

Autour du design pattern view presenter

par Jean-Alain Baeyens ([autres articles](#))

Date de publication : 05/03/2008

Dernière mise à jour : 15/03/2008

Par l'utilisation du pattern MVP vous allez découvrir comment créer une application pouvant utiliser différents interfaces graphiques (Windows, Console, Web, ...) en minimisant au maximum la redondance de code.

- I - Introduction
- II - Le modèle.
 - II-A - La théorie
 - II-B - Implémentation classique du modèle
- III - Conception multi plateforme.
 - III-A - La couche de présentation
 - III-A-1 - La classe Presenter
 - III-A-2 - L'interface IViewer
 - III-B - La couche Viewer
 - III-B-1 - L'interface Windows
 - III-B-2 - L'interface console
 - III-B-3 - L'interface Web
- IV - Placer le Presenter derrière un service web.
- V - Conclusion
- VI - Référence
- VII - Sources

I - Introduction

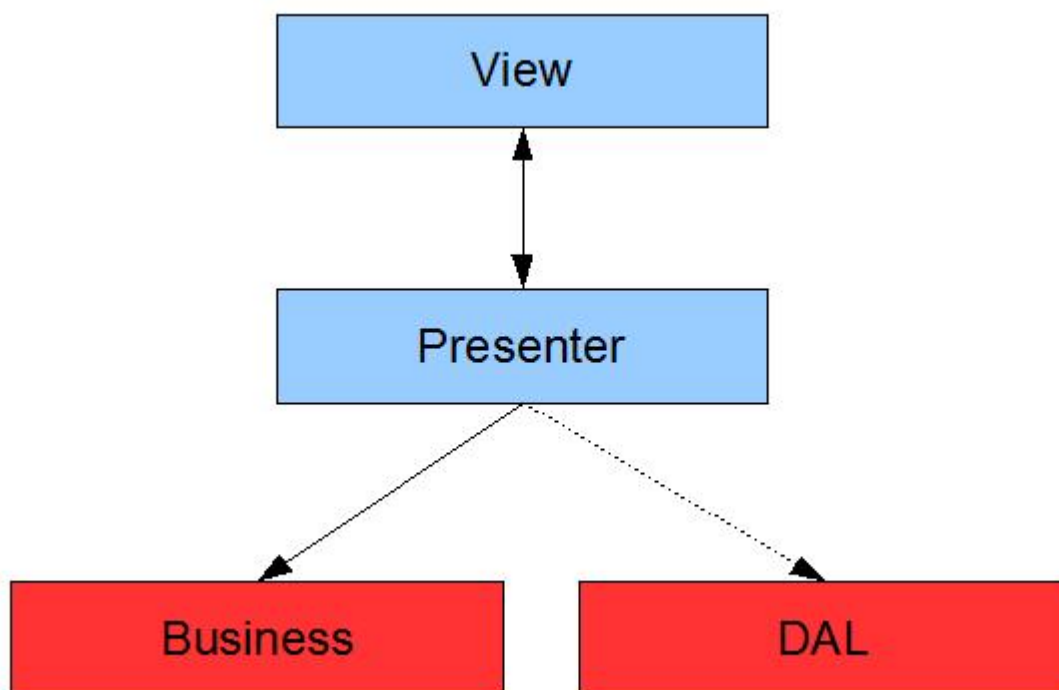
A l'origine, mon intention n'était pas de vous parler spécifiquement du pattern "Model View Presenter" mais bien de réfléchir à la réalisation d'une architecture permettant d'utiliser différents interfaces graphiques tout en minimisant la redondance de code. La couche intermédiaire ainsi créée doit prendre en charge non seulement des tâches comme la validation, la persistance, le chargement de données mais aussi le cas échéant, la gestion des transitions entre les étapes d'encodage. Cette réflexion m'a conduit à une implémentation du pattern MVP, ce qui démontre si besoin en est de l'intérêt des patterns.

Après un aperçu théorique et un exemple de l'implémentation classique, je vais vous présenter ma vision de l'implémentation du modèle MVP.

II - Le modèle.

II-A - La théorie

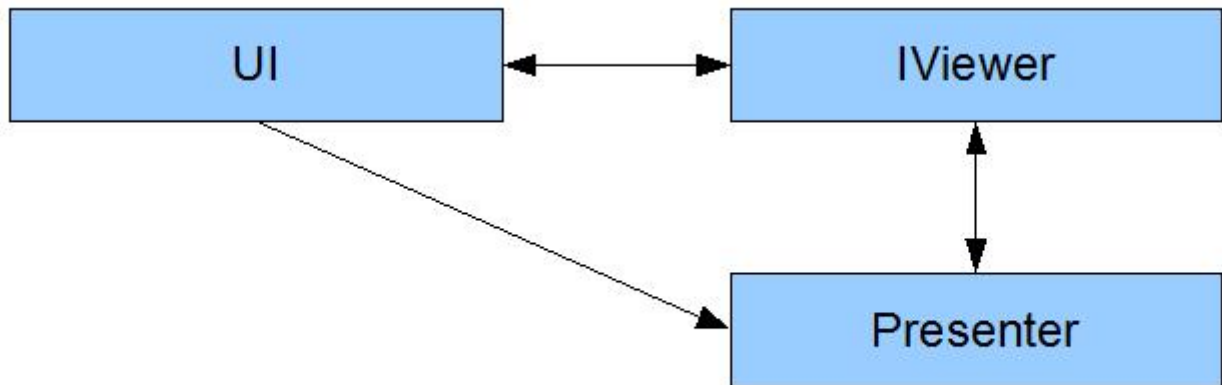
Le modèle "View Presenter" consiste à diviser la couche d'interface graphique en deux. La partie la plus basse, le "Presenter", sera la seule à communiquer avec la couche métier et éventuellement avec celle d'accès aux données dans les limites où c'est acceptable dans votre architecture globale.



