

Utilisation des outils de débogage du framework

par Johann Blais ([Mon espace personnel](#))

Date de publication : 17/04/2008

Dernière mise à jour :

L'objectif de cet article est de présenter les différents outils intégrés directement dans le framework .NET pour faciliter le débogage.

Préambule.....	3
Introduction.....	4
I - Utiliser l'objet Debugger.....	5
I-A - La méthode Break.....	5
I-B - La méthode Launch.....	6
I-C - La méthode Log.....	9
II - Utiliser l'objet Debug.....	11
II-A - Ecriture des messages dans la trace.....	11
II-B - Contrôler le déroulement d'un programme.....	13
II-B-1 - La méthode Assert.....	13
II-B-2 - La méthode Fail.....	14
II-C - Utilisation des fichiers de configuration.....	14
III - Utiliser les attributs de débogage.....	16
III-A - DebuggerBrowsableAttribute.....	16
III-A-1 - DebuggerBrowsableState.Never.....	16
III-A-2 - DebuggerBrowsableState.RootHidden.....	17
III-B - DebuggerDisplayAttribute.....	18
III-C - DebuggerHiddenAttribute.....	20
III-D - DebuggerNonUserCodeAttribute.....	20
III-E - DebuggerStepThroughAttribute.....	20
III-F - DebuggerStepperBoundaryAttribute.....	20
III-G - DebuggerTypeProxyAttribute.....	21
III-H - DebuggerVisualizerAttribute.....	22
Conclusion.....	24
Remerciements.....	25

Préambule

Cet article s'adresse aux développeurs qui sont déjà familiers avec les phases de débogage classiques ainsi qu'avec les concepts généraux (exécution pas à pas, entrée/sortie d'une méthode, etc.).

Introduction


Le débogage fait partie intégrante de la phase de développement mais aussi de la maintenance d'une application. Il est donc important de se faciliter la tâche en intégrant, dès le démarrage du développement, les outils nécessaires à un débogage rapide et efficace. C'est l'objectif principal de cet article. Nous verrons d'abord les principaux objets disponibles dans le framework ainsi que leur utilisation concrète, puis nous terminerons par quelques techniques pour modifier le comportement du débogueur intégré dans Visual Studio.

I - Utiliser l'objet Debugger

L'objet principal permettant d'interagir avec un débogueur externe est nommé Debugger et se trouve dans le namespace System.Diagnostics. En règle générale, lors d'une phase de débogage, vous définissez vos points d'arrêts à différents endroits dans le code, ensuite vous lancez l'application et la faites se comporter de telle manière qu'elle finisse par arriver sur l'un ou l'autre de ces points. Il vous reste ensuite à exécuter le code pas à pas pour arriver à déterminer l'origine du bogue.

L'objet Debugger introduit un tout nouveau mode de fonctionnement, les points d'arrêts sont définis dans le code à différents endroits déterminés lors de l'écriture initiale par le développeur. Quel est l'intérêt de ce fonctionnement, me direz-vous ? Tout simplement, le développeur de l'application est la personne la plus à même de savoir quand, et surtout où dans le code, une condition ne devrait pas se produire.

Nous n'allons pas décrire entièrement l'objet Debugger, mais uniquement les méthodes qui présentent un réel intérêt pour le sujet qui nous intéresse.

 *Il est important de noter dès à présent que l'objet Debugger n'a d'effet sur le code et le fonctionnement du programme que lorsque l'application s'exécute en mode Debug.*

I-A - La méthode Break

Nous venons d'évoquer la nouvelle approche qui consiste à définir les points d'arrêts lors du développement (en amont de la phase de débogage même si les deux sont indissociables).

Pour définir un point d'arrêt directement dans le code de l'application, il suffit d'appeler la méthode Break de l'objet Debugger. Cette méthode a pour effet de mettre en pause l'exécution du code dans Visual Studio et de positionner le curseur d'instruction active sur la ligne concernée (la ligne qui contient le Debugger.Break()).

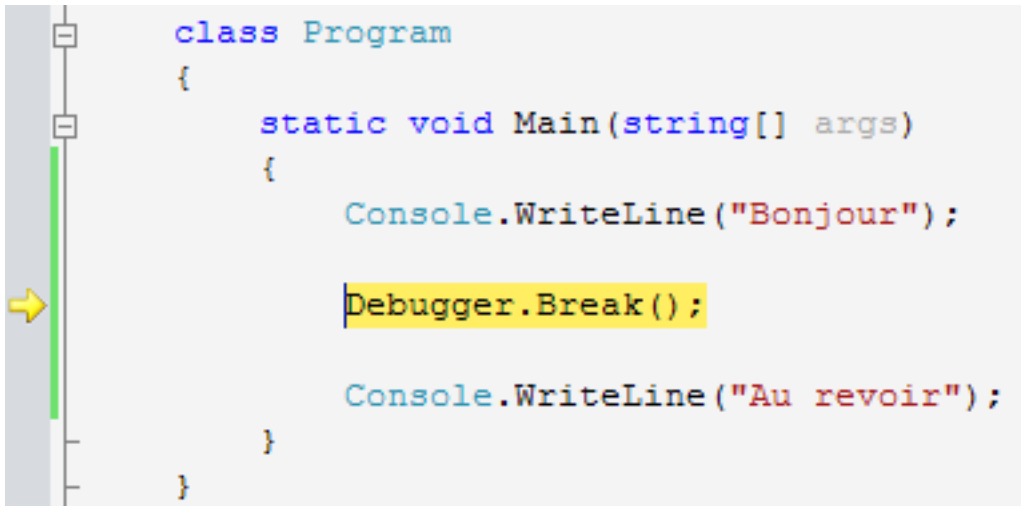
Un exemple simple

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Bonjour");

        Debugger.Break();

