        string Process();
        /// <summary>
        /// Exporta/envia os dados
        /// </summary>
        /// <returns></returns>
        string Export();
        #endregion
    }
}
```

## Definindo um contrato

A primeira coisa a se fazer é definir como o *host* deverá se comunicar com o *plug-in*, ou seja, a partir do momento em que o *plug-in* estiver disponível para ser utilizado, é preciso definir uma maneira do *host* utilizá-lo. Isso pode ser feito através da definição de uma "Interface". A interface funciona como um contrato, que estabelece como "as partes" deverão se comunicar.

Para isso, utilizando o *VS.NET 2005*, crie um projeto do tipo "*Class Library*" com o nome de "*PluggableBase*". Dentro da classe deste projeto, crie a interface, como na **Listagem 1**.

Com a definição desta interface, o que nós estamos fazendo é: dizendo que um *plug-in*, para ser utilizado pela aplicação *host*, deverá implementar a propriedade "*Name*" e os métodos "*Import*", "*Process*" e "*Export*". Essa é a única coisa que o *host* conhecerá do *plug-in*. Os detalhes do conteúdo desses métodos serão de responsabilidade do plug-in.

## Construindo alguns plug-ins de exemplo

O *plug-in* será o responsável pela implementação dos métodos da interface. No nosso exemplo, o primeiro plug-in será responsável por recuperar os últimos 10 pedidos da tabela "*Orders*" do banco de dados *Northwind*, irá "simular" um processamento (método "*Process*"), selecionando os pedidos considerados "especiais" e então irá enviar um email com a lista de pedidos processados (método "*Export*").

Já o segundo plug-in irá receber um arquivo "texto", irá novamente simular um processamento e então irá gerar um arquivo "Xml" como saída.

Para isso, crie um novo projeto "*Class Library*" no *VS.NET 2005* com o nome de "*Plugin1*" e renomeie a classe para "*Parser1*". Faça uma referência para o projeto da interface ("*PluggableBase*"). Dentro da classe "*Parser1*", faça a implementação da interface "*IPluggable*". Para isso, caso você esteja usando C#, você pode usar a opção de "*Refactor*" do *VS.NET* para simplificar seu trabalho.

Informe na frente do nome da classe "*Parser1*" o nome da interface que você deseja implementar (*IPluggable*) e clique com o botão direito sobre o nome da interface. Agora basta escolher a opção "*Implement Interface*" como mostrado na **Figura 1** e pronto, não esqueça de adicionar o *namespace Pluggable* na seção *unsing*.

Os métodos que deverão ser implementados serão automaticamente criados dentro da classe, como na **Listagem 2.**

Agora vamos alterar a classe, colocando código customizado para a propriedade "*Name*" e os métodos "*Import*", "*Process*"

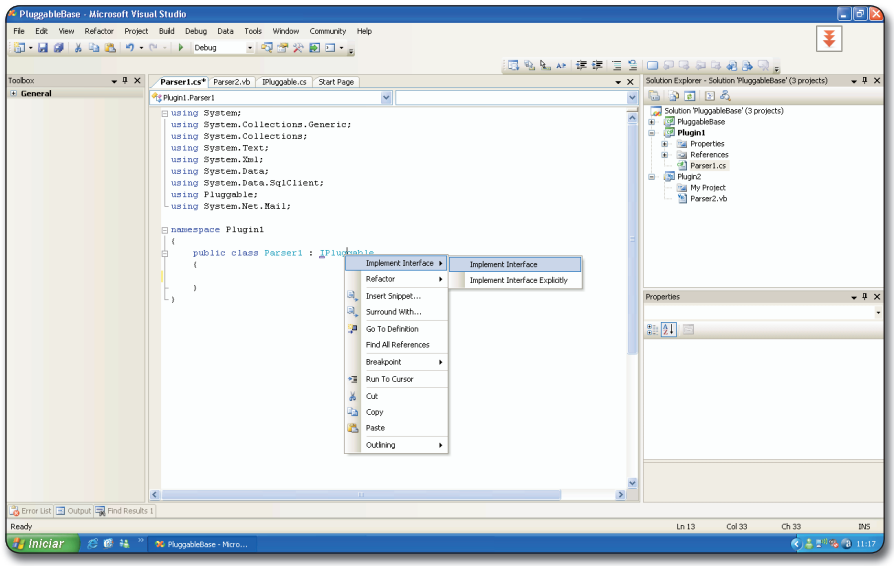


**Figura 1.** Implementado a interface IPluggable

**Listagem 2.** Plug-in1