Architecture globale

Comme le "Presenter" ne se contente pas de répondre à des sollicitations de l'interface graphique mais prend également en charge son pilotage, la dépendance est bidirectionnelle entre les deux parties. Afin de minimiser cette dépendance, le "Presenter" communique avec le "View" au travers d'un interface.

L'interface est défini dans le "Presenter" alors que le "View" doit contenir une classe qui l'implémente.

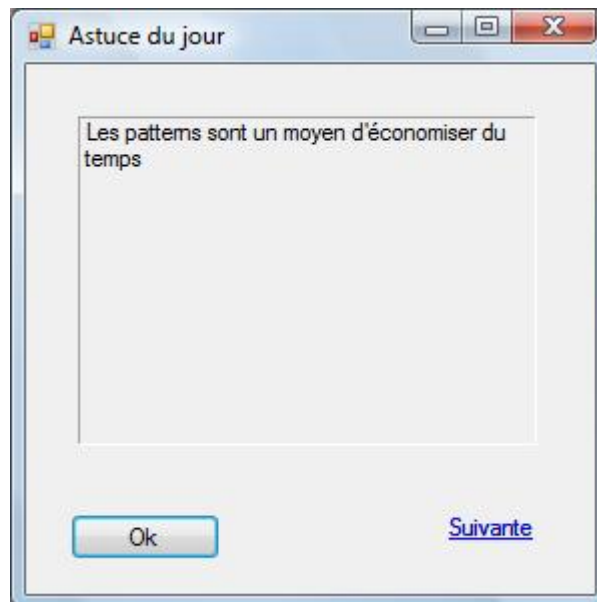


Le pattern MVP.

Vous pouvez opter pour une architecture stricte où l'interface graphique doit impérativement passer par le Viewer pour communiquer avec le Presenter ou pour une architecture plus souple où il est autorisé à appeler directement le Presenter. Cette dernière offre l'avantage d'un code plus léger mais évidemment crée plus de dépendances. A vous de voir en fonction de votre projet si vous optez pour plus de simplicité ou un couplage plus faible.

II-B - Implémentation classique du modèle

A titre d'exemple, nous allons réaliser une fenêtre d'astuce.



Selon le principe du pattern MVP, le code dans la fenêtre doit être minimal. C'est pourquoi en temps normal, nous y aurions trouvé les appels aux services business mais ici rien de tout cela. La fenêtre se contente d'émettre des événements lorsqu'une action est demandée. Notez uniquement l'instanciation du Presenter qui est associé à la fenêtre lors du chargement de celle-ci.

```
public partial class AstuceForm : Form, IAstuceView
```

```

{
    public AstuceForm()
    {
        InitializeComponent();
    }

    public string Astuce
    {
        set
        {
            this.astuceLabel.Text = value;
        }
    }

    private AstucePresenter presenter;
    public event EventHandler CloseRequested;
    public event EventHandler NextRequested;

    private void okButton_Click(object sender, EventArgs e)
    {
        CloseRequested(this, EventArgs.Empty);
    }

    private void nextLinkLabel_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
    {
        NextRequested(this, EventArgs.Empty);
    }

    private void AstuceForm_Load(object sender, EventArgs e)
    {
        presenter = new AstucePresenter(this);
    }
}


```

Comme prévu dans le modèle, la fenêtre implémente un interface. Le presenter ne connaît que cet interface. Il doit lui fournir les moyens de communiquer avec le viewer. Le Presenter a besoin de pouvoir lui transmettre le texte de l'astuce et de s'abonner aux événements.

```

interface IAstuceView
{
    string Astuce { set; }
    event EventHandler CloseRequested;
    event EventHandler NextRequested;
    void Close();
}

```

 *Notez que l'interface définit la méthode Close que nous n'avons pas spécifiquement implémentée mais elle est héritée de la classe Form.*

Dans le constructeur, le Presenter s'abonne aux événements et transmet le texte de l'astuce à afficher. Le constructeur reçoit le viewer et le conserve pour communiquer avec lui.

```

class AstucePresenter
{
    private IAstuceView viewer;
    private int astuceCourant;

    public AstucePresenter(IAstuceView view)
    {
        viewer = view;
        astuceCourant = 0; //normalement valeur random
        viewer.Astuce = getAstuceDuJour();
        viewer.CloseRequested += OnCloseRequested;
    }
}

```

```
viewer.NextRequested += OnNextRequested;
}

private string getAstuceDuJour()
{
    // Pour l'exemple, il n'y en a que 2.
    if (astuceCourant == 0)
    {
        astuceCourant++;
        return "Les patterns sont un moyen d'économiser du temps";
    }
    else
    {
        astuceCourant = 0;
        return "Utilisez le modèle View presenter pour mieux valider votre application par des tests unitaires";
    }
}

private void OnCloseRequested(object sender, EventArgs e)
{
    viewer.Close();
}

private void OnNextRequested(object sender, EventArgs e)
{
    viewer.Astuce = getAstuceDuJour();
}
}
```

Le Presenter décide des actions à mener lorsqu'un événement survient sur le viewer. Par exemple lorsque vous cliquez sur suivant, la fenêtre déclenche l'événement "NextRequested"

III - Conception multi plateforme.

Après l'exemple d'implémentation classique du modèle réalisé dans le paragraphe précédent, nous allons utiliser le même concept pour rendre une application multiplateform et donner au Presenter la possibilité de gérer un interface graphique composé d'écrans successifs.

Afin de bien comprendre, nous allons réaliser un exemple simple. Le code qui n'est pas directement utile pour le modèle sera minimisé au maximum. Il s'agira de demander le code d'un utilisateur ainsi que son type. Sur base de ces informations, son profil sera affiché.