        Console.WriteLine("Au revoir");
    }
}
```

A chaque exécution de la méthode Main (en mode Debug), au moment d'exécuter l'instruction Debugger.Break(), le débogueur de Visual Studio va mettre l'exécution en pause et se positionner directement sur la ligne Debugger.Break(). A l'opposé, si vous exécutez le programme en mode Release, rien ne se passera. Nous avons la confirmation ferme et définitive que Debugger est un objet destiné au développeur et non pas à l'utilisateur de l'application.



Le débogueur s'arrête automatiquement sur l'instruction `Debugger.Break()`

L'objectif de l'exemple était principalement de présenter la syntaxe, il est clair que l'intérêt d'un tel point d'arrêt est nul. Alors où mettre un point d'arrêt automatique, me demanderez-vous ?

On peut distinguer plusieurs cas d'utilisation de cette instruction, faisons rapidement un petit tour de ces différentes situations.

Premièrement, lors du déclenchement d'une exception par exemple, juste avant le `throw`, vous ajoutez un `Debugger.Break()`. Ainsi en débogage, au lieu de catcher l'exception plus haut dans la pile des appels, vous vous arrêtez directement avant et avez ainsi accès à toute la pile des appels et à l'état actuel de l'application (état à l'origine de l'exception)

Le deuxième usage de cette instruction est la génération de points d'arrêts conditionnels. Dans le cas où vous devez déboguer le parcours d'un `for/while/foreach`, vous avez parfois envie de placer un point d'arrêt pour un élément spécifique de la collection parcourue. Certaines conditions sont plus facilement écrites dans le code qu'en utilisant les points d'arrêts conditionnels de Visual Studio.

Un exemple de point d'arrêt conditionnel en pseudo-code

```

Pour Chaque Client unClient De maListeDeClientsActif
  Si unClient = monClientSelectionneDansLaFenetre
    Debugger.Break
  Fin Si
Fin Pour Chaque

```

Dans le précédent exemple, j'ai simulé une condition d'arrêt assez simple, mais les phases de débogage impliquent parfois des séquences extraordinaires qui appellent des conditions d'arrêt précisément ciblées. Passons à présent à une autre méthode extrêmement pratique de la classe `Debugger`.

I-B - La méthode `Launch`

Commençons par mettre en place le décor. Vous avez développé une application console qui prend en paramètre tout un tas de données. L'exécution "à la main" est considérée comme impossible étant donnée la complexité des paramètres incriminés (par exemple, un programme qui accepte le contenu d'un fichier binaire généré par une application externe). Pour l'instant, votre application déclenche des exceptions lorsque les données ne peuvent pas être traitées correctement. Le problème est que pour déboguer une telle application, il faudrait pouvoir relancer l'exécution du programme avec les mêmes paramètres que ceux qui ont mené au déclenchement de l'exception. Or, nous venons de considérer que c'était impossible. La seule solution reste à essayer de deviner ce qui peut se passer dans le code en parcourant toutes les classes impliquées dans le traitement. C'est long, c'est fastidieux, c'est décourageant. Voici une solution.

Ne serait ce pas merveilleux de pouvoir attacher un débogueur à notre programme lors du déclenchement d'une exception et avoir ainsi la possibilité de voir l'état exact de toutes les variables au moment du déclenchement de l'exception ? Je vous vois d'ici avec vos yeux larmoyants et votre langue pendante, attendant comme le messie, la solution miracle que je m'apprête à vous offrir (oui, offrir, je suis comme ça).

Réjouissez-vous, il s'agit de la méthode Launch de l'objet Debugger qui a pour effet lors de l'exécution hors d'un environnement de demander à l'utilisateur quel débogueur attacher au programme afin de continuer l'exécution.

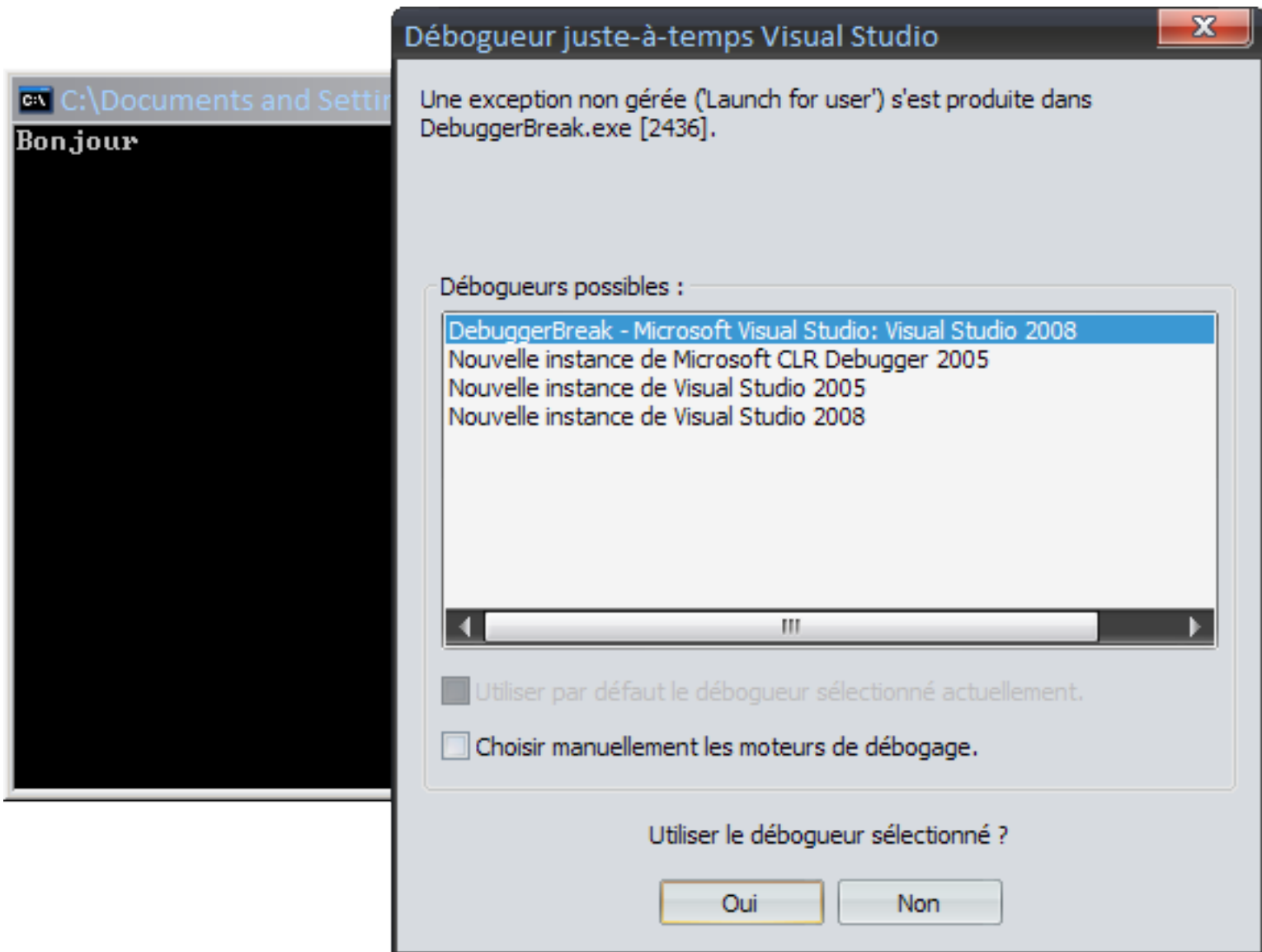
Un programme console demandant le lancement du débogueur

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Bonjour");

        Debugger.Launch();

        Console.WriteLine("Au revoir");
    }
}
```

