

# Aides au design, les Smart Tags

*Designer Action List.*

[Sébastien FERRAND](#) (MVP C#), dimanche 1<sup>er</sup> janvier 2006

## Droit de diffusion:

L'ensemble ou partie de ce document ainsi que le code à disposition, ne peut être diffusé sur d'autres sites Web sans l'autorisation préalable de son créateur.

## 1. Introduction.

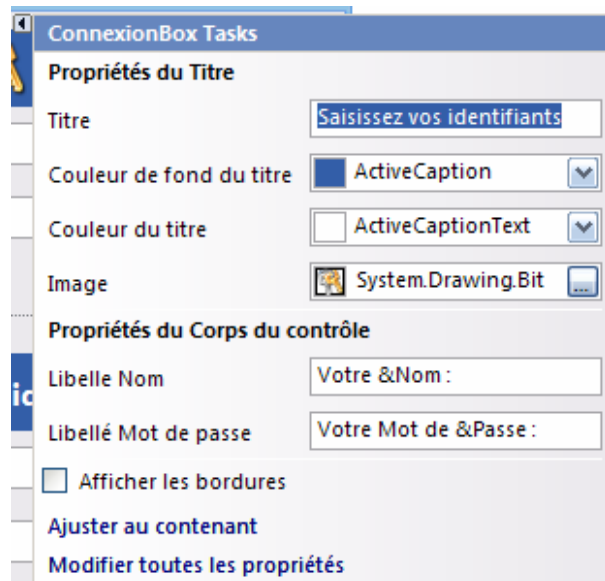
Nous sommes tour à tour développeur puis utilisateur de contrôles personnalisés. Dans ces rôles nous utilisons depuis des années le `PropertyGrid` pour renseigner nos propriétés, créer des *Handlers* pour nos événements... Dans la dernière mouture du *Framework .net*, Microsoft offre aux développeurs une nouvelle façon de travailler grâce aux *Smart Tags*.

Nous allons donc voir ensemble dans cet article comment fournir aux développeurs qui utiliseront vos contrôles cet outil simple à mettre en œuvre.

## 2. Présentation des Smart Tags.

Les Smart Tags sont apparût avec Microsoft Office System 2003 et proposent aux utilisateurs des outils contextuels personnalisables. Ce principe a été repris dans le *Framework .net 2.0* sous le nom de *Designer Action List* et se présente sous la forme d'un panneau qui s'affiche sur la droite des contrôles qui en sont pourvus (voir ci-contre).

Les concepteurs des contrôles sont libres d'y faire apparaître les informations qu'ils souhaitent.



### 3. Principes de base.

---

Lorsque vous créez un nouveau contrôle utilisateur, le *Framework* ne vous crée aucune *Action List*, c'est à vous qu'incombe cette tâche, c'est en fait à vous de choisir quelles seront les actions qui seront accessibles facilement aux développeurs qui utiliseront vos contrôles.

Une fois que vous aurez déterminé cette liste d'actions, vous devrez les regrouper au sein d'une classe dérivant de `DesignerActionList`, dont nous verrons l'implémentation plus tard, que nous relierons à notre contrôle grâce à un designer personnalisé.

La technique semble lourde mais pourtant, vous verrez qu'avec un peu de pratique c'est plutôt rapide à mettre en œuvre, et les développeurs qui utiliseront vos contrôles vous en remercieront.

Nous allons donc à présent utiliser le contrôle `ConnexionBox` que vous trouverez dans le code associé à cet article. Ce contrôle présente une simple interface de connexion comme celles que nous trouvons régulièrement dans les applications qui requièrent une authentification. L'article ne portant pas sur la création de contrôles, nous n'étudierons pas son code, qui est de toute façon très simple.

### 4. La classe `DesignActionList`.

---

Comme je l'ai exposé ci-dessus, nous devons créer une classe qui héritera de `DesignActionList` et qui fournira la liste des actions qui seront possible à partir de notre *Smart Tag*.

La classe `DesignActionList` nous propose de surcharger une propriété et une méthode.

La propriété `AutoShow` permet de décrire le comportement du panneau à sa création ; si `AutoShow` est égal à `true` alors le panneau s'ouvrira automatiquement à sa création, sinon, il faudra une action du développeur pour qu'il s'ouvre. Je vous conseille de ne pas surcharger cette propriété afin de garder le comportement défini dans les options de Visual Studio (Menu `Tools/Tools`, dans la section `Windows Forms Designer/General`, la propriété "`Automatically Open Smart Tags`").