La couche de présentation ("Presenter"), prendra en charge la validation des données introduites dans la couche de visualisation ("View"), le chargement du contenu à présenter dans une liste et la gestion de la succession des opérations liées à l'interface utilisateur.

Dans cet exemple et afin d'obtenir un code plus court et ainsi de faciliter la compréhension, nous opterons pour la version souple dupattern.

Sur base de la même couche de présentation, nous construirons une application console, windows et web.

III-A - La couche de présentation

Pour la couche de présentation, nous allons créer un projet de type librairie de classe. Appelons ce projet "UIPresenter".

Il va contenir deux éléments, la classe "Presenter" et l'interface "IViewer".

III-A-1 - La classe Presenter

Cette classe doit stocker les différentes informations. Pour réaliser l'exemple, ces informations sont le code, le type, le nom et le prénom. Nous devons donc ajouter les propriétés correspondantes.

Les propriétés

```
private string nom;
public string Nom
{
    get { return nom; }
    set { nom = value; }
}

private string prenom;
public string Prenom
{
    get { return prenom; }
    set { prenom = value; }
}

private string type;
public string Type
{
    get { return type; }
    set { type = value; }
}

private string code;
```

Les propriétés

```
public string Code
{
    get { return code; }
    set { code = value; }
}
```

La classe Presenter devra également connaître le viewer qu'elle va utiliser. Comme il est dépendant de l'interface graphique, c'est lui qui devra instancier le Presenter et lui transmettre sa propre référence.

Le constructeur

```
private IViewer viewer;

public Presenter(IViewer callerInstance)
{
    viewer = callerInstance;
}
```

Nous devons également disposer d'un moyen de démarrer le traitement réalisé par notre Presenter. Il est possible d'opter pour un déclenchement automatique en plaçant le code adéquat dans le constructeur mais ici nous opterons pour un démarrage explicite au moyen d'une méthode. Cette méthode prend en charge le lancement de la première opération, c'est à dire la demande du code utilisateur et son type.

La méthode Start

```
public void Start()
{
    viewer.CallRequestUserCode();
}
```


Comme vous le voyez, il s'agit pour le Presenter de demander au Viewer d'afficher l'interface utilisateur permettant cette saisie. Notre interface IViewer devra contenir une définition pour "CallRequestUserCode".

Ce premier interface utilisateur est composé de deux éléments, le choix du type dans une liste et l'encodage du code. Après la saisie, les données sont validées. Le Presenter décide alors de la suite à donner. Dans notre exemple tout simple, il s'agit uniquement du passage à l'écran suivant.

Pour réaliser ces derniers objectifs, la classe Presenter doit également offrir une méthode permettant la validation, une méthode permettant de remplir la liste des types et une permettant de lui signaler que l'encodage est terminé.

La méthode LoadListType

```
public List<string> LoadListType()
{
    List<string> types = new List<string>();
    types.Add("Local");
    types.Add("Domaine");
    return types;
}
```

 *Utilisez des types aussi généraux que possible car à priori la classe Presenter n'a aucune idée de la manière dont la liste sera présentée.*

La méthode Validate

```
public bool Validate()
```

La méthode Validate

```
{  
    return (code.Length > 2);  
}
```

La méthode CodeInputEnded

```
public void CodeInputEnded()  
{  
    loadProfile(code);  
    viewer.CallDisplayUserProfile();  
}  
  
private void loadProfile(string code)  
{  
    // TODO call business service  
    nom = "Son nom";  
    prenom = "Son prénom";  
}
```


Dans cette dernière partie du code, le Presenter demande au Viewer d'afficher le profil de l'utilisateur. Notre interface IViewer doit définir la méthode CallDisplayUserProfile.

III-A-2 - L'interface IViewer

Cette interface doit définir les méthodes qui devront obligatoirement être implémentées pour permettre au Presenter d'agir sur l'UI. Ces méthodes sont déterminées par les procédures mises en oeuvre dans la classe Presenter.

L'interface IViewer

```
namespace UIPresenter  
{  
    public interface IViewer  
    {  
        void CallRequestUserCode();  
        void CallDisplayUserProfile();  
    }  
}
```

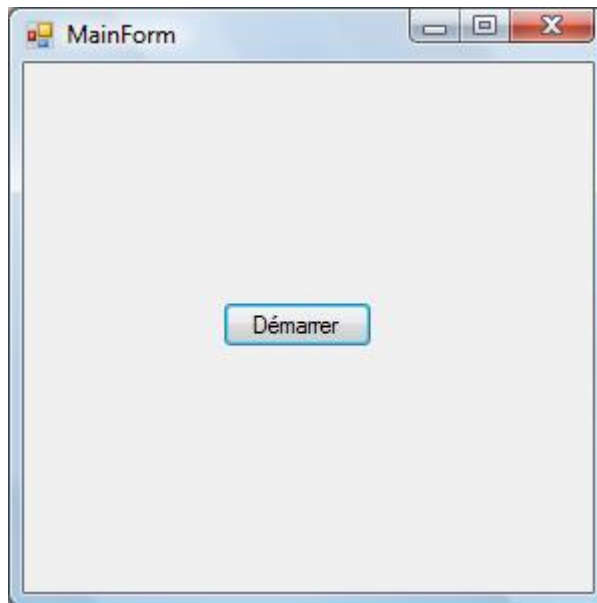
 Comme vous le constatez, les méthodes ne renvoient aucune valeur. Ceci est préférable si vous souhaitez supporter une majorité d'environnement. Certains environnements ne sont pas capables de renvoyer une valeur. C'est par exemple le cas en ASP.NET. Les valeurs en retour seront transmises à la classe Presenter en utilisant les propriétés.