```
using System;
using System.Collections.Generic;
using System.Text;
using Pluggable;
namespace Plugin1
{
    public class Parser1 : IPluggable
    {
        #region IPluggable Members
        public string Name
        {
            get { throw new Exception("The method or operation is not implemented."); }
        }
        public string Import()
        {
            throw new Exception("The method or operation is not implemented.");
        }
        public string Process()
        {
            throw new Exception("The method or operation is not implemented.");
        }
        public string Export()
        {
            throw new Exception("The method or operation is not implemented.");
        }
        #endregion
    }
}
```

**Listagem 3.** Métodos import, process e export

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
using System.Xml;
using System.Data;
using System.Data.SqlClient;
using Pluggable;
using System.Net.Mail;
namespace Plugin1
{
    public class Parser1 : IPluggable
    {
        #region Constantes que poderiam estar em banco ou arquivo de configuração
        const string CONNECTION_STRING = "Data Source=serverName;Initial Catalog=Northwind;User ID=sa";
        const string SQL_STATEMENT = "SELECT TOP 10 * FROM ORDERS ORDER BY 1 DESC";
        #endregion
        // membros privados
        private DataTable orders;
        private ArrayList specialOrders = new ArrayList();
        #region IPluggable Members
        string IPluggable.Name {
            get {return "Parser 1"; }
        }
        string IPluggable.Import()
        {
            // Recupera dados a partir do banco de dados
            SqlDataAdapter sqlAdp = new SqlDataAdapter(SQL_STATEMENT, CONNECTION_STRING);
            DataSet ds = new DataSet("Pedidos");
            sqlAdp.Fill(ds);
            this.orders = ds.Tables[0];
            return "Dados importados com sucesso.";
        }
```

e "*Export*", como mostrado na **Listagem 3**. Primeiro, criamos uma constante para armazenar a "string de conexão" que será usada para acessar o banco de dados *Northwind*. Criamos também uma constante para armazenar o comando SQL que será executado para extrair os dez últimos pedidos. Criamos o *DataTable* "*orders*" como membro privado da classe para armazenar os pedidos retornados do banco e o *ArrayList* "*specialOrders*" para armazenar os pedidos considerados especiais.

> **Nota**: os métodos *Process* e *Export* foram removidos da listagem por restrições de espaço e se encontram juntamente como o exemplo completo disponível para download.

Agora descreveremos a implementação da interface: A propriedade "*Name*" foi alterada para retornar o nome do *plug-in* ("*Parser 1*"). Essa propriedade será usada pelo host, somente para mostrar o nome do *plug-in* em execução. O Método "*Import*" foi implementado de forma a extrair os dez últimos pedidos da tabela "*orders*" do banco de dados de exemplo "*Northwind*" e armazená-los na *DataTable* "*orders*", equanto que o método "*Process*" irá simular algum processamento (no nosso exemplo, iremos conferir aos pedidos endereçados ao cliente "*Simons Bistro*" o status de "pedido especial") e finalmente o método "*Export*" irá verificar se existe algum pedido especial dentre os últimos pedidos importados e processados para que estes sejam enviados via email para o pessoal do estoque.

Claro que esta é uma situação hipotética levantada por nós. Você poderia implementar a importação de onde e como você queira, processar da maneira que você achar mais conveniente e também exportar da melhor maneira possível.