Ce code exécuté dans Visual Studio affiche bêtement les deux messages, car le débogueur est déjà attaché. Par contre, le même programme exécuté seul produit l'affichage suivant.



L'instruction Debugger.Launch déclenche l'affichage d'une boîte de sélection du débogueur

Il vous suffit ensuite de déboguer comme vous le faites d'habitude.

i Pour savoir si le débogueur a bien été attaché, il suffit de tester la valeur de retour de la méthode Launch. Elle renvoie false si le débogueur n'a pas pu être attaché, et vrai dans le cas contraire (ou s'il était déjà attaché).

i Une dernière information sur la méthode Launch, elle fonctionne en mode Debug ou Release. Cependant en mode Release, vous n'aurez accès qu'au code machine pour déboguer, ce qui n'est pas des plus pratiques (sauf évidemment pour ceux qui ont fait MSIL en première langue).

Certain(e)s d'entre vous auront déjà saisi une autre utilité de cette méthode Launch. Je vous donne un indice, que ce passe t'il quand vous essayer de lancer à partir de Visual Studio le débogage d'un service Windows. Vous avez un beau message disant que vous ne pouvez pas. Et maintenant, imaginez le bien que pourrait vous faire le code suivant :

Un service tout simple et qui demande tout seul à être débogué. Merci qui ?

```
namespace ServiceLaunch
{
```

Un service tout simple et qui demande tout seul à être débogué. Merci qui ?

```
public partial class MonService : ServiceBase
{
    public MonService()
    {
        InitializeComponent();
    }

    protected override void OnStart(string[] args)
    {
#if DEBUG
        Debugger.Launch();
#endif
    }

    protected override void OnStop()
    {
    }
}
```

C'est beau, hein ?

I-C - La méthode Log

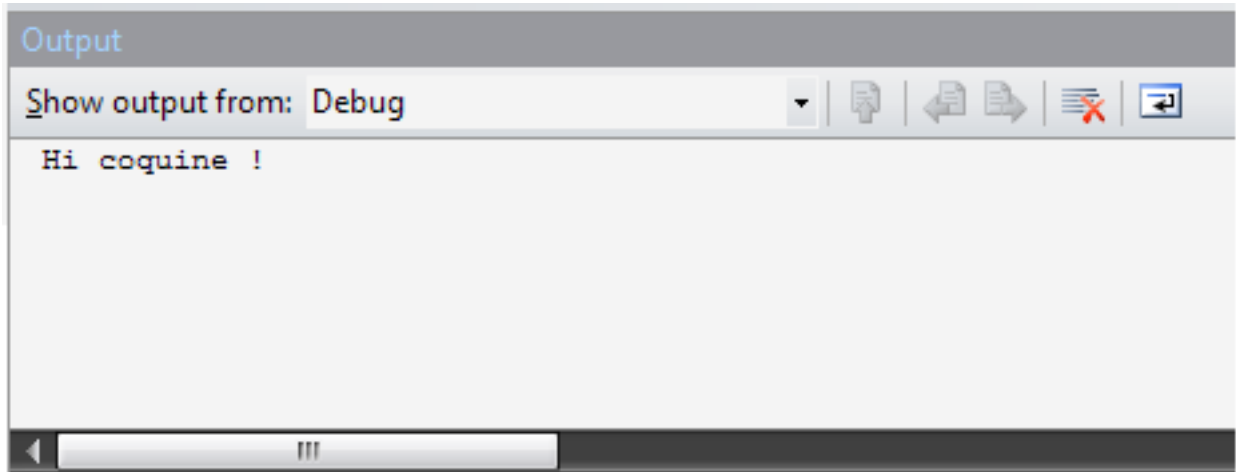
La dernière méthode que nous allons voir ensemble est la méthode Log. Elle permet d'envoyer des messages vers le débogueur qui en général l'affiche dans une fenêtre de sortie. Cette méthode prend en paramètre un entier qui représente le niveau de gravité du message. Le deuxième paramètre est la catégorie, c'est une chaîne limitée à 256 caractères. Le dernier paramètre est le message à proprement parler. L'utilité de cette méthode est à mon sens assez limitée, mis à part peut-être pour afficher des messages de progression ou d'information pour un traitement particulier en mode Debug (par exemple, le temps d'exécution d'une certaine méthode).

Un exemple simple utilisant la méthode Debugger.Log


```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Bonjour");

        Debugger.Log(0, "Information", "Hi coquine !");

        Console.WriteLine("Au revoir");
        Console.ReadLine();
    }
}
```



Le résultat de l'exécution de l'instruction `Debugger.Log`

 *Voilà pour la présentation des méthodes principales de la classe `Debugger`. S'il y avait une chose à retenir et qui vous sauvera la vie lors du débogage, c'est l'utilisation de la méthode `Launch` lors du démarrage d'un service.*

II - Utiliser l'objet Debug

L'objet Debug a deux fonctions principales :

- l'écriture de message dans la trace d'exécution,
- le contrôle de la logique d'exécution.

