# TraceViewer : Un Add-In par l'exemple

## Introduction

Pourquoi cet add-in ? Tout simplement car je ressens le manque d'une fonctionnalité dans Visual Studio qui me permettrait de visualiser des traces lors du développement d'application sur Pocket PC.

Jusqu'à aujourd'hui j'utilisais Pocket Console qui fonctionne très bien, mais qui oblige à regarder en permanence l'écran du PPC et quand, par ailleurs, vous développez une application avec des fenêtres, il n'est pas pratique de basculer entre votre fenêtre et "Pocket Console".

Récemment, j'ai découvert sur cet article de Code Project décrivant la réalisation d'un logiciel permettant de recevoir et d'afficher les traces, provenant d'un Pocket PC, dans une fenêtre DOS.Tout cela fonctionne très bien, mais, comme je suis un peu maniaque, je trouvais qu'il manquait toujours l'intégration avec Visual Studio.

C'est donc pour cela que je vous propose cet article, afin d'une part d'expliquer le développement d'un Add-In et, d'autre part, de vous faire profiter de ce logiciel que je trouve (suis je le seul ?), bien pratique.

Afin de faire profiter le plus grand nombre de ce logiciel, le code source sera exceptionnellement en Anglais, je reviendrai sur ce point à la fin de l'article.

Nous allons donc nous pencher sur la création d'un Add-In, pas à pas, en ajoutant de plus en plus de fonctionnalités.

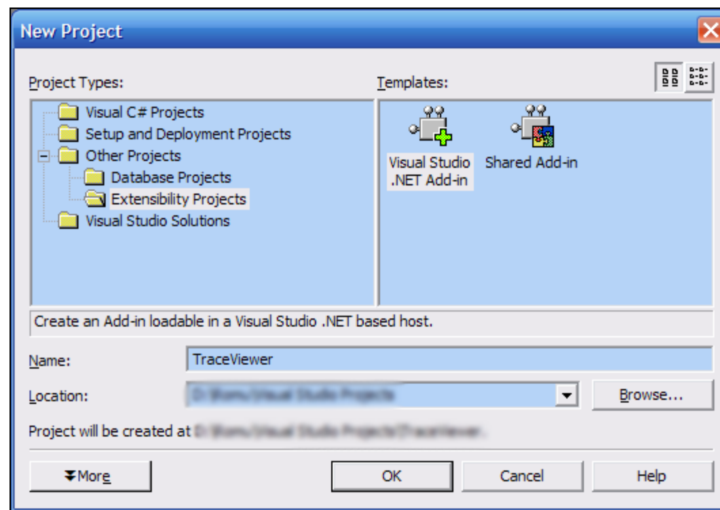
*Note 1 : pour une fois, oublions les restrictions du CF, place au "framework" .Net complet.*

*Note 2 : Comme vous pourrez le constater, l'aide pour les extensions de VS ne vaut pas celle fournit pour le framework .Net. En écrivant des Add-Ins, vous comprendrez vite qu'une connexion Internet est indispensable pour télécharger les exemples et fouiller dans le code. Une section "Bibliographie" est disponible à la fin du document pour vous aider.*

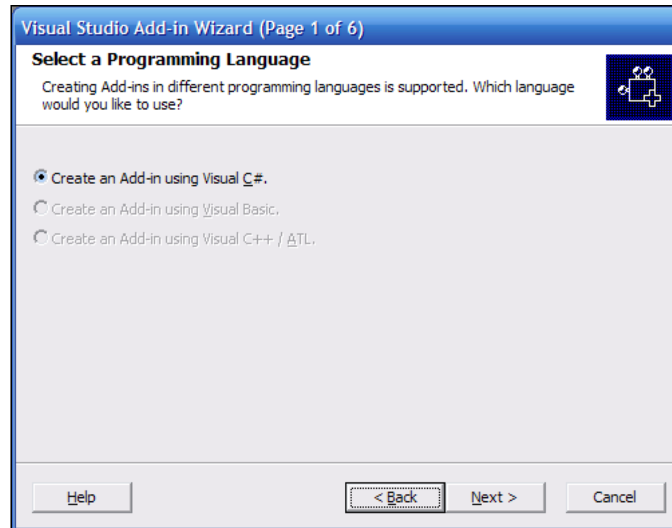
## Création de l'Add-In

### *Les premiers pas*

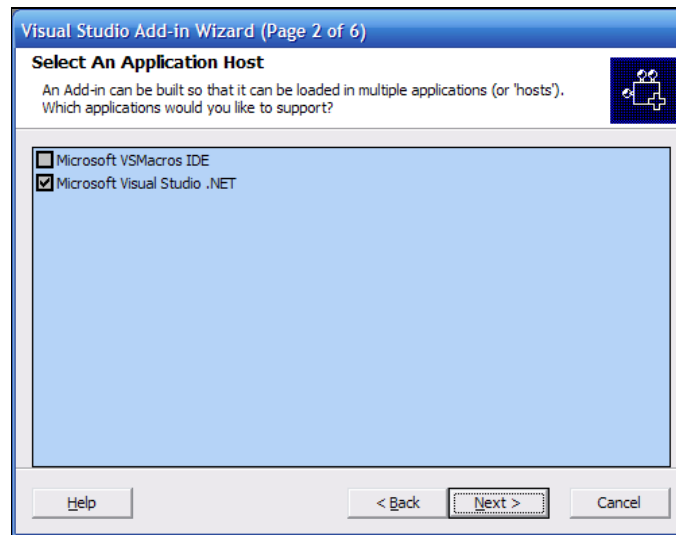
Pour créer l'Add-In, rien de plus simple, nous utilisons les assistants de VS.