Até mesmo os passos poderiam ser diferentes. Você poderia, por exemplo, implementar um quarto passo se isto fizer sentido para todos os *plug-ins* que vão ser usados, ou então implementar um passo único para manter a coisa ainda mais flexível. O contrato é você quem define, só não vale quebrá-lo

depois de tudo funcionando. Agora iremos seguir passos semelhantes para o segundo plug-in, como mostrado na **Listagem 4**.

Nós iremos construir esse segundo *plug-in* em *VB.NET* para mostrar que o *host* realmente não está interessado nos detalhes de implementação, desde que os *plug-ins* respeitem a interface definida.

> **Nota**: os métodos *Process* e *Export* foram removidos da listagem por restrições de espaço e se encontram juntamente como o exemplo completo disponível para download.

Na **Listagem 4**, o método "*Import*" irá importar arquivos (em um formato inventado por nós, com extensão *.dat*) delimitados por um separador configurável (no caso, um *pipe* "|"), a partir de um diretório de importação (constante *IMPORT_PATH*) e colocá-los em um diretório de "*Archive*" (constante *ARCHIVE_PATH*). Depois irá criar uma lista de clientes baseado no conteúdo do arquivo. Para facilitar a manipulação das informações do arquivo depois de importado, criamos a classe interna "*Cliente*", com algumas propriedades ("*ID*", "*Nome*", "*Endereço*" e "*Idade*") que correspondem ao conteúdo do nosso arquivo fictício.

Esta lista de clientes será então processada (método "*Process*") para verificar a existência de clientes com idade para participar de uma determinada promoção, gerando assim uma lista contendo os clientes participantes desta promoção (*listaPromocao*).

Note que na exportação (método "*Export*"), decidimos simular a geração de conteúdo em *XML* em um diretório de saída (constante *EXPORT_PATH*). Para isso criamos o *dataTable* "*Promocao*", populamos ele a partir da lista *listaPromocao* e então usamos o método *WriteXml*() para gerar o arquivo *XML*. Mas você poderia gravar isto em um banco, enviar email para os clientes, etc.

Neste exemplo, as configurações são feitas baseadas em constantes. Em um

**Listagem 4**. Segundo plug-in

```
Imports System
Imports System.Collections.Generic
Imports System.Text
Imports System.Xml
Imports System.Data
Imports System.Data.SqlClient
Imports Pluggable
Public Class Plugin2
    Implements Pluggable.IPluggable
    'Constantes utilizadas
    Private Const IMPORT_PATH As String = "C:\Temp\PlugableFiles\Import\"
    Private Const EXPORT_PATH As String = "C:\Temp\PlugableFiles\Export\"
    Private Const ARCHIVE_PATH As String = "C:\Temp\PlugableFiles\Archive\"
    Private Const EXTENSAO As String = "*.dat"
    Private Const SEPARADOR As String = "|"
    Private Const IDADE_PROMOCAO As Integer = 18
    'Listas utilizadas
    Private listaClientes As List(Of Cliente) = New List(Of Cliente)
    Private listaPromocao As List(Of Cliente) = New List(Of Cliente)
    'Classe utilizada para manipular os dados de cliente
    Private Class Cliente
        Private _id As Integer
        Public ReadOnly Property ID() As Integer
            Get
                Return ID
            End Get
        End Property
        Private _nome As String
        Public ReadOnly Property Nome() As String
            Get
                Return _nome
            End Get
        End Propert
        Private _endereco As String
        Public ReadOnly Property Endereco() As String
            Get
                Return _endereco
            End Get
        End Property
        Private _idade As Integer
        Public ReadOnly Property Idade()
            Get
                Return _idade
            End Get
        End Property
        Public Sub New(ByVal id As Integer, ByVal nome As String, ByVal endereco As
          String, ByVal idade As Integer)
            Me._id = id
            Me._nome = nome
            Me._endereco = endereco
            Me._idade = idade
        End Sub

    End Class
    Public ReadOnly Property Name() As String Implements Pluggable.IPluggable.Name
        Get
            Return "Parser 2"           ' nome amigável para o plug-in
        End Get
    End Property

    'Recebe um arquivo em formato texto e preenche uma lista de objetos clientes
    Public Function Import() As String Implements Pluggable.IPluggable.Import
        Dim sr As System.IO.StreamReader
        Dim auxiliar As String
        Dim c As Cliente
        Dim di As New System.IO.DirectoryInfo(IMPORT_PATH)

        For Each fi As System.IO.FileInfo In di.GetFiles(EXTENSAO)
          sr = New System.IO.StreamReader(New System.IO.FileStream(fi.FullName, IO.FileMode.Open))
            While Not sr.EndOfStream()
                auxiliar = sr.ReadLine()
                Dim arr() As String = auxiliar.Split(SEPARADOR)
                'testando se achamos os 4 campos na linha do arquivo
                If arr.Length = 4 Then
                    c = New Cliente(CInt(arr(0)), arr(1), arr(2), CInt(arr(3)))
                    Me.listaClientes.Add(c)
                End If
            End While
            sr.Close()
            sr.Dispose()
            If Not System.IO.File.Exists(fi.FullName.Replace(IMPORT_PATH, ARCHIVE_PATH)) Then
                fi.MoveTo(ARCHIVE_PATH & fi.Name)
            End If
        Next
        If Me.listaClientes.Count > 0 Then
            Return "Arquivo importado com sucesso."
        Else
            Return "Sem dados para importar."
        End If

    End Function
```