II-A - Ecriture des messages dans la trace

La trace est un système qui permet de garder des preuves du déroulement d'un programme. En ajoutant des messages dans la trace, vous la possibilité de reconstituer le cycle complet d'exécution d'un programme. Ceci dit, l'intérêt de la trace est de pouvoir y stocker des messages qui auront une utilité pour la suite. Il est inutile de garder une trace de "L'utilisateur a cliqué sur la combobox", par contre, il serait important de noter que "Quand l'utilisateur a cliqué sur la combobox, cette exception s'est déclenchée mais a été masquée à l'utilisateur car elle n'était pas critique".

La trace est un concept abstrait, cela signifie que si personne ne l'écoute, elle n'a aucun impact sur le programme. Pour écouter la trace, on lui adjoint des observateurs (Listeners). Les observateurs sont les flux dans lesquels la trace va être écrite. Par exemple, il existe un listener par défaut dans le framework appelé DefaultTraceListener. Ce listener redirige les messages de trace vers la méthode Debugger.Log (détaillée plus haut) et vers une fonction Win32 OutputDebugString. Nous ne nous intéresserons pas à la dernière fonction car elle dépasse le cadre de cet article. Il existe aussi plusieurs classes dans le framework pour rediriger la trace vers d'autres sorties.

Voici plusieurs observateurs définis dans le framework :

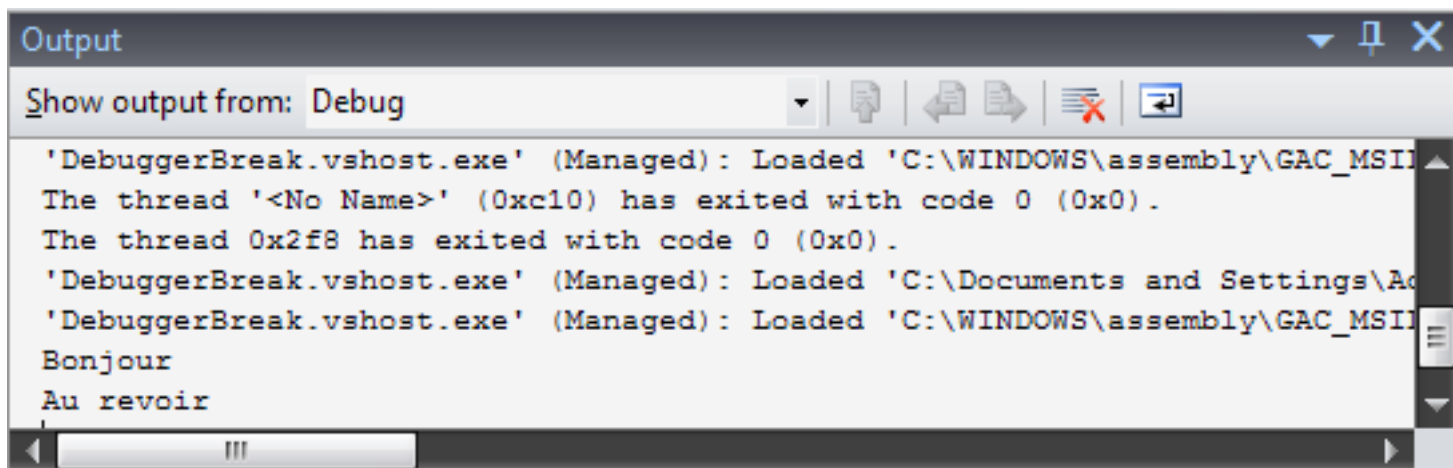
- DefaultTraceListener,
- TextWriterTraceListener (redirige les messages vers un fichier externe),
- EventLogTraceListener (utiliser l'Event Log pour stocker les messages).

Vous pouvez aussi utiliser votre propre observateur, il vous suffit de surcharger la classe TraceListener.

Un exemple simple d'utilisation de Debug.WriteLine

```
class Program
{
    static void Main(string[] args)
    {
        Debug.WriteLine("Bonjour");
        Debug.WriteLine("Au revoir");

        Console.ReadLine();
    }
}
```



Les messages s'affichent dans la fenêtre de sortie

Dans l'exemple précédent, nous avons utilisé la méthode WriteLine qui affiche le message spécifié dans le flux de trace et renvoie à la ligne. Il existe en fait quatre méthodes pour écrire dans la trace :

- Write : écrit simplement le message;
- WriteIf : écrit le message si la condition spécifiée est vraie;
- WriteLine : écrit le message et revient à la ligne;
- WriteLineIf : écrit le message et revient à la ligne si la condition spécifiée est vraie.

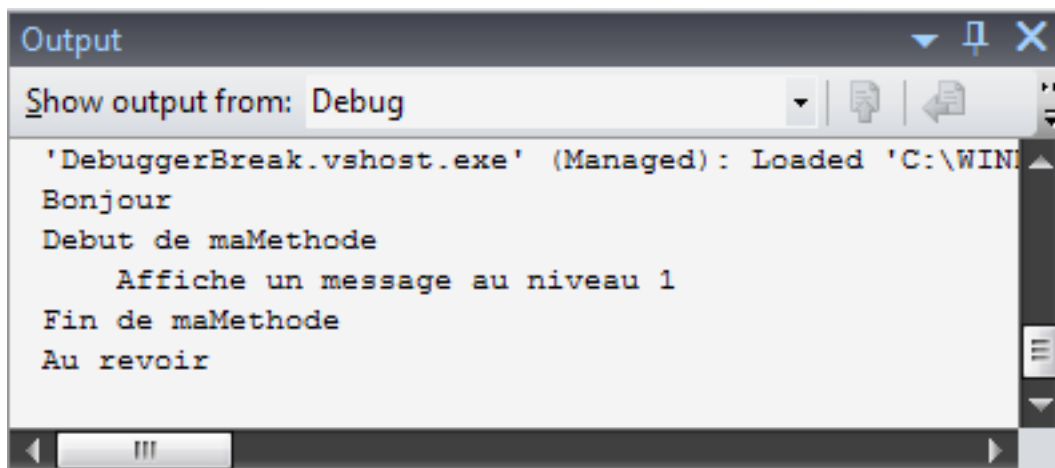
Vous pouvez aussi contrôler l'indentation des messages que vous envoyez vers la trace. Par exemple, si vous souhaitez tracer plusieurs choses dans le corps d'une méthode, vous pouvez indenter les messages internes à la méthode pour signifier que les messages ont été postés dans le corps de la méthode.