III-B - La couche Viewer

Cette couche est dépendante du type d'interface utilisateur choisi. Dans cet article, nous allons réaliser un interface Windows, un console et un Web.

III-B-1 - L'interface Windows

L'exemple ne porte que sur l'opération "afficher le profil d'un utilisateur". Pour accéder à cette opération, nous allons préalablement construire une application Windows simple servant de point d'entrée. Il s'agira d'une simple fenêtre avec un bouton.



Fenêtre principale

Nous devons créer une classe qui implémente `IViewer`. Outre l'implémentation des méthodes "CallRequestUserCode" et "CallDisplayUserProfile", cette classe va contenir une méthode "Start" pour démarrer le traitement et une propriété "Manager" qui contiendra une référence vers le "Presenter". Cette propriété n'est accessible qu'en lecture et est assignée dans le constructeur.

i Le "Viewer" connaît le type de "Presenter" qu'il doit utiliser et passe sa référence comme paramètre au constructeur du "Presenter" qui pourra ainsi l'appeler.

La classe UIVIEWER

```
public class UIVIEWER : IVIEWER
{
    #region Properties

    private Presenter manager;
    public Presenter Manager
    {
        get { return manager; }
    }

    #endregion

    #region Constructeurs

    public UIVIEWER()
    {
        manager = new Presenter(this);
    }

    #endregion

    #region Public Methodes

    public void Start()
    {
        manager.Start();
    }

    #endregion
}
```

La classe UIViewer


```
#endregion


#region Interface Implementation

public void CallRequestUserCode()
{
    CodeRequestForm form = new CodeRequestForm(this);
    form.ShowDialog();
    manager.CodeInputEnded();
}

public void CallDisplayUserProfile()
{
    DisplayInfoForm form = new DisplayInfoForm();
    form.CodeLabel.Text = manager.Code;
    form.NomLabel.Text = manager.Nom;
    form.PrenomLabel.Text = manager.Prenom;
    form.ShowDialog();
}

#endregion
}
```


 Le Manager est défini publique de façon à pouvoir l'appeler directement depuis les différentes fenêtres. Si vous préférez le modèle strict, vous ne devez pas rendre ce champ publique. Vous devrez alors créer des méthodes relayant les fonctionnalités utilisées directement, comme c'est le cas pour "Start".

 Notez que dans la méthode "CallRequestUserCode" la dernière instruction est l'appel au "Presenter" pour lui indiquer la fin de l'opération d'encodage. A la place, il est possible d'utiliser des événements mais le code serait plus complexe et n'apporte rien à la compréhension du modèle.

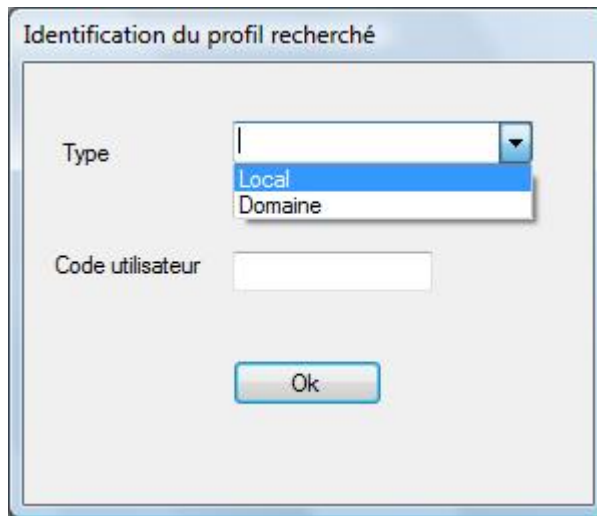
L'appel à ce "Viewer" se fait depuis le code associé au bouton de notre fenêtre principale.

Le lancement de la procédure gérée par MVP

```
private void startButton_Click(object sender, EventArgs e)
{
    UIViewer ui = new UIViewer();
    ui.Start();
}
```

 Créer une méthode "Start" publique plutôt que de l'inclure dans le constructeur permet d'effectuer le cas échéant certaines opérations sur le "Viewer" avant de démarrer l'affichage.

Le "Viewer" démarre le "Presenter" qui à son tour si vous vous en souvenez appelle "CallRequestUserCode". La première fenêtre est affichée.



Fenêtre d'encodage du code

Comme vous pouvez le constater, la liste des types est correctement remplie. Dans le constructeur de la fenêtre, il est fait appel au "Presenter". C'est donc bien le "Presenter" qui décide du contenu de la liste.

Remplissage de la liste

```

public CodeRequestForm(UIViewer caller)
{
    owner = caller;
    InitializeComponent();
    loadListTypes(owner.Manager.LoadListType());
}

private void loadListTypes(List<string> types)
{
    foreach (string type in types)
    {
        typeComboBox.Items.Add(type);
    }
}
    
```

La validation des données peut être réalisée lors de la fermeture de la fenêtre. La validation est également réalisée par le "Presenter" mais à la demande de l'interface graphique.

Validation des données

```

private void CodeRequestForm_FormClosing(object sender, FormClosingEventArgs e)
{
    owner.Manager.Code = userCodeTextBox.Text;
    owner.Manager.Type = typeComboBox.Text;
    e.Cancel = !owner.Manager.Validate();
}
    
```



Vous pouvez aussi considérer que non seulement le code de la validation doit être intégré dans le Presenter de façon à être unique mais que la validation doit non plus être demandée par l'interface graphique mais être contrôlée par le Presenter. De cette manière, il n'est pas possible de créer un interface graphique qui ne réalise pas la validation.