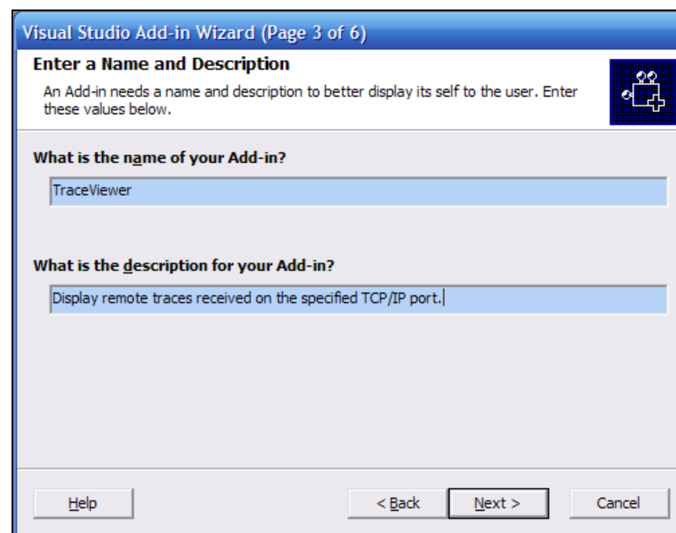

La première étape est donc de créer un nouveau projet dans VS qui sera de type "Extensibility" que nous appellerons "TraceViewer". Nous validons.



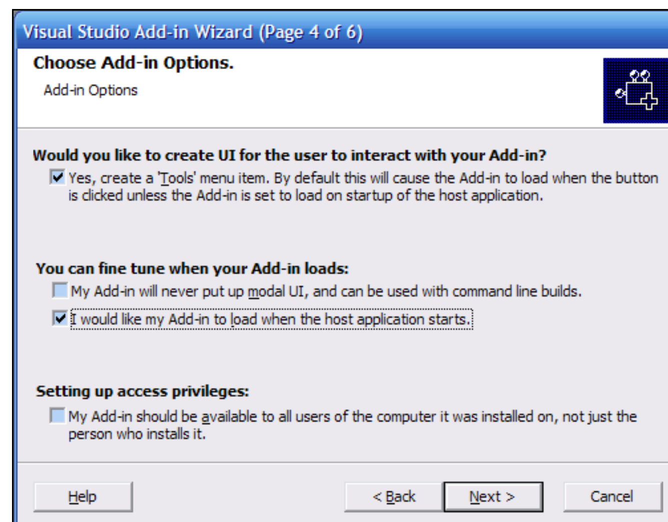

VS nous présente ensuite des fenêtres permettant de paramétrer notre Add-In. Ici, le projet sera créé en C#.



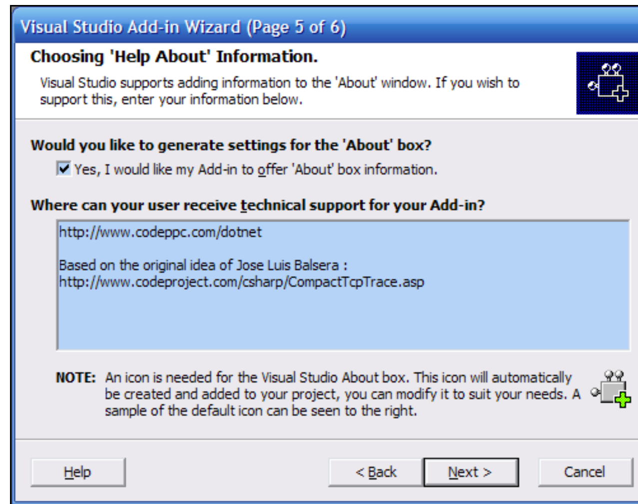

La seule application hôte possible pour notre Add-In sera Visual Studio.




Nous saisissons le nom ainsi qu'une rapide description de l'Add-In :

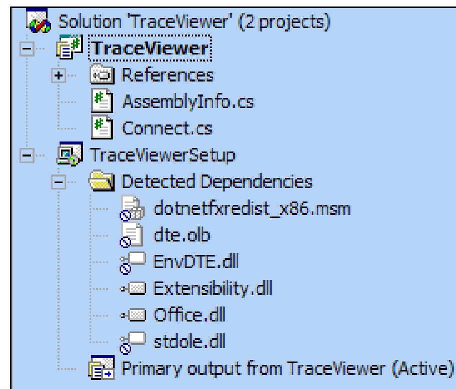


Il est temps de spécifier que nous voulons une entrée de menu et que notre Add-In doit démarrer au lancement de VS.




Il ne nous reste plus qu'à saisir les informations de la boîte "A propos…" et nous avons terminé la création du projet.

Si, maintenant, nous examinons la solution générée :




Nous remarquons que 2 projets ont été générés, l'Add-In lui-même, ainsi que le projet d'installation associé. Construisons la solution et oh! Miracle, un module d'installation MSI (Microsoft Installer) a été généré, nous n'avons plus qu'à nous concentrer sur l'aspect fonctionnel de l'Add-In, ce qui est déjà bien assez.