Exemple d'utilisation de l'indentation

```
class Program
{
    static void Main(string[] args)
    {
        Debug.WriteLine("Bonjour");
        maMethode();
        Debug.WriteLine("Au revoir");

        Console.ReadLine();
    }

    private static void maMethode()
    {
        Debug.WriteLine("Debut de maMethode");
        Debug.Indent();
        Debug.WriteLine("Affiche un message au niveau " + Trace.IndentLevel);
        Debug.Unindent();
        Debug.WriteLine("Fin de maMethode");
    }
}
```



Les messages sont indentés dans la sortie du débogueur.

Le niveau d'indentation peut être directement contrôlé par l'accès à la propriété **IndentLevel**. La taille d'un niveau d'indentation est modifiable par la propriété **IndentSize**.

! *Je vous recommande vivement de remettre l'indentation à son niveau initial lorsque vous quittez une méthode qui a modifié le niveau ou la taille de l'indentation. De cette manière le formatage de la trace n'est pas faussé pour les messages suivants. C'est une sorte de "laissez la trace dans l'état dans lequel vous l'avez trouvée lorsque vous êtes arrivés".*

II-B - Contrôler le déroulement d'un programme

Limiter l'utilisation de l'objet Debug à la simple écriture dans la trace serait très réducteur. C'est un peu comme si on disait d'une boîte aux lettres qu'elle sert à occuper le sommet du piquet qui traîne à côté du portail.

II-B-1 - La méthode Assert

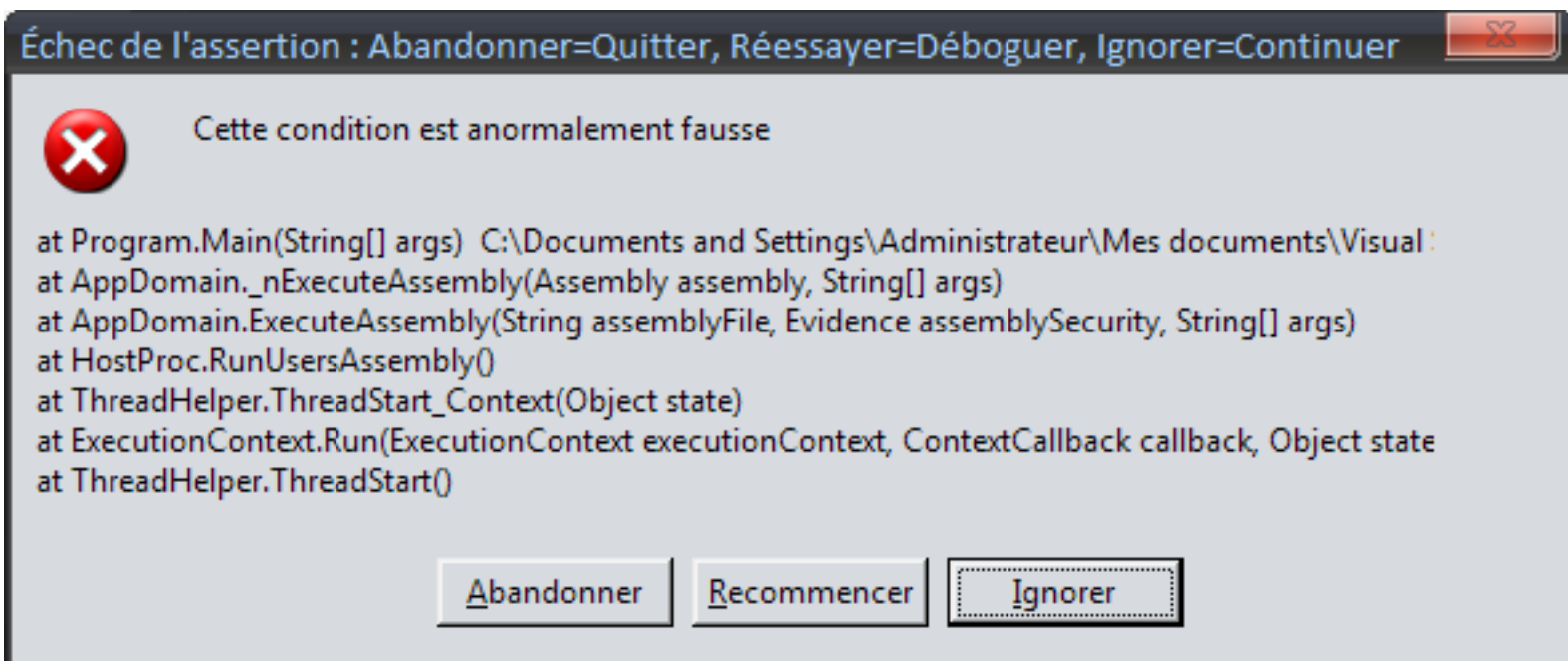
En fait Debug permet d'insérer des tests tout au long du programme. Si ce test à un moment donné est faux, un message va être affiché à l'utilisateur (sous réserve que le listener par défaut n'est pas été supprimé). Cela permet de définir des conditions qui devraient toujours être vraies. Par exemple, si l'application est lancée et gère les utilisateurs, la variable contenant l'utilisateur ne devrait jamais être null. Ces tests sont représentés par la méthode Assert. Cette méthode possède plusieurs surcharges :

- Assert(Boolean) : affiche un MessageBox contenant la pile des appels si la condition est *false*.
- Assert(Boolean, String) : affiche un MessageBox contenant le message spécifié si la condition est *false*.
- Assert(Boolean, String, String) : affiche un MessageBox contenant les deux messages spécifiés si la condition est *false*. Les deux messages sont en général utilisés sous la forme "message succinct" et "message détaillé".

Exemple d'utilisation de la méthode Debug.Assert

```
class Program
{
    static void Main(string[] args)
    {
        Debug.Assert(args == null, "Cette condition est anormalement fausse");
    }
}
```

Le code précédent teste si des arguments ont été passés sur la ligne de commande. Dans le cas où il n'y a pas de paramètres, la condition va être vérifiée et le message affiché.