Code du Presenter incluant le déclenchement de la validation

```

public void CodeInputEnded()
    
```

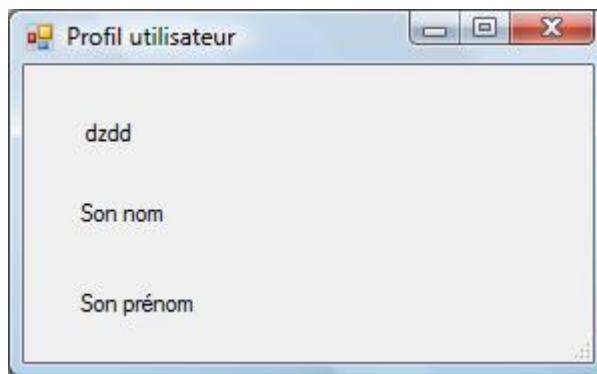
Code du Presenter incluant le déclenchement de la validation

```

{
    if (Validate())
    {
        loadProfile(code);
        viewer.CallDisplayUserProfile();
    }
    else
    {
        viewer.CallRequestUserCode("Données non valides");
    }
}

```

Comme nous l'avons vu précédemment, après la fermeture de cette fenêtre, le "Viewer" notifie le "Presenter" de la fin de l'encodage. Le "Presenter" déclenche l'affichage du profil.



Fenêtre d'affichage

III-B-2 - L'interface console

Sur base du même "Presenter", nous pouvons créer une application console. Il suffit d'utiliser un Viewer adapté.

Tout d'abord, il faut également démarrer le processus. Dans une application console, le plus évident est de l'inclure dans la procédure "Main".

La classe d'appel

```

class Program
{
    static void Main(string[] args)
    {
        UIViewer ui = new UIViewer();
        ui.Start();
    }
}

```

Ensuite nous avons besoin du "Viewer" lui-même.

LA classe UIViewer

```

public class UIViewer : IViewer
{
    private Presenter manager;
    public Presenter Manager
    {
        get { return manager; }
    }
}

```

LA classe UIVIEWER

```

    }

    public UIVIEWER()
    {
        manager = new Presenter(this);
    }

    public void Start()
    {
        manager.Start();
    }
}

```

Le traitement de demande des informations est séquentiel mais utilise les mêmes propriétés et méthodes que pour une application Windows.

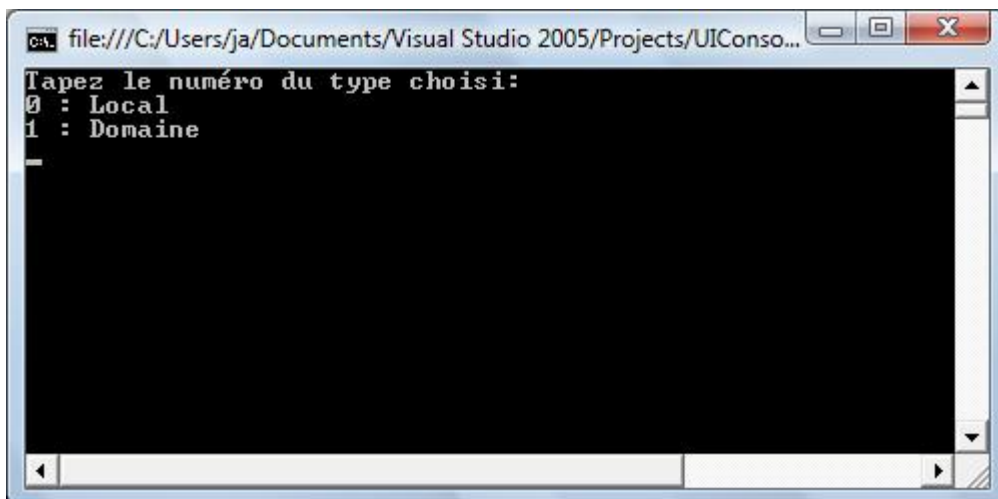
L'encodage du code utilisateur

```

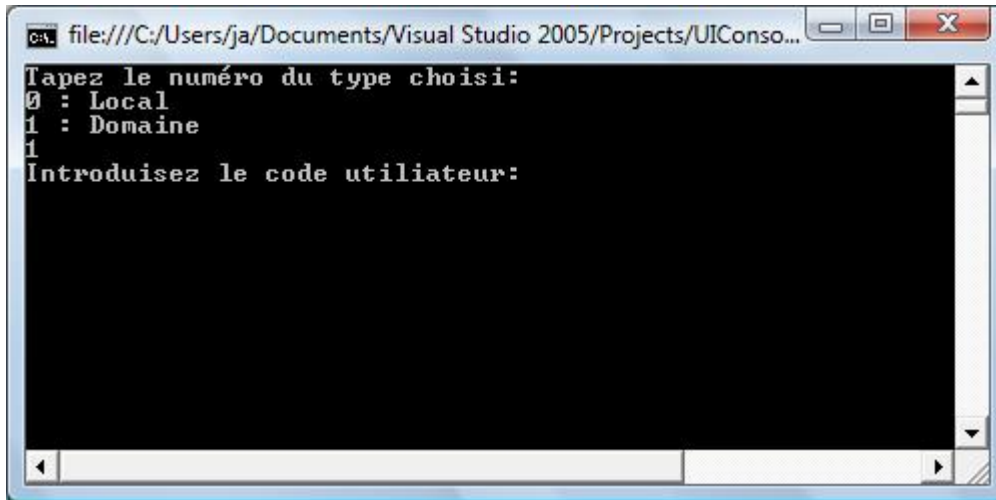
public void CallRequestUserCode()
{
    loadListTypes(manager.LoadListType());
    Console.WriteLine("Introduisez le code utilisateur:");
    manager.Code = Console.ReadLine();
    if (manager.Validate())
    {
        manager.CodeInputEnded();
    }
    else
    {
        Console.WriteLine("Données non valides");
        Console.ReadLine();
    }
}

private void loadListTypes(List<string> types)
{
    Console.WriteLine("Tapez le numéro du type choisi:");
    for (int i=0; i<types.Count; i++)
    {
        Console.WriteLine(string.Format("{0} : {1}", i.ToString(), types[i]));
    }
    manager.Type = types[Convert.ToInt32(Console.ReadLine())];
}