Regardons maintenant d'un peu plus près ce que nous a généré VS, nous allons nous intéresser plus particulièrement à la classe "Connect" :

```
namespace TraceViewer
{
    using System;
    using Microsoft.Office.Core;
    using Extensibility;
    using System.Runtime.InteropServices;
    using EnvDTE;

    Read me for Add-in installation and setup information.

    /// <summary>
    ///   The object for implementing an Add-in.
    /// </summary>
    /// <seealso class='IDTExtensibility2' />
    [GuidAttribute("8102C6F1-1BDA-4F32-BA69-024F8AF12FBB"), ProgId("TraceViewer.Connect")]
    public class Connect : Object, Extensibility.IDTExtensibility2, IDTCommandTarget
    {
        /**/
        public Connect()...
        /**/
        public void OnConnection(object application, Extensibility.ext_ConnectMode connectMode, object addInInst, ref System.Array custom)...
        /**/
        public void OnDisconnection(Extensibility.ext_DisconnectMode disconnectMode, ref System.Array custom)...
        /**/
        public void OnAddInsUpdate(ref System.Array custom)...
        /**/
        public void OnStartupComplete(ref System.Array custom)...
        /**/
        public void OnBeginShutdown(ref System.Array custom)...
        /**/
        public void QueryStatus(string commandName, EnvDTE.vsCommandStatusTextWanted neededText, ref EnvDTE.vsCommandStatus status, ref object commandText)...
        /**/
        public void Exec(string commandName, EnvDTE.vsCommandExecOption executeOption, ref object varIn, ref object varOut, ref bool handled)...
        private _DTE applicationObject;
        private AddIn addInInstance;

    }
}
```

Nous remarquons en premier lieu que VS a généré un code "à la sauce" COM, en effet VS n'est pas (encore ?) une application .Net, mais hautement basé sur COM. Nous observons, ensuite, que notre classe "Connect" implémente l'interface "IDTExtensibility2" qui permet à notre Add-In d'interagir avec VS ainsi que l'interface "IDTCommandTarget" qui est dédiée à la gestion des commandes (menus, boutons…).

La classe générée implémente donc le squelette de toutes les méthodes pour la connexion/déconnexion de l'Add-In, sa mise à jour, le chargement/déchargement de VS, l'exécution ou la gestion de l'état des commandes. Bref, cette classe représente le point d'entrée de notre Add-In, c'est pas elle que tout va maintenant commencer.

## *Ajout du menu*

Examinons un peu le code de la méthode "OnConnection" qui est le point d'entrée de notre Add-In.

```
public void OnConnection(object application, Extensibility.ext_ConnectMode connectMode, object addInInst, ref System.Array custom)
{
    applicationObject = (_DTE)application;
    addInInstance = (AddIn)addInInst;
    if(connectMode == Extensibility.ext_ConnectMode.ext_cm_UISetup)
    {
        object []contextGUIDS = new object[] { };
        Commands commands = applicationObject.Commands;
        _CommandBars commandBars = applicationObject.CommandBars;

        // When run, the Add-in wizard prepared the registry for the Add-in.
        // At a later time, the Add-in or its commands may become unavailable for reasons such as:
        //   1) You moved this project to a computer other than which is was originally created on.
        //   2) You chose 'Yes' when presented with a message asking if you wish to remove the Add-in.
        //   3) You add new commands or modify commands already defined.
        // You will need to re-register the Add-in by building the TraceViewerSetup project,
        // right-clicking the project in the Solution Explorer, and then choosing install.
        // Alternatively, you could execute the ReCreateCommands.reg file the Add-in Wizard generated in
        // the project directory, or run 'devenv /setup' from a command prompt.
        try
        {
            Command command = commands.AddNamedCommand(addInInstance,                                    // VS Instance
                                                       "TraceViewer",                                    // Command name
                                                       "TraceViewer",                                    // Command Text
                                                       "Executes the command for TraceViewer",           // Command ToolTip
                                                       true,                                             // Menu with bitmap
                                                       59,                                               // Bitmap ID
                                                       ref contextGUIDS,                                 // Commande GUID
                                                       (int)vsCommandStatus.vsCommandStatusSupported+(int)vsCommandStatus.vsCommandStatusEnabled);
            CommandBar commandBar = (CommandBar)commandBars["Tools"];
            CommandBarControl commandBarControl = command.AddControl(commandBar, 1);
        }
        catch(System.Exception /*e*/)
        {
        }
    }
}
```

Et là tout de suite, le gros commentaire vous met dans l'ambiance, la gestion des commandes est un domaine **<u>très</u>** sensible quand on développe un Add-In. D'une exécution sur l'autre, la commande va apparaître, disparaître…