La méthode `GetSortedActionItems` retourne une collection de `DesignerActionItem` triée correspondant aux actions de votre panneau, c'est ici que nous allons travailler à partir de maintenant.

Pour commencer, penchons-nous sur la classe `DesignerActionItem`, d'après la définition qu'on trouve dans la MSDN, cette classe sert de base pour les types qui représentent des éléments du *Smart Tag*. Le *Framework* nous propose les 3 types suivant :

1. `DesignerActionMethodItem` : qui affichera un lien dans le panneau afin d'exécuter une action ;
2. `DesignerActionPropertyItem` : qui permettra de modifier la valeur d'une propriété ;
3. `DesignerActionTextItem` : qui affichera un texte quelconque.

Pour notre exemple, nous allons permettre à notre développeur de saisir le titre de notre contrôle, modifier des couleurs et changer l'image (vous trouverez dans le code d'autres propriétés mais celles-ci n'apportent rien de plus à l'article).

Nous allons donc créer notre collection de `DesignActionItem` en surchargeant la méthode `GetSortedActionItems` comme ceci :

```
public override DesignerActionItemCollection GetSortedActionItems()
{
    DesignerActionItemCollection actionItems;
    // Création de l'objet
    actionItems = new DesignerActionItemCollection();
}
```

```

// on ajoute les actions possibles
actionItems.Add(
    new DesignerActionPropertyItem(
        "Titre",
        "Titre",
        "Titre",
        "Titre de la boite de connexion"));

actionItems.Add(
    new DesignerActionPropertyItem(
        "CouleurFondEntete",
        "Couleur de fond du titre",
        "Titre",
        "Indique la couleur à utiliser pour le fond du titre"));

actionItems.Add(
    new DesignerActionPropertyItem(
        "CouleurTextEntete",
        "Couleur du titre",
        "Titre",
        "Indique la couleur à utiliser pour le titre"));

actionItems.Add(
    new DesignerActionPropertyItem(
        "ImageEntete",
        "Image",
        "Titre",
        "Indique l'image à afficher en haut à droite"));

return actionItems;
}

```

Vous pouvez dès à présent remarquer que quelque soit le type de mes paramètres, j'utilise la même classe (`DesignerActionPropertyItem`) et les mêmes paramètres, le Framework utilise ensuite la classe `TypeConverter` de chaque propriété pour la convertir en `string`, attention tout de même, si celui-ci n'existe pas, la propriété sera affichée en lecture seule.

La surcharge du constructeur de la classe `DesignerActionPropertyItem` utilisée ci-dessus permet de définir les informations suivantes :

- Le nom de la propriété ;
- Le libellé qui sera affiché ;
- La catégorie de la propriété (comme dans le `PropertyGrid`) ;
- La description, qui s'affichera en `ToolTip` lorsque la souris s'attardera au dessus du libellé.

Une petite remarque cependant, le nom de la propriété ci-dessus ne correspond pas à celle de l'objet `ConnexionBox` mais à la classe `ConnexionBoxDesignerActionList`, vous devez donc (re)définir chacune d'entres-elles :

```

public string Titre
{
    get { return this.connexionBox.Titre; }
    set { this.connexionBox.Titre = value; }
}

public Image ImageEntete
{
    get { return this.connexionBox.ImageEntete; }
    set { this.connexionBox.ImageEntete = value; }
}

public Color CouleurFondEntete
{

```

```

    get { return this.connexionBox.CouleurFondEntete; }
    set { this.connexionBox.CouleurFondEntete = value; }
}

public Color CouleurTextEntete
{
    get { return this.connexionBox.CouleurTextEntete; }
    set { this.connexionBox.CouleurTextEntete = value; }
}

```

Le membre connexionBox ci-dessus est défini dans le constructeur de la manière suivante :

```

private ConnexionBox connexionBox;
public ConnexionBoxDesignerActionList(IComponent component) : base(component)
{
    connexionBox = component as ConnexionBox;
}

```

Nous allons aussi ajouter un lien pour que notre cher développeur puisse facilement ‘doker’ son contrôle dans le formulaire (ou panel...) qui le contient.