```



Encodage du type

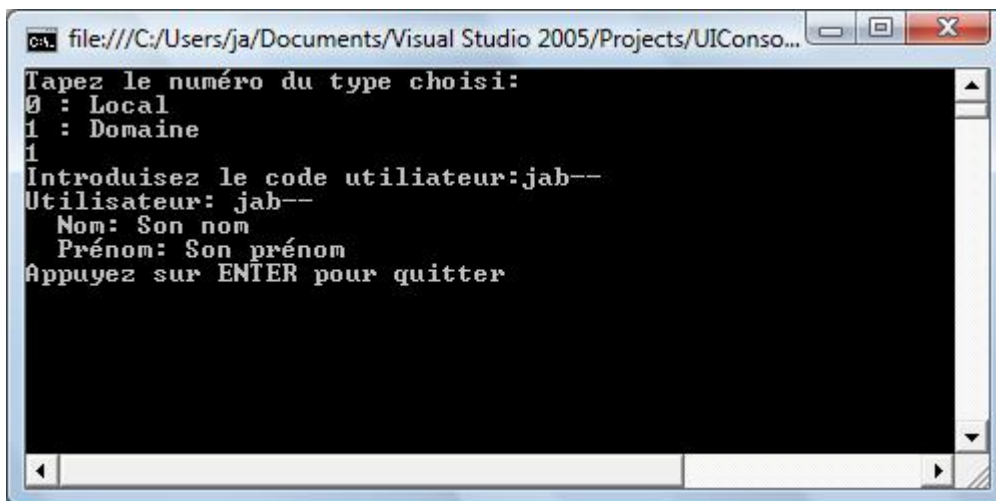


Encodage du code

L'affichage des informations

```

public void CallDisplayUserProfile()
{
    Console.WriteLine(string.Format("Utilisateur: {0}", manager.Code));
    Console.WriteLine(string.Format("  Nom: {0}", manager.Nom));
    Console.WriteLine(string.Format("  Prénom: {0}", manager.Prenom));
    Console.WriteLine("Appuyez sur ENTER pour quitter");
    Console.ReadLine();
}
    
```




Affichage du résultat

III-B-3 - L'interface Web

De la même manière, l'interface Web devra implémenter sa propre classe UIViewer. Dans l'exemple qui suit, le déclenchement du processus se fait via l'appel à la page "default.aspx". Cette page ne contient que l'appel au Presenter qui se fait dans la méthode "Page_Load". Vous pourriez aussi utiliser un bouton ou tout autre déclencheur.

Méthode Page_Load

```
protected void Page_Load(object sender, EventArgs e)
{
    UIViewier viewer = new UIViewier();
    HttpContext.Current.Session.Add("Viewer", viewer);
    viewer.Start();
}
```

 *S'agissant d'une application web, il faut assurer la persistance de notre objet "viewer". Le plus simple est d'utiliser une variable de session mais toute autre technique est également valable.*

La classe "UIViewier" est semblable aux précédentes. L'appel des pages adéquates se fait par redirection.

La classe UIViewier

```
public class UIViewier : IViewer
{
    private Presenter manager;
    public Presenter Manager
    {
        get { return manager; }
    }

    public UIViewier()
    {
        manager = new Presenter(this);
    }

    public void Start()
    {
        manager.Start();
    }

    public void CallRequestUserCode()
    {
        HttpContext.Current.Response.Redirect("CodeRequestPage.aspx");
    }

    public void CallDisplayUserProfile()
    {
        HttpContext.Current.Response.Redirect("DisplayProfilPage.aspx");
    }
}
```

Le code associé à la page d'introduction des données

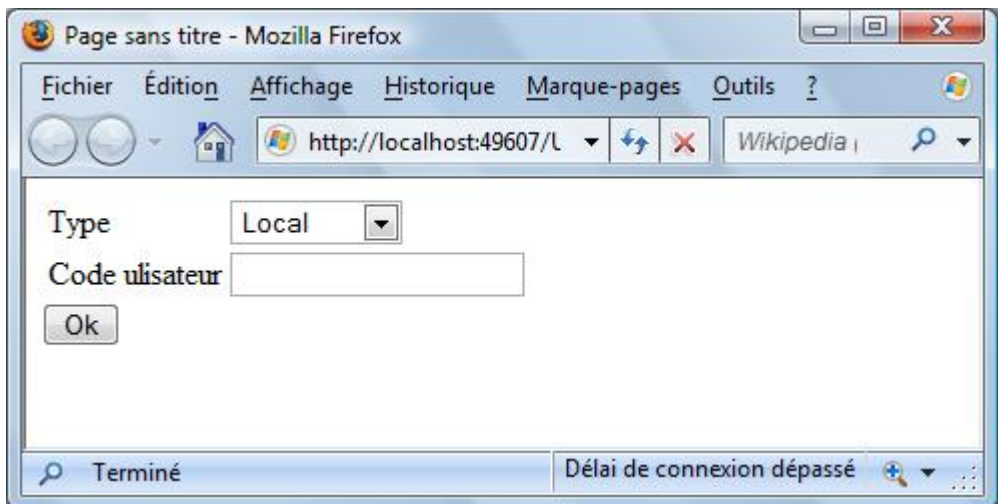
```
public partial class CodeRequestPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            UIViewier viewer = (UIViewier)HttpContext.Current.Session.Contents["Viewer"];
            loadListTypes(viewer.Manager.LoadListType());
        }
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        UIViewier viewer = (UIViewier)HttpContext.Current.Session.Contents["Viewer"];
        viewer.Manager.Code = UserCodeTextBox.Text;
        viewer.Manager.Type = typeDropDownList.SelectedValue;
    }
}
```