Pour gérer cela au mieux, nous allons isoler le code d'ajout de la commande dans une méthode séparée, et modifier très légèrement son code.

```
private void AddCommand()
{
    object []contextGUIDS = new object[] { };
    Commands commands = applicationObject.Commands;
    _CommandBars commandBars = applicationObject.CommandBars;

    // When run, the Add-in wizard prepared the registry for the Add-in.
    // At a later time, the Add-in or its commands may become unavailable for reasons such as:
    //   1) You moved this project to a computer other than which is was originally created on.
    //   2) You chose 'Yes' when presented with a message asking if you wish to remove the Add-in.
    //   3) You add new commands or modify commands already defined.
    // You will need to re-register the Add-in by building the TraceViewerSetup project,
    // right-clicking the project in the Solution Explorer, and then choosing install.
    // Alternatively, you could execute the ReCreateCommands.reg file the Add-in Wizard generated in
    // the project directory, or run 'devenv /setup' from a command prompt.
    try
    {
        Command command = commands.AddNamedCommand( addInInstance,                          // VS Instance
                                                    "Options",                              // Command name
                                                    "Set Trace Viewer Options",             // Command Text
                                                    "Modify the Trace Viewer settings",     // Command ToolTip
                                                    true,                                   // Menu with bitmap
                                                    59,                                     // Bitmap ID
                                                    ref contextGUIDS,                       // Commande GUID
                                                    (int)vsCommandStatus.vsCommandStatusSupported+(int)vsCommandStatus.vsCommandStatusEnabled);

        CommandBar commandBar = (CommandBar)commandBars["Tools"];   // The command is added to the "Tools" VS menu.
        CommandBarControl commandBarControl = command.AddControl(commandBar, 1);
    }
    catch(System.Exception /*e*/)
    {
    }
}
```

De la même manière, nous allons ajouter une méthode "RemoveCommand".

```
private void RemoveCommand()
{
    try
    {
        Command command = applicationObject.Commands.Item ("TraceViewer.Connect.Options", -1);
        command.Delete();
    }
    catch(Exception)
    {
        // Thrown if command doesn't already exist.
    }
}
```

On remarquera que, dans ce cas, la commande est spécifiée avec son nom complet, le "program ID" COM de notre classe "Connect" étant "TraceViewer.Connect", il suffit de rajouter le nom de la commande pour avoir le nom complet. Modifions maintenant la méthode "OnConnection" de la façon suivante.

```
        public void OnConnection(object application, Extensibility.ext_ConnectMode connectMode, object addInInst, ref System.Array custom)
        {
            applicationObject = (_DTE)application;
            addInInstance = (AddIn)addInInst;

#if DEBUG

            // Remove the command if it does exist
            RemoveCommand();
            // In debug mode, the command is always remove/added to quickly see changes in VS.
            AddCommand();
#else
            // Normal operating mode, the commande is added once, at the first Add-In start
            if(connectMode == Extensibility.ext_ConnectMode.ext_cm_UISetup)
            {
                AddCommand();
            }
#endif
        }
```

Pourquoi ce code ? Tout simplement car enlever et ajouter la commande à chaque exécution va forcer VS à reconstruire la table des commandes à chaque lancement.
Cela permet d'être sur que toutes les modifications faîtes dans le code lors du développement seront prises en compte d'une exécution à l'autre, c'est utile en mode développement ("debug).
Par contre, en mode production ("release"), nous suivons les recommandations de Microsoft, à savoir que la commande est enregistrée une seule fois, lors du premier chargement de l'Add-In.

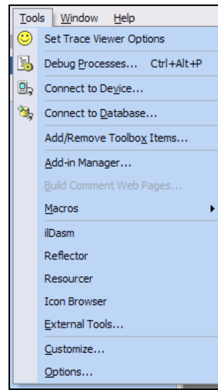
*Note : Si jamais vous rencontrez ces problèmes de disparation ou d'apparition multiple de commandes, plusieurs solutions sont disponibles pour remédier au problème :*

- *VS a généré pour vous un fichier ".reg" qui permet de forcer le rechargement de votre commande ("ReCreateCommands.reg"),*
- *Forcez le reconstruction de la liste des commandes en ouvrant un fenêtre DOS sur "<VSFolder>\Common7\IDE" puis tapez "devenv /setup"*
- *Une dernière solution est de construire le projet d'installation puis de supprimer/installer l'Add-In. Il conviendra alors, dans ce cas, de parcourir le "registry" pour changer TOUS les chemins pointant vers "TraceViewer.dll" ET "TraceViewer.tlb" pour pointer vers les versions du dossier "debug" de votre projet.*

Nous suivons le même principe pour la méthode "OnDisconnection".

```
        public void OnDisconnection(Extensibility.ext_DisconnectMode disconnectMode, ref System.Array custom)
        {
#if DEBUG
            RemoveCommand();
#endif
        }
```

Exécutons notre programme (F5) et voyons ce qu'il se passe. Tout d'abord vous pouvez constater que VS lance…VS, c'est normal pour déboguer un Add-In de VS, il faut lancer d'abord VS.




Et voilà le travail !!!

Nous allons maintenant nous attarder sur la méthode "QueryStatus".

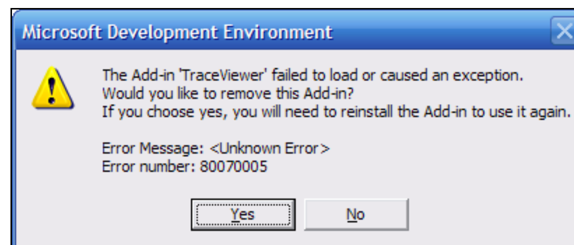
```
public void QueryStatus(string CmdName, EnvDTE.vsCommandStatusTextWanted NeededText,
                        ref EnvDTE.vsCommandStatus StatusOption, ref object CommandText)
{
    if(NeededText == EnvDTE.vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)
    {
        if(CmdName == "TraceViewer.Connect.Options")
        {
            StatusOption = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported|vsCommandStatus.vsCommandStatusEnabled;
        }
    }
}
```

A quoi sert le premier test ? Je n'en ai pas la moindre idée, mais TOUS les exemples sur Internet font comme cela, alors, il doit vraiment falloir faire comme cela.