Un message typique d'assertion (la fenêtre a été artificiellement tronquée)

II-B-2 - La méthode Fail

La deuxième méthode intéressante pour vérifier certaines conditions est la méthode Fail. Nous n'allons pas détailler cette méthode, car il s'agit d'un Assert avec la condition toujours *true*. A chaque fois que le code va arriver sur la méthode Fail, le message d'erreur va être affiché.

Comparaison de Fail et Assert

```
// Assert
Debug.Assert(currentUser == null, "Utilisateur ne devrait jamais être null");

// Serait équivalent à
if (currentUser == null)
    Debug.Fail("Utilisateur ne devrait jamais être null");
```

II-C - Utilisation des fichiers de configuration

Une fonctionnalité intéressante de la classe Debug est la possibilité de contrôler les sorties de trace et des assertions en utilisant une section spéciale du fichier **.config**.

Cette section permet de configurer un fichier de sortie pour les assertions

```
<configuration>
  <system.diagnostics>
    <assert assertuienabled="true" logfile="Log.log" />
  </system.diagnostics>
</configuration>
```

Voici le contenu du fichier de Log après le déclenchement de l'assertion

```
---- ÉCHEC DE L'ASSERTION DE DÉBOGAGE----
---- Message court d'assertion ----
Cette condition est anormalement fausse
---- Message long d'assertion ----

at Program.Main(String[] args) C:\Documents and Settings\...\Program.cs(13)
```

Voici le contenu du fichier de Log après le déclenchement de l'assertion

```
at AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
at AppDomain.ExecuteAssembly(String assemblyFile, Evidence assemblySecurity, String[] args)
at HostProc.RunUsersAssembly()
at ThreadHelper.ThreadStart_Context(Object state)
at ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state)

at ThreadHelper.ThreadStart()
```


Ce code permet d'ajouter un nouvel observateur pour la trace, il retire aussi l'observateur par défaut.

```
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="4">
      <listeners>

        <add name="traceWriter" type="System.Diagnostics.TextWriterTraceListener" initializeData="trace.log" />
        <remove name="Default" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

III - Utiliser les attributs de débogage

Le framework intègre plusieurs attributs utilisables pour décorer les classes mais aussi leurs méthodes, attributs et propriétés. Ces attributs permettent de modifier, de personnaliser la manière dont le débogueur va traiter les objets et leurs éléments. Le masquage complet d'un élément est le dernier stade de cette personnalisation. Passons sans plus tarder à la description des différents attributs. Il est important de comprendre que ces attributs sont destinés à être interprétés par le débogueur, ils n'ont **aucune influence sur le fonctionnement réel** du programme.

 *Il existe une option dans Visual Studio nommé "Just My Code" qui permet de limiter le débogage uniquement au code considéré comme "écrit par l'utilisateur", à la différence par exemple du code généré par Visual Studio ou d'autres outils externes. Nous verrons avec la description de chaque attribut la manière dont cette option est prise en compte.*

III-A - DebuggerBrowsableAttribute

Cet attribut détermine la visibilité initiale d'un membre de classe dans la fenêtre de débogage. Il peut être appliqué sur les propriétés et attributs d'une classe.

Voici une brève description des différentes valeurs possibles de cet attribut :

- `DebuggerBrowsableState.Never` : L'élément n'est pas visible dans la fenêtre de débogage
- `DebuggerBrowsableState.Collapsed` : L'élément est visible mais sous forme réduite (dans le cas où l'élément contient des sous-éléments), il s'agit de la valeur par défaut.
- `DebuggerBrowsableState.RootHidden` : L'élément est masqué mais ses éléments enfants sont insérés directement dans la vue actuelle.

III-A-1 - DebuggerBrowsableState.Never

L'image suivante illustre l'utilisation de la valeur `DebuggerBrowsableState.Never`. Le membre décoré avec cet attribut + valeur n'apparaît pas dans la fenêtre de visualisation.

```

namespace DemoDebuggerBrowsableAttribute
{
    class Demo
    {
        static void Main(string[] args)
        {
            DemoObject obj = new DemoObject();
        }
    }

    class DemoObject
    {
        private int uneValeurVisible = 0;
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private int uneValeurInvisible = 0;
    }
}
    
```

Exemple de comportement d'un attribut de classe décoré avec DebuggerBrowsable(DebuggerBrowsableState.Never)

III-A-2 - DebuggerBrowsableState.RootHidden

L'image suivante illustre l'utilisation de la valeur DebuggerBrowsableState.RootHidden. Le membre décoré avec cet attribut + valeur n'apparaît pas dans la fenêtre de visualisation, cependant ses éléments enfants sont directement ajoutés dans la vue de l'objet. Cette pratique est très utile pour faciliter le débogage d'objet contenant des listes ou des objets complexes (par exemple, un objet client qui contient un objet adresse)

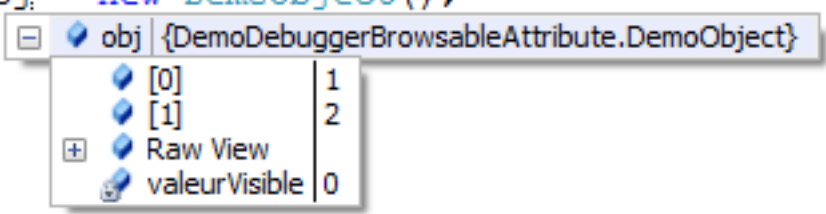
```

namespace DemoDebuggerBrowsableAttribute
{
    class Demo
    {
        static void Main(string[] args)
        {
            DemoObject obj = new DemoObject();
        }
    }

    class DemoObject
    {
        private int valeurVisible = 0;
        [DebuggerBrowsable(DebuggerBrowsableState.RootHidden)]
        private DemoList valeurListe = new DemoList { 1, 2 };
    }

    class DemoList : List<int> { }
}

```



*Exemple de comportement d'un attribut de classe décoré avec
[DebuggerBrowsable(DebuggerBrowsableState.RootHidden)]*

Dans cet exemple les éléments contenus dans valeurListe sont directement attachés à la vue de l'objet. En fonctionnement normal, ces éléments auraient été visuellement contenus dans un élément valeurListe.

III-B - DebuggerDisplayAttribute

Cet attribut permet de contrôler le texte affiché dans la fenêtre de visualisation. Il suffit de spécifier dans la chaîne de format le texte ainsi que les différents attributs ou propriétés de la classe à afficher.