Le code associé à la page d'introduction des données

```
if (viewer.Manager.Validate())
{
    HttpContext.Current.Session.Add("Viewer", viewer);
    viewer.Manager.CodeInputEnded();
}

private void loadListTypes(List<string> types)
{
    foreach (string type in types)
    {
        typeDropDownList.Items.Add(type);
    }
}
```

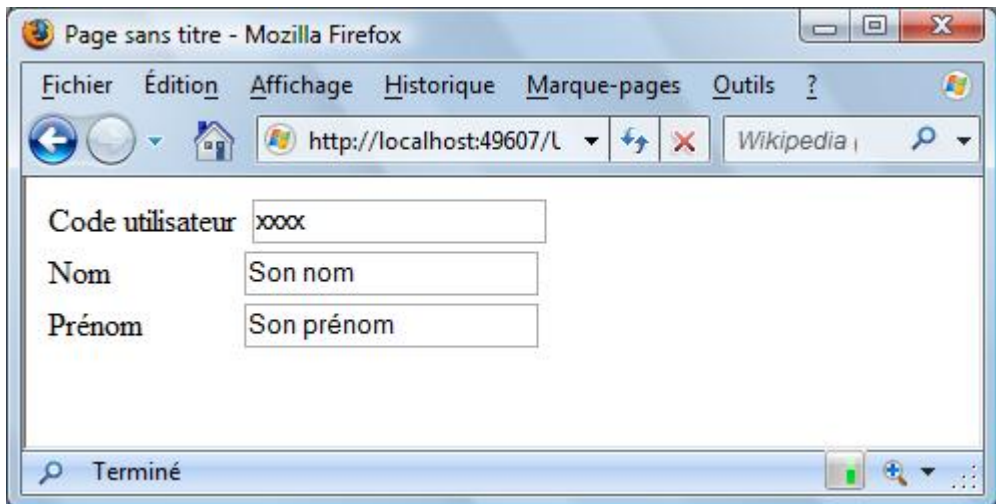


Fenêtre d'encodage

Le clic sur le bouton

```
protected void Button1_Click(object sender, EventArgs e)
{
    UIViewer viewer = (UIViewer)HttpContext.Current.Session.Contents["Viewer"];
    viewer.Manager.Code = UserCodeTextBox.Text;
    viewer.Manager.Type = typeDropDownList.SelectedValue;

    if (viewer.Manager.Validate())
    {
        HttpContext.Current.Session.Add("Viewer", viewer);
        viewer.Manager.CodeInputEnded();
    }
}
```




Fenêtre d'affichage du résultat


Le code associé à la page d'affichage du résultat

```
public partial class DisplayProfilPage : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        UIViewer viewer = (UIViewer)HttpContext.Current.Session.Contents["Viewer"];
        CodeTextBox.Text = viewer.Manager.Code;
        NomTextBox.Text = viewer.Manager.Nom;
        PrenomTextBox.Text = viewer.Manager.Prenom;
    }
}
```

IV - Placer le Presenter derrière un service web.

Maintenant que nous voici armé d'une couche unique qui prend en charge une bonne part du travail de nos différents interfaces utilisateurs, il serait logique que non seulement ce code soit unique mais en plus qu'il soit déployé à un seul endroit. Pour une question de portabilité, un web service semble s'imposer.

Le gros problème avec un web service, c'est qu'en l'absence de l'implémentation de la norme " Notification", ils ne sont pas conçu pour transmettre des messages vers le client. Le code que vous allez trouver dans ce chapitre est la preuve qu'il est possible de contourner ce problème mais le prix à payer est élevé surtout au niveau de la clarté du code.

 *L'exemple donné ici a été construit sommairement dans le but de démontrer la faisabilité. Il ne s'agit pas d'un modèle clé en main et si vous optez pour cette voie, il vous faudra travailler à l'amélioration du système.*

Comme le web service ne peut pas appeler directement les méthodes du client, c'est le client qui doit réaliser l'appel de ce que le web service lui demande. Il nous faut mettre en place un système pour envoyer au client les informations nécessaires, soit le nom de la méthode à exécuter ainsi que ces paramètres. Ces informations doivent être sérialisable. L'idée c'est que chaque méthode du web service qui a pour vocation de piloter l'interface renvoie une réponse standard. Il s'agit de la classe Response.

La classe Response

```
[Serializable]
public class Response
{
    public Response()
    {
    }

    private string methodeName;
    public string MethodeName
    {
        get { return methodeName; }
        set { methodeName = value; }
    }

    private object[] parameters;

    public object[] Parameters
    {
        get { return parameters; }
        set { parameters = value; }
    }
}
```

Maintenant, il nous reste à mettre en place un mécanisme côté client pour lui permettre d'exécuter ce qui lui est demandé. Comme le message reçu est de type textuel, seuls les mécanismes de réflexion vont nous permettre d'atteindre notre but.

La classe ReflexionServices

```
public abstract class ReflexionServices
{
    public static void Execute(object instance, Response methode)
    {
        MethodInfo meth = instance.GetType().GetMethod(methode.MethodeName);
        if (meth == null)
        {
        }
    }
}
```

La classe ReflexionServices

```

    {
        throw new ApplicationException(string.Format("La méthode {0} n'existe pas pour le type
d'objet {1}"
                                                    , methode.MethodeName
                                                    , instance.GetType().Name));
    }
    meth.Invoke(instance, methode.Parameters);
}
}

```

Maintenant, comme le Presenter est instancié et utilisé par le web service, il faut le persister sinon il sera perdu entre chaque appel. Une solution consiste à le persister sur le client ce qui facilitera le chargement des données. Vous pouvez également recréer le Presenter à chaque appel et utiliser une entité légère pour contenir les données. Dans ce cas, c'est elle qui doit être persistée. Pour permettre l'envoi au client, le Presenter où l'entité légère est ajouté à la classe Response.