Pour le reste on teste si la commande, dont VS veut rafraîchir le statut, est la notre et, dans ce cas, nous autorisons la commande car, notre Add-In étant démarré au lancement de VS, il est supposé toujours actif.

Nous reviendrons sur la méthode "Exec" un peu plus tard. Nous allons maintenant nous occuper de la fenêtre d'affichage des traces.

*Note : si jamais, en écrivant un Add-In, votre code plante (mais si…ça arrive), vous verrez probablement la fenêtre suivante s'afficher :*




*Je vous conseille alors vivement de répondre non à la question posée, car sinon, vous devrez re-installer votre Add-In et donc aller, à nouveau, modifier le "registry" comme décrit précédemment. Lors de la réponse "No", VS va juste décharger votre Add-In et vous pourrez à nouveau ultérieurement le relancer, c'est beaucoup plus pratique.*

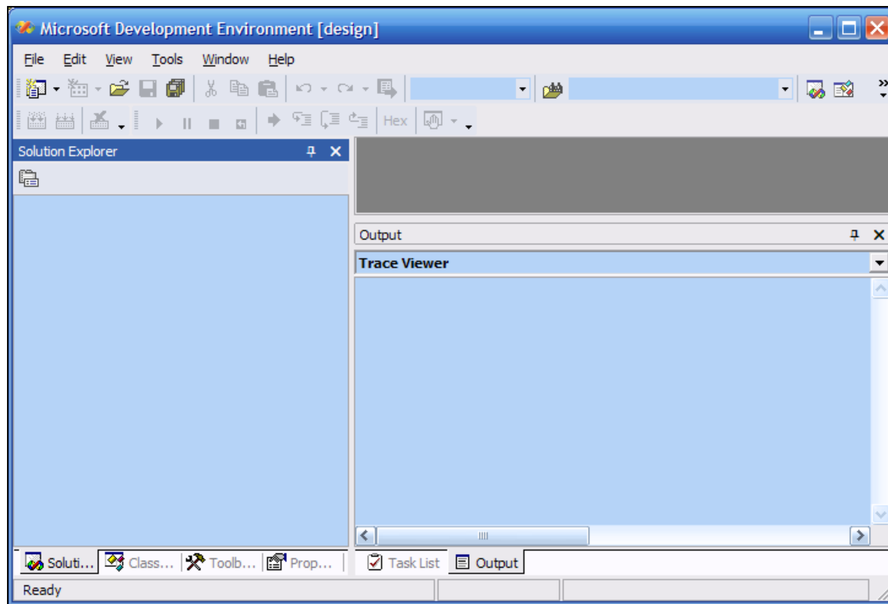
### *Création de la fenêtre d'affichage des traces*

Nous travaillons toujours dans la méthode "OnConnection" et nous ajoutons à la fin de cette méthode les lignes suivantes.

```
// Create the Visual Studio output window
Window window = applicationObject.Windows.Item( EnvDTE.Constants.vsWindowKindOutput );
OutputWindowPane pane = ((OutputWindow) window.Object).OutputWindowPanes.Add("Trace Viewer");
```

Dans ce bout de code :

- La première ligne nous permet d'obtenir l'instance de la fenêtre "Output",
- Nous ajoutons à la fenêtre "Output" un nouveau sortie que nous appelons "Trace Viewer".




Nous avons donc créé une fenêtre dans VS, magique je vous dis !!!

Avant de nous occuper du traitement et de l'affichage des messages, nous allons nous pencher sur la gestion des options de notre Add-In.

## La gestion des options

### *La classe de gestion*

Pour recevoir des traces, notre Add-In va devoir écouter sur un port TCP, donc être serveur. Qui dit TCP/IP serveur dit adresse IP et numéro de port TCP associé. C'est pourquoi nous allons créer une nouvelle classe que nous appèlerons "Options". Cette classe aura pour rôle :

- maintenir les valeurs des options,
- les stocker de manière permanente (sur disque),
- les restaurer au lancement de l'Add-In,
- afficher une boîte de dialogue permettant de modifier les valeurs.

Pour les stocker de façon permanente, nous allons suivre les recommandations de Microsoft concernant les options des Add-In. Ces données doivent être stockées dans le "registry" à la clé suivante :

**\Software\Microsoft\VisualStudio\X.Y\Addins\TraceViewer.Connect\Options**

X.Y désigne la version de VS qui exécute l'Add-In. La ruche est laissée à l'appréciation du développeur de l'Add-In : "LOCAL_MACHINE" et tous les utilisateurs partagent les mêmes valeurs, "CURRENT_USER" et chacun peut avoir sa propre configuration.

Pour notre part, nous stockerons ces données dans la ruche utilisateur. L'interface publique de notre classe "Options" sera la suivante :

```
public class Options
{
    --- Private ---

    /**/
    public UInt16 Port...
    /**/
    public UInt16 Index...
    /**/
    public bool Changed...
    /**/
    public bool Started...
    /**/
    public Options(_DTE dte)...
    /**/
    public void Read()...
    public void Write()...
    public void Change()...
}
```

Je vous laisse le soin de lire le code des méthodes "Read" et "Write" qui est très simple. Nous allons donc nous occuper de réaliser la boîte de dialogue permettant de changer ces valeurs.