```

namespace DemoDebuggerAttributes
{
    class Demo
    {
        static void Main(string[] args)
        {
            DemoObject obj = new DemoObject ();
        }
    }

    class DemoObject
    {
        [DebuggerDisplay("Valeur = {valeurVisible}")]
        private int valeurVisible = 0;

        [DebuggerDisplay("Nombre d'éléments = {valeurListe.Count}")]
        private DemoList valeurListe = new DemoList { 1, 2 };
    }

    class DemoList : List<int> { }
}

```

L'affichage des membres de l'objet est modifié par les attributs appliqués

Cette technique permet de personnaliser complètement l'affichage d'un objet et de ses membres. Les variables entre accolades doivent correspondre à des membres accessibles en utilisant le mot-clé `this` au moment de l'exécution.

i Si un objet possède une méthode `ToString()` spécifique, le retour de `ToString` sera utilisé pour afficher l'objet dans la fenêtre de visualisation. Si `ToString()` et un attribut `DebuggerDisplay` sont définis pour l'objet courant, c'est la valeur de l'attribut qui aura priorité.

```

class Demo
{
    static void Main(string[] args)
    {
        DemoObject obj = new DemoObject();
    }
}

class DemoObject
{
    public override string ToString()
    {
        return String.Format("Valeur {0} et {1} élément(s)",
            valeurVisible, valeurListe.Count);
    }
}

```

Affichage du retour de ToString() dans l'entête

III-C - DebuggerHiddenAttribute

Cet attribut "masque" la méthode/propriété au débogueur, ou plus précisément, que le débogueur ne pourra pas entrer dans la méthode/propriété même si vous appuyez sur F11 (step into) lors du pas-à-pas. Cela empêche aussi de définir un point d'arrêt dans un membre marqué par cet attribut. On utilise cet attribut pour les méthodes dont le contenu n'a pas lieu d'être parcouru lors du débogage. Cet attribut ne tient pas compte de l'option "Just My Code".

III-D - DebuggerNonUserCodeAttribute

Cet attribut permet de marquer des sections de code comme ne faisant pas partie du "code utilisateur" en référence à l'option "Just My Code". Cela signifie que si l'option "JMC" est activée, le code en question ne sera pas "débogable". Sa principale utilité est de masquer au débogueur les sections de code générées par un outil. Cet attribut permet tout de même de rétablir la possibilité de déboguer le code en désactivant l'option "JMC" (chose que DebuggerHidden ne permet pas).

III-E - DebuggerStepThroughAttribute

Cet attribut informe le débogueur qu'il ne doit pas entrer dans le corps de l'élément décoré par cet attribut. L'option Just My Code n'est pas gérée par cet attribut. Il peut être utilisé sur une classe, une structure, une méthode ou un constructeur. La différence avec l'attribut DebuggerHidden est que ce dernier n'autorise pas d'ajouter un point d'arrêt dans un membre décoré, tandis que DebuggerStepThrough le permet.

III-F - DebuggerStepperBoundaryAttribute

L'attribut DebuggerStepperBoundary est celui dont la description est la plus absconse. Son utilisation est liée à l'attribut DebuggerNonUserCode. Lorsque le débogueur arrive sur une méthode décorée par DebuggerNonUserCode, l'exécution reprend au prochain bloc de code considéré comme "code utilisateur". Le problème est lorsque ce code est exécuté par un autre thread dans le cas d'une application multithread, le code dans lequel le débogueur saute peut n'avoir aucun rapport avec le contexte de débogage avant le saut. L'attribut DebuggerStepperBoundary permet de résoudre ce problème. Cet attribut informe le débogueur que le code suivant

cet attribut doit être exécuté sans "pas à pas" (en mode continu). Dans la pratique, lorsque le débogueur rencontre cet attribut, il se comporte comme si l'utilisateur avec pressé F5 (continuer l'exécution).

III-G - DebuggerTypeProxyAttribute

Cet attribut permet de définir un type qui sera utilisé en lieu et place de l'objet réel dans la fenêtre de débogage.

Prenons par exemple le code suivant :

```
class Program
{
    static void Main(string[] args)
    {
        Client monClient = new Client { Nom = "Moustachu", Prenom = "Georges" };
    }
}

[DebuggerTypeProxy(typeof (Client.ClientProxy))]
class Client
{
    public String Nom { get; set; }
    public String Prenom { get; set; }

    internal class ClientProxy
    {
        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        private Client client;

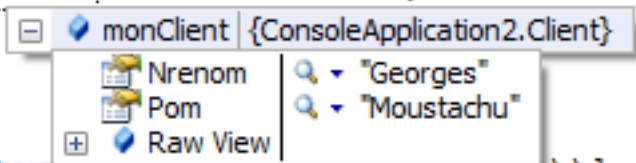
        public String Pom { get { return client.Nom; } set { client.Nom = value; } }
        public String Nrenom { get { return client.Prenom; } set { client.Prenom = value; } }

        public ClientProxy(Client client) { this.client = client; }
    }
}
```

La classe ClientProxy définie dans la classe Client est la classe qui va être utilisée pour visualiser les instances de Client dans la fenêtre de visualisation du débogueur. Vous remarquerez que le constructeur de ClientProxy prend en paramètre un objet de Client. C'est ce constructeur qui est appelé par le débogueur pour construire l'objet qui va être utilisé lors de la visualisation. Le paramètre ne doit pas forcément être du type exact de la classe, il faut simplement que ce soit un type dans lequel on peut caster une référence du type d'origine. Dans mon cas, il faut que ce soit un type dans lequel on peut caster un Client (à savoir : Client elle-même ou Object).