**Listagem 5.** Método GetPlugins

```
public static List<IPluggable> GetPlugins()
        {
            List<IPluggable> pluginList = new List<IPluggable>();
            try
            {

                // Diretório onde deverão ser colocados os plug-ins
                string path = "C:\\temp\\plugins";

                // Procura por todas as "dlls" no caminho especificado
                foreach (string file in System.IO.Directory.GetFiles(path, "*.dll"))
                {

                    // Carrega o assembly
                    System.Reflection.Assembly assembly = System.Reflection.Assembly.LoadFrom(file);


                    // Para cada um dos tipos dentro do assembly
                    foreach (Type type in assembly.GetTypes())
                    {

                        // Verifica se a interface IPluggable é implementada
                        if (type.GetInterface(typeof(IPluggable).FullName) != null)
                        {

                            // Cria uma instância do plugin
                            IPluggable validPlugin = (IPluggable)Activator.CreateInstance(type);

                            // Se conseguiu criar, adiciona a lista de plugins
                            if (validPlugin != null)
                            {
                                pluginList.Add(validPlugin);
                            }
                        }
                    }
                }

                catch (Exception ex)
                {
                    throw ex;
                }
                return pluginList;
            }
```

**Listagem 6.** Main

```
        static void Main(string[] args)
        {
            System.Timers.Timer timer = new System.Timers.Timer(5000);
            timer.Elapsed += new System.Timers.ElapsedEventHandler(ExecutePlugins);
            timer.Start();

            Console.WriteLine("Aguardando execução de plug-ins...");
            Console.ReadKey();
        }
```

**Listagem 7.** Método Execute pluginins

```
private static void ExecutePlugins(object sender, System.Timers.ElapsedEventArgs e)
{
   foreach (IPluggable o in GetPlugins())
   {
                Console.WriteLine("-----------------------------------------------------");
                Console.WriteLine(DateTime.Now.ToString());
                Console.WriteLine("Plug-in sendo executado: " + o.Name);
                Console.WriteLine(o.Import());
                Console.WriteLine(o.Process());
                Console.WriteLine(o.Export());
                Console.
                WriteLine("-----------------------------------------------------");
    }

    Console.WriteLine("Aguardando próxima execução de plug-ins...");
}
```

cenário real onde a necessidade de modificações pode ser freqüente, uma boa prática seria configurar todos estes parâmetros em um arquivo de configuração externo ou mesmo em um banco de dados. Além disso, você poderia ter uma camada específica para acesso aos dados. Mas a idéia deste artigo é demonstrar o conceito de arquiteturas plugáveis e não a melhor forma de desenvolvimento multicamadas.