Mais je pense qu'une petite explication s'impose concernant la propriété "Index". Sur le PC qui exécute l'Add-In, vous pouvez très bien avoir plusieurs cartes réseau, auquel cas, vous aurez plusieurs adresses IP. Les classes de System.Net sont capables (voir dans la suite de l'article) de vous donner la liste des adresses configurées sur la machine, ainsi pas besoin de mémoriser cette adresse, seul l'index de celle-ci dans la liste suffit.

La valeur "Started", quant à elle est là uniquement pour savoir si l'Add-In a déjà été démarré ou non. En effet, comme celui-ci est un serveur TCP/IP sur un port précis (voir plus loin), il ne peut être démarré qu'une seule. Mais le cycle de vie d'un Add-In dans VS, fait que celui-ci peut être chargé/déchargé à la demande de l'utilisateur. D'où la nécessité de gérer cette variable.

Dans notre classe "Connect", nous créons un attribut privé de type "Options", qui sera instancié dans la méthode "OnConnection" comme suit :

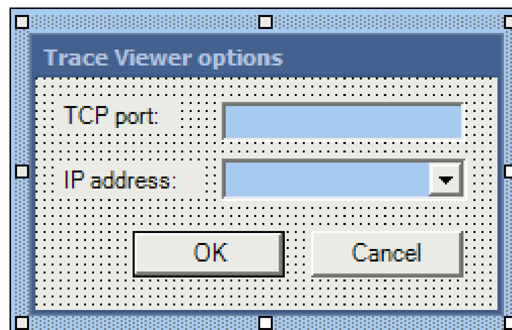
```
// We read the options.
Options options = new Options(applicationObject);

// If the port is 0, display an error message and
// the trace viewer is not started
if(options.Port == 0)
{
    pane.OutputTaskItemString("The Trace Viewer port must not be 0" + Environment.NewLine,
        EnvDTE.vsTaskPriority.vsTaskPriorityHigh,
        "Trace Viewer add-in",
        EnvDTE.vsTaskIcon.vsTaskIconNone,
        "", 0,
        "The TCP Trace Viewer must be set to a non null value, please edit the registry value and re-start the Add-In",
    return;
}
```

Passons maintenant à la boîte de dialogue qui va nous permettre de modifier ces options.

## *La boîte de dialogue*

Pour cette partie là, nous allons un peu nous éloigner des recommandations de Microsoft. En effet, la documentation des Add-In décrit comment ajouter une fenêtre "Trace Viewer" dans la fenêtre "Options" de VS. On y apprend alors que les dialogues de la fenêtre "Options" sont des contrôles ActiveX (en C++ donc) qui sont comme des adaptateurs de classes C#. Je vous avoue qu'à l'heure du code géré (Java, .Net), je me dis qu'ils n'ont pas fait trop d'efforts chez Microsoft.

C'est pourquoi, nous n'allons pas suivre des recommandations, et faire au plus simple, une petite boîte de dialogue qui sera directement accédée par une commande de VS. Cette boîte aura l'aspect suivant :




Un "ToolTip" est affiché lorsque le pointeur de la souris survole les contrôles afin d'informer l'utilisateur de l'usage de chaque valeur.

La liste des adresses IP (explication dans le chapitre précédent) est automatiquement remplie par le code suivant :

```
// Fill the addresses combo box
string name = Dns.GetHostName();
this.cbAddresses.DataSource = Dns.GetHostByName(name).AddressList;
```

Il ne reste plus qu'à ajouter les propriétés publiques pour affecter et récupérer les valeurs saisies :

```
public UInt16 Port
{
    get
    {
        return UInt16.Parse(tbPort.Text);
    }
    set
    {
        tbPort.Text = value.ToString();
    }
}

public UInt16 Index
{
    get
    {
        return (UInt16) cbAddresses.SelectedIndex;
    }
    set
    {
        cbAddresses.SelectedIndex = value;
    }
}
```

### *La méthode "Change"*

Une fois notre boîte de dialogue est terminée, il ne nous reste plus (pour la classe "Options" tout du moins) qu'à finir la méthode "Change".

Néanmoins, il convient de gérer le changement de ces valeurs, c'est-à-dire de bien déployer les nouvelles valeurs au sein de notre Add-In. La manière "propre", au moins disons la plus élégante, de faire cela en .Net serait d'utiliser les évènements (avec les "delegate"). Nous allons ici faire plus simple.

La méthode "Change" ressemble alors à ce qui suit.

```
public void Change()
{
    OptionsDialog dialog = new OptionsDialog();
    dialog.Port = port;
    dialog.Index = index;
    if(dialog.ShowDialog() == DialogResult.OK)
    {
        // Allow the Options user to verify if the values
        // have been changed or not.
        changed = (port != dialog.Port) || (index != dialog.Index);
        if(changed)
        {
            port = dialog.Port;
            index = dialog.Index;
            Write();
        }
    }
}
```

Avant de voir comment nous allons afficher ce dialogue, nous allons passer à l'affichage des traces.

## Réception et affichage des traces

Nous commençons là le vif du sujet en ajoutant une classe appelée "TraceViewer" qui aura l'interface publique suivante :

```
namespace TraceViewer
{
    /// <summary>
    /// This class listens to the TCP port registered into the registry
    /// and displays messages.
    /// </summary>
    public class TraceViewer
    {
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="pane">pane used to display messages</param>
        /// <param name="port">Listened UDP port</param>
        /// <param name="index">Index of the IP address if several are created on the machine</param>
        public TraceViewer(OutputWindowPane pane, UInt16 port, UInt16 index)...
        /// <summary>
        /// Start a Trace Viewer session.
        /// </summary>
        /// <param name="port">The TCP port to be listened to.</param>
        /// <param name="index">The index of the IP address to use.</param>
        public void Start(UInt16 port, UInt16 index)...
        /// <summary>
        /// Terminate the current Trace Viewer Session.
        /// </summary>
        public void Terminate()...
    }
}
```

Nous remarquons que le constructeur requiert 3 paramètres :

- "pane sera le panneau en charge de l'affichage des traces,
- "port" est la valeur du port TCP qui sera écouté,
- "index" est l'index de l'adresse IP qui sera utilisée pour écouter, nous avons cela précemment.