La classe Response finale

```

[Serializable]
public class Response
{
    private Response() { }

    public Response(Presenter activeSession)
    {
        session = activeSession;
    }

    private Presenter session;


    public Presenter Session
    {
        get { return session; }
        set { session = value; }
    }

    private string methodeName;
    public string MethodeName
    {
        get { return methodeName; }
        set { methodeName = value; }
    }

    private object[] parameters;

    public object[] Parameters
    {
        get { return parameters; }
        set { parameters = value; }
    }
}

```

 La classe gagne une nouvelle propriété de type Presenter. Cette propriété est chargée dans le constructeur. Le constructeur par défaut devient privé. Il est nécessaire pour que la classe soit sérialisable.

Nous pouvons passer à l'écriture du web service.

Le web service WSPresenter

```

[WebService(Namespace = "http://tempuri.org/")]

```

Le web service WSPresenter

```
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class WSPresenter : System.Web.Services.WebService
{
    public WSPresenter()
    {
    }

    [WebMethod]
    public Response Start()
    {
        Presenter process = new Presenter();
        Response action = process.Start();
        return action;
    }

    [WebMethod]
    public bool Validate(object process)
    {
        return ((Presenter)process).Validate();
    }

    [WebMethod]
    public List<string> LoadListType(object process)
    {
        return ((Presenter)process).LoadListType();
    }

    [WebMethod]
    public Response CodeInputEnded(object process)
    {
        Response action = ((Presenter)process).CodeInputEnded();
        return action;
    }
}
```

Naturellement, les méthodes du web service ne sont que des relais pour notre Presenter. Il doit être adapté pour retourner une réponse correctement formatée.

La classe Presenter

```
public Response Start()
{
    Response result = new Response(this);
    result.MethodName = "CallRequestUserCode";
    return result;
}

public Response CodeInputEnded()
{
    loadProfile(code);
    Response result = new Response(this);
    result.MethodName = "CallDisplayUserProfile";
    return result;
}
```

A ce stade, il ne nous reste que le Viewer à adapter. Nous nous contenterons de modifier l'exemple de l'application console.

La classe UIViewer

```
public class UIViewer : IViewer
{
    #region Properties
```

La classe UIViewer

```
private Presenter session ;
private WSPresenter manager;
public WSPresenter Manager
{
    get { return manager; }
}
#endregion

#region Constructeur
public UIViewer()
{
    manager = new WSPresenter();
}
#endregion

#region Public Methodes
public void Start()
{
    Response reponse = manager.Start();
    session = reponse.Session;
    ReflexionServices.Execute(this, reponse);
}

public void CallRequestUserCode()
{
    loadListTypes(manager.LoadListType(session), session);
    Console.WriteLine("Introduisez le code utilisateur:");
    session.Code = Console.ReadLine();
    if (manager.Validate(session))
    {
        Response reponse = manager.CodeInputEnded(session);
        session = reponse.Session;
        ReflexionServices.Execute(this, reponse);
    }
    else
    {
        Console.WriteLine("Données non valides");
        Console.ReadLine();
    }
}

public void CallDisplayUserProfile()
{
    Console.WriteLine(string.Format("Utilisateur: {0}", session.Code));
    Console.WriteLine(string.Format(" Nom: {0}", session.Nom));
    Console.WriteLine(string.Format(" Prénom: {0}", session.Prenom));
    Console.WriteLine("Appuyez sur ENTER pour quitter");
    Console.ReadLine();
}
#endregion

#region Private Methodes
private void loadListTypes(string[] types, Presenter session)
{
    Console.WriteLine("Tapez le numéro du type choisi:");
    for (int i=0; i<types.Length; i++)
    {
        Console.WriteLine(string.Format("{0} : {1}", i.ToString(), types[i]));
    }
    session.TypeUser = types[Convert.ToInt32(Console.ReadLine())];
}
#endregion
}
```

Notez la présence du champ session. Il est nécessaire de le récupérer et de le stocker à chaque fois que vous recevez une réponse de type Response depuis le web service. Le champ session n'est en fait que la persistance du

Presenter. A mon sens l'utilisation d'une classe "entité" pour contenir et transporter les données serait préférable. Dans ce cas le Presenter ne posséderait qu'une propriété de type "entité" et c'est cette entité qui serait transmise par le web service et persistée sur le client. Dans ce tutoriel, je voulais garder le plus de code commun avec le premier exemple pour permettre une comparaison.

La réalisation d'un Presenter derrière un web service est possible malgré l'absence de l'implémentation de la notification. Malheureusement, le code perd beaucoup en lisibilité et en robustesse.

V - Conclusion

L'utilisation de ce pattern peut offrir beaucoup de possibilité et doit en tout cas être envisagé lorsque vous êtes amené à développer pour différentes plate formes. Il offre également plus de possibilités en terme de tests unitaires qu'un développement traditionnel car vos tests pourront également porter sur un niveau supérieur. Du côté négatif, il faut considérer le code plus complexe à mettre en oeuvre.

En ce qui concerne l'utilisation d'un web service, il s'agit plus de R&D que de présentation d'un pattern utilisable.

VI - Référence

 **Design Patterns: Model View Presenter.** Jean-Paul Boodhoo MSDN Magazine

VII - Sources

Source [Solution pour la fenêtre astuce](#)

Source [Solution sans web service](#)

Source [Solution avec web service](#)