## Construindo a aplicação host

A aplicação *host* será a responsável por "monitorar" um determinado "repositório", que no nosso exemplo será uma pasta, carregar o *plug-in*, verificar se realmente o *plug-in* implementa a interface desejada e executá-lo. Para fazer isso, usaremos um recurso importante da plataforma .net o reflection.

## Plugando as partes : a mágica do "reflection"

Um dos pontos positivos do *Framework .net* e da *CLR* (*Common Language Runtime*) é a riqueza de informações disponíveis sobre os objetos (tipos) em execução. "*Reflection*" é o recurso que nos permite navegar e investigar esse tipo de informação e ele está disponível através das classes do *namespace System.Reflection*.

Para continuar com o nosso exemplo, crie um projeto do tipo "*Console Application*" com o nome "*Host*". A classe "*Program*" deste "console" deverá conter o método da **Listagem 5**, que será explicado mais a seguir.

Neste método recuperamos todos os arquivos com extensão "*dll*" da pasta especificada, utilizando o método *GetFiles* da classe *System.IO.Directory*. Para cada um dos arquivos encontrados, utilizando as classes do *namespace System.Reflection*, nós iremos "investigar" o *assembly*, verificando se o mesmo contém um tipo que implemente a interface desejada "*IPluggable*". Em caso afirmativo, uma instância do tipo é criada e a mesma é carregada na lista genérica "*pluginList*".

O resultado deste método é uma lista contendo instâncias de todos os plug-ins que deverão ser executados. Dentro do projeto "*Host*", altere o método "*Main*"

da classe "*Program*", colocando o código da **Listagem 6**.

Observe que estamos utilizando uma classe "*Timer*", para fazer com que a aplicação *host* acorde de tempos em tempos, carregue os *plug-ins* que estiverem disponíveis e os execute, chamando o método "*ExecutePlugins* ", os detalhes do método "*ExecutePlugins*" são mostrados na **Listagem 7**.

Aqui é onde de fato estamos realizando a chamada dos métodos expostos pelos *plug-ins* através da interface "*IPluggable*". A grande "sacada" deste processo é se preocupar apenas com o contrato de chamada dos métodos e não com o que está sendo executado de fato.

Repare que estamos chamando "*Import*", "*Process*" e "*Export*", porém não temos nenhum detalhe a respeito de qual instância estamos usando, ou mesmo do que o método está fazendo. A única coisa que sabemos até este momento (e é a única coisa que precisamos) é que todas as instâncias contidas na lista retornada por "*GetPluging*" assinaram o contrato "*IPluggable*" e portanto contém implementações de "*Import*", "*Process*" e "*Export*".

## Testando a aplicação

O teste da aplicação *host* será feito em duas etapas, para que nós possamos mostrar como é possível fazer a publicação de um *plug-in*, sem a necessidade de recompilação do *host*.

Primeiro, compile os projetos "*Plugin1*" e "*Plugin2*". Porém, copie somente o *assembly* "*Plugin1*" para o caminho especificado na variável "*path*" do método "*GetPlugins*" do host.

Execute a aplicação "*Host*". A saída é mostrada na **Figura 2**.