*Note : La gestion du protocole UDP aurait été plus simple que TCP (pas de processus client/serveur), mais le problème vient du fait qu'ActiveSync ne transite pas ces trames là, donc TCP est obligatoire dans le cas où l'on veut pouvoir lire les traces provenant d'un Pocket PC*

Pourquoi répéter les paramètres IP à la méthode "Start", tout simplement pour pouvoir redémarrer le processus d'écoute sans créer une nouvelle instance de la classe.

Le code du constructeur sera très simple : mémorisation du panneau de sortie, et lancement du processus. Le code de la méthode "Start" sera, lui, beaucoup plus intéressant.

```
public void Start(UInt16 port, UInt16 index)
{
    // Get the IP address to use
    string name = Dns.GetHostName();
    IPAddress[] addrs = Dns.GetHostByName(name).AddressList;
    pane.OutputString("*** Listened address is " + addrs[index] + Environment.NewLine);
    pane.OutputString("*** Listened TCP port is " + port + Environment.NewLine);

    // Start the listening socket
    IPEndPoint ep = new IPEndPoint(addrs[index], port);
    server = new TcpListener(ep);
    server.Start();
    pane.OutputString("*** Trace Viewer started" + Environment.NewLine);

    // Create and start a thread to process connection requests
    listener = new System.Threading.Thread(new ThreadStart(this.ProcessTcpConnection));
    listener.Start();
}
```

Détaillons un peu ce code :

- Le premier bloc nous sert à récupérer le nom du PC qui exécute l'Add-In, avec ce nom nous obtenons la liste des adresses IP du PC (c'est là que la variable "index" nous est utile), puis nous affichons ces paramètres de démarrage dans la fenêtre de sortie
- Le deuxième bloc de code de la méthode est celui qui lance l'écoute sur le port TCP voulu,
- Le dernier bloc démarre le "thread" de traitement des demandes de connexion.

Nous allons donc maintenant nous préoccuper de la méthode de traitement des demandes de connexion.

```
private void ProcessTcpConnection()
{
    while(true)
    {
        try
        {
            TcpClient client = server.AcceptTcpClient();
            pane.OutputString("*** New connection" + Environment.NewLine);
            object[] obj = new object[2];
            obj[0] = client;
            obj[1] = pane;
            ThreadPool.QueueUserWorkItem(new WaitCallback(ProcessMessages), obj);
        }
        catch(Exception e)
        {
            // This exception is raised when we close VS and no connection
            // have been created.
        }
    }
}
```

Cette méthode est une simple boucle infinie. La méthode "AcceptTcpClient" est bloquante et ne sort que lorsque une demande de connexion est parvenue au serveur, dans ce cas là, nous lançons le "thread" de traitement des messages reçus.

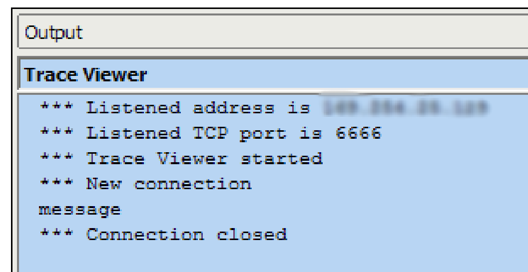
Seul le code intéressant de la méthode "ProcessMessages" est présenté ici.

```
TcpClient client = (TcpClient) ((object[]) o)[0];
OutputWindowPane pane = (OutputWindowPane) ((object[]) o)[1];
Byte[] data = new Byte[512];

try
{
    NetworkStream stream = client.GetStream();
    int count = 0;
    // From the NetworkStream doc, if the read count is null,
    // this is because the connection has been shutdown by the remote
    // peer.
    while ((count = stream.Read(data, 0, data.Length)) != 0)
    {
        pane.OutputString(Encoding.Unicode.GetString(data, 0, count) + Environment.NewLine);
    }

    pane.OutputString("*** Connection closed" + Environment.NewLine);

    // From the NetworkStream doc, a call to the TcpClient class
    // calls the NetworkStream.Close.
    client.Close();
}
```

Vous remarquerez que le code de José Luis Balsera a été simplifié, en effet la documentation est très claire à ce sujet, pas besoin de marqueur de fin de message ou autre, dès qu'on ne lit plus, c'est que la connexion est terminée.

La méthode "Terminate" est, quant à elle, très simple, je vous laisse le soin d'examiner son code. Cette méthode sera d'ailleurs appelée dans la méthode "OnDisconnection" de notre classe "Connect" afin de terminer correctement le programme.

Faisons maintenant un petit essai avec un programme de test sur PC (livré dans l'archive du code). Nous ajoutons tout d'abord un membre à la classe "Connect" de type "TraceViewer". L'instance est créée après l'ajout de la commande de menu dans la méthode "OnConnection" :




Voilà donc ce que donne l'affichage des traces reçues.