```
static void Main(string[] args)
{
    Client monClient = new Client { Nom = "Moustachu", Prenom = "George"
}

[DebuggerTypeProxy(typeof (Client.ClientProxy))]
```



Résultat de l'utilisation de l'attribut DebuggerTypeProxy



Le prochain paragraphe s'adresse à tous ceux qui prennent parfois plaisir à torturer leurs collègues en utilisant des constructions ésoériques et des fonctionnalités cachées. Je recommande aux autres de passer directement à la section suivante.

Vous aurez sans doute remarqué le choix judicieux du proxy pour l'objet Client, il contient deux propriétés nommées Pom et Nrenom. Ce proxy est juste assez étrange pour que l'on ne soupçonne pas forcément la farce. Le problème est que la classe ClientProxy est située dans la classe Client, il ne faut pas chercher bien loin pour la trouver. Qu'à cela ne tienne, il est possible de définir une autre classe externe à la classe Client comme type pour le proxy. Ainsi, on gagne en subtilité. Cependant il reste encore un problème, l'attribut qui décore la classe Client en elle-même. Un collègue un peu méfiant aura tôt fait de subodorer la farce en voyant cet attribut. N'y aurait il pas un moyen de masquer cet attribut ? En fait non. Mais il est possible de le déplacer ailleurs, car il peut avoir une portée dans toute l'assembly. Pour cela, il faut un peu modifier son écriture :

```
[assembly: DebuggerTypeProxy(typeof(DebuggerTypeProxyExample.ClientProxy),
    Target = typeof(DebuggerTypeProxyExample.Client))]
```

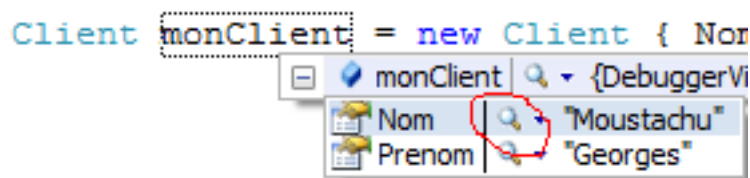
Avec cette déclaration, vous pouvez associer depuis n'importe quel fichier, un proxy défini n'importe où dans le projet, à n'importe quelle classe accessible depuis l'assembly.

Des heures de fun garanties. Satisfait ou remboursé :)

III-H - DebuggerVisualizerAttribute

Il ne nous reste qu'un dernier attribut à voir ensemble, il s'agit de DebuggerVisualizer. Il s'inscrit dans la lignée de DebuggerTypeProxy, dans le sens où il sert à définir un visualiseur pour l'objet. Un visualiseur est une fenêtre d'affichage des données de l'objet débogué. Dans la pratique, le débogueur et le visualiseur communiquent par l'intermédiaire de Stream dans lequel on sérialise/dé-sérialise les objets à échanger.

Pour rappel, le visualiseur est la fenêtre affichée par Visual Studio lorsqu'on clique sur la petite loupe située à côté des variables dans la fenêtre de visualisation.



Le symbole du visualiseur

A la différence du DebuggerTypeProxy, le visualiseur travaille à partir d'une autre assembly, il peut être considéré comme un plugin de Visual Studio. Les visualiseurs (dans une assembly à part) doivent d'ailleurs être stockés dans le dossier Mes Documents\Visual Studio 200x\Visualizers. Ils sont alors chargés par le débogueur et utilisés lorsque l'utilisateur le demande.

Reprenons notre classe de l'exemple précédent :

```
namespace DebuggerVisualizerExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Client monClient = new Client { Nom = "Moustachu", Prenom = "Georges" };
        }
    }

    [Serializable]
    public class Client
    {
        public String Nom { get; set; }
        public String Prenom { get; set; }
    }
}
```

```
}

```

La classe pour laquelle on souhaite créer un visualiseur doit être sérialisable (pour être transférée dans les Stream évoqués précédemment). Au niveau du programme standard, il n'y a rien d'autre à faire, c'est l'assembly qui contient les visualiseurs qui se charge de mettre en place les liens entre classes et visualiseurs. Voyons alors comment déclarer un visualiseur. Le code ci-dessous doit être ajouté dans un nouveau projet de type "bibliothèque de classes" et avoir une référence vers l'assembly contenant l'objet visualisé (ici, il s'agit de mon application console).

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

// Ne pas oublier d'ajouter la référence à l'assembly Microsoft.VisualStudio.DebuggerVisualizers
[assembly: DebuggerVisualizer(typeof(ClientDebuggerVisualizer.ClientVisualizer),

    // Créer un type spécifique héritant de VisualizerObjectSource pour pouvoir modifier l'objet débogué.
    typeof(VisualizerObjectSource),
    Target = typeof(DebuggerVisualizerExample.Client),
    Description = "Visualiseur pour les Client")]

namespace ClientDebuggerVisualizer
{
    public class ClientVisualizer : DialogDebuggerVisualizer
    {
        protected override void Show(IDialogVisualizerService windowService, IVisualizerObjectProvider
objectProvider)
        {
            Client monClient = (Client)objectProvider.GetObject();
            MessageBox.Show(String.Format("Nom : {0}, Prénom : {1}", monClient.Nom, monClient.Prenom));
        }
    }
}

```

Le visualiseur doit forcément hériter de DialogDebuggerVisualizer (une classe abstraite définissant simplement la méthode Show). C'est dans l'implémentation de la méthode Show qu'on va afficher notre propre fenêtre de visualisation. Dans cet exemple, il s'agit d'un bête MessageBox, il est évident qu'il serait plus utile de définir une fenêtre plus fonctionnelle. Ce visualiseur ne permet pas non plus de modifier l'objet actif (ici un Client), il faudrait pour cela utiliser une spécialisation de VisualizerObjectSource plutôt que le type VisualizerObjectSource lui-même (et utiliser celle-ci dans la déclaration de l'attribut). Je laisse ce dernier point comme un exercice pour le lecteur.

Une fois notre assembly compilée, il suffit de la déposer dans le répertoire Visualizers (cité précédemment) et à lancer le débogage de notre application console. Une petite loupe devrait maintenant apparaître en face des Client dans la fenêtre de visualisation. Le clic sur cette loupe doit faire apparaître le MessageBox configuré dans le visualiseur.

Conclusion

Vous avez maintenant en main toutes les clés disponibles dans le framework pour vous faciliter le débogage ainsi que l'utilisation du débogueur. Il ne vous reste plus qu'à vous habituer à les utiliser, vous gagnerez ainsi un temps précieux.

Remerciements

Je tiens à remercier Aspic pour sa relecture attentive, ainsi que tous les membres de la rédaction pour leurs commentaires.