Agora, sem parar o host, copie o *assembly* "*Plugin2*" para o mesmo caminho especificado na variável "*path*" do método "*GetPlugins*" do *host*. A saída é mostrada na **Figura 3**. Observe que o *host* foi capaz de perceber o novo *assembly* (*Plugin2*) e fazer a execução do mesmo.

| Classe | Método | Description |
|---|---|---|
| System.Reflection.Assembly | GetTypes | Retorna um *array* contendo todos os tipos encontrados no *assembly*. |
| System.Type | GetInterface | Procura pela interface passada como parâmetro. |
| System.Activator | CreateInstance | Cria uma instância do tipo passado como parâmetro. |

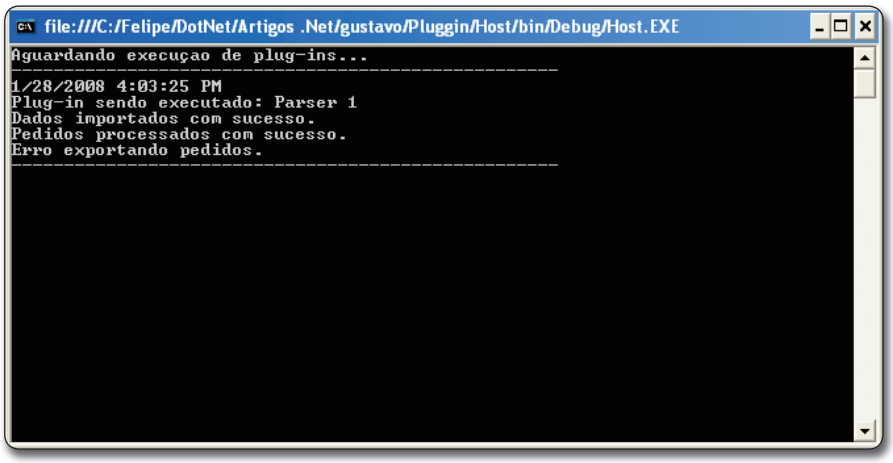


**Figura 2.** Execução apenas do Parser 1

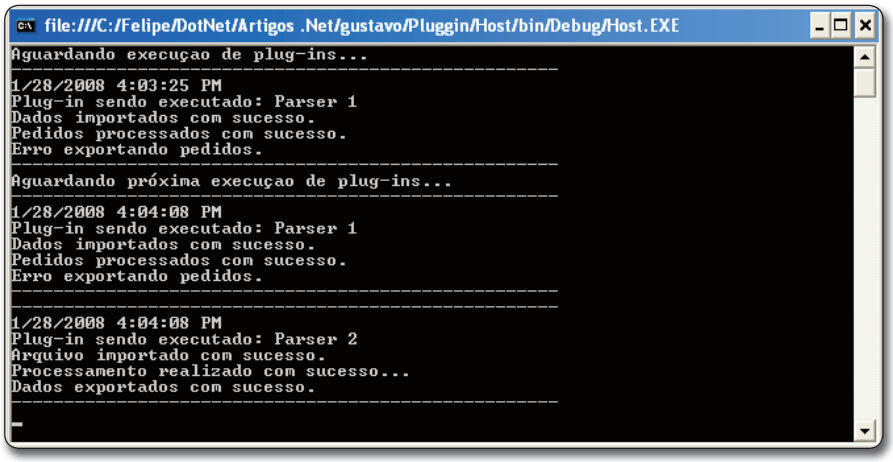


**Figura 3.** Execução do host agora com dois plug-ins (Parser 1 e Parser 2)

## Conclusão

Neste artigo procuramos mostrar um exemplo de aplicação com uma arquitetura *plugável*. O importante aqui é a técnica, ou seja, procuramos enfatizar a utilização da arquitetura de forma que você possa adaptá-la e utilizá-la em aplicações reais, utilizando, por exemplo, um serviço *Windows* como "*host*". A própria "*Enterprise Library*" da Microsoft utiliza essa técnica de ativação de *assemblies* em tempo de execução, de forma que você possa "estender" a biblioteca e criar seus próprios "*providers*". Outro ponto importante é a possibilidade de facilitar o "*deployment*" da aplicação, permitindo que novos componentes sejam adicionados, sem a necessidade de compilação da aplicação ou mesmo sem a necessidade de "parar" o serviço. ●