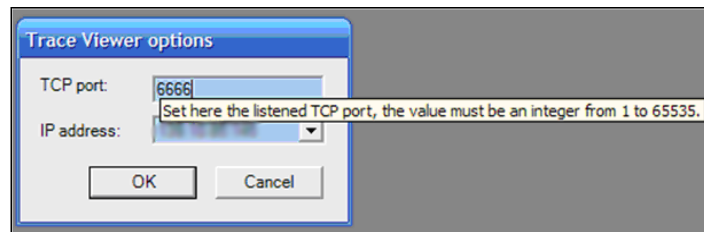
*Note : lorsque vous déboguez un Add-In, si vous placez des points d'arrêts dans les méthodes de la classe "Connect", vous serez peut être surpris de voir VS "perdre les pédales" : s'arrêter à des endroits farfelus (commentaires), afficher des valeurs qui ne correspondent pas aux votres…cela vient peut être du fait que vous exécutez plusieurs Add-In dans VS. Cela est très facile à vérifier, quand vous êtes sur un point d'arrêt qui semble farfelu, vérifiez dans la fenêtre "Locals" que le type de la classe "Connect" affiché est bien le votre, si ce n'est pas le cas, continuez l'exécution et tout rentrera dans l'ordre.*

## Changement des options

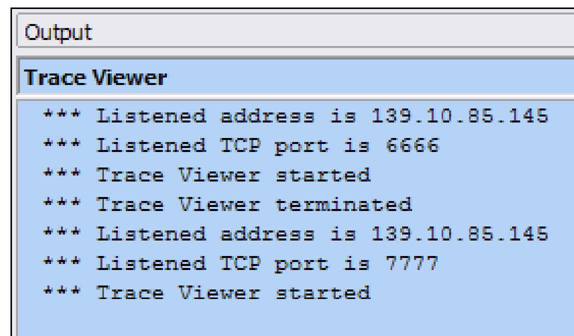
Ben je crois qu'il ne nous reste plus qu'à nous occuper de la méthode "Exec", c'est à dire d'écrire le code qui appellera le dialogue de changement des options et qui répercutera ces changements sur "TraceViewer".

```
public void Exec(   string CmdName, EnvDTE.vsCommandExecOption ExecuteOption,
    ref object VariantIn, ref object VariantOut, ref bool handled)
{
    handled = false;
    if(ExecuteOption == EnvDTE.vsCommandExecOption.vsCommandExecOptionDoDefault)
    {
        if(CmdName == "TraceViewer.Connect.Options")
        {
            handled = true;
            options.Change();
            if(options.Changed)
            {
                viewer.Terminate();
                viewer.Start(options.Port, options.Index);
            }
        }
    }
}
```

Si nous activons notre menu.




Ca marche !! Changeons la valeur du port, validons et observons la fenêtre de sortie.




# A l'usage

Dans l'archive, vous trouverez un dossier "Trace" qui contient une classe "TraceListener" que vous pouvez utiliser dans vos programmes. Cette classe vous permettra de re-diriger tous les appels à "System.Diagnostics.Debug.WriteXXX" vers notre Add-In "Trace Viewer".

J'ai, de plus, pris contact avec l'équipe de développement de Log4Net, afin de leur demander de créer une configuration de l'outil pour envoyer la sortie des messages vers "Trace Viewer", ainsi qu'avec l'équipe de "Pocket Console" pour leur faire la même demande.

J'espère surtout que c'est cette dernière démarche qui aboutira. En effet cela permettrait d'utiliser "Trace Viewer" quel soit le langage et la plateforme utilisée sur le Pocket PC. Que vous développiez

en natif, Java, eVB ou .Net, tous les messages seraient affichés dans la fenêtre de notre Add-In et serait bien pratique.

Au cas où ces démarches échoueraient, je fais au volontariat. Le code source (en C++) de "Pocket Console" est disponible sur leur site, il suffirait juste d'isoler le code du pilote et, au lieu d'ouvrir une fenêtre, de re-diriger les sorties en ouvrant un "socket" client vers notre Add-In. Pour les volontaires, mon adresse courriel est à la fin du document. Cette solution est vraiment l'idéale et rendrait service à beaucoup.

Alors à votre bon cœur !!

# Bibliographie

Voici quelques liens intéressants pour le développement d'Add-In à Visual Studio :

- Un document PDF très clair sur les Add-In
- Un tutorial intéressant sur CSharp Corner
- les exemples Microsoft
- les Add-In proposés lors d'un concours sur Internet
- l'Add-In du gestionnaire de version (excellent d'ailleurs) Subversion, appelé AnkhSvn, tout le code source est fourni, c'est bien conçu, mais un peu lourd pour tout analyser

Vous trouverez aussi d'autres exemples sur les site CodeGuru et Code Project.
Mais tout cela en Anglais bien sur.

# Conclusion

J'espère que cet article vous aidera à développer vos propres Add-In, c'est un sujet intéressant, il y a plein de choses à faire, mais c'est assez ardu et la documentation n'est pas spécialement bien étoffée.

J'espère aussi que, comme moi, vous trouverez un intérêt à l'utilisation de cet Add-In, qu'il facilitera, un peu, vos développements Pocket PC. Si vous avez des problèmes à l'utilisation de "Trace Viewer", ou si vous avez des idées d'évolutions, remarques ou autre, mon adresse est ouverte :

Softromu_chez_hotmail.com (Remplacez le "_chez_" par @).