Pour faire ceci, rien de plus simple, ajouter l'extrait suivant dans la méthode GetSortedActionItems :

```

    actionItems.Add(
        new DesignerActionMethodItem(
            this,
            "ToggleDockStyle",
            "Ajuster au contenant",
            true));

```


Et ajoutez la méthode suivante à votre classe :

```

public void ToggleDockStyle()
{
    if (this.connexionBox.Dock == System.Windows.Forms.DockStyle.Fill)
    {
        this.connexionBox.Dock = System.Windows.Forms.DockStyle.None;
    }
    else
    {
        this.connexionBox.Dock = System.Windows.Forms.DockStyle.Fill;
    }
}

```

Ainsi, lorsque le développeur cliquera sur ce lien le contrôle s’ajustera (ou non) automatiquement au contrôle qui le contient. Nous pourrions remplacer le texte « Ajuster au contenant » par une méthode nous retournant le texte adapté au contexte, mais celui-ci ne serait pas modifié avant la réouverture du panneau.

 Nous avons vu ici les 2 principaux éléments du panneau, mais vous pouvez aussi ajouter du texte grâce aux classes DesignerActionTextItem et sa classe fille DesignerActionHeaderItem. Leur utilisation n’étant pas compliquée, je vous laisse regarder par vous-même leur utilisation dans le code source joint à cet article.

A présent, nous devons lier notre contrôle et cette classe car dans l’état, elle ne sert à rien, je vous invite donc à lire le paragraphe suivant.

## 5. Création d’un Designer personnalisé.

Je vous rassure tout de suite, nous n’allons pas recréer un designer de toutes pièces mais nous allons hériter de la classe ControlDesigner qui nous fournira tout ce dont nous aurons besoin.

Dans la version 2.0 du Framework .net, une nouvelle propriété a fait son apparition : `ActionLists`, grâce à elle, nous relierons notre contrôle à notre classe `ConnexionBoxDesignerActionList`.

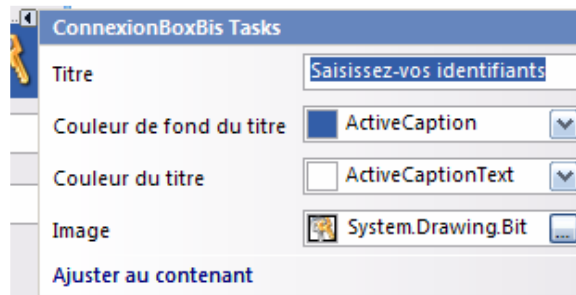
Ce qui nous arrange, c'est que cette propriété est de type `DesignerActionListCollection` et que notre classe hérite de `DesignerActionList`, nous allons pouvoir créer une collection et y ajouter une instance.

```
public override DesignerActionListCollection ActionLists
{
    get
    {
        if (actionListCollection == null) {
            actionListCollection = new DesignerActionListCollection();
            actionListCollection.Add(
                new ConnexionBoxDesignerActionListBis(this.Component));
        }
        return actionListCollection;
    }
}
```

Il ne nous reste plus qu'à dire à Visual Studio que nous voulons utiliser ce designer pour notre contrôle, cette opération ne requiert qu'une ligne au dessus de la déclaration de notre contrôle :

```
[Designer(typeof(ConnexionBoxDesigner))]
public partial class ConnexionBox : UserControl
```

Notre *Smart Tag* est enfin terminé et nous obtenons le résultat suivant :



## 6. Enrichissons un peu...

Cependant, dans ce que nous venons de voir, certaines choses me dérangent... étant un peu fainéant, j'ai tendance à vouloir coder le moins de lignes possible, c'est pourquoi, je trouve dommage de devoir écrire dans la méthode `GetSortedActionListItems` de la classe `ConnexionBoxDesignerActionList` le détail de chacune des propriétés que nous voulons ajouter à notre panneau. Pourquoi ? Et bien, en général, ces informations (nom, catégorie, description), nous les avons déjà renseignées dans notre code grâce aux attributs `DisplayName`, `Category` et `Description`.

Par exemple, nous écrirons notre propriété `Titre` de la manière suivante :


```
[DisplayName("Titre")]
[Category("Titre")]
[Description("Titre de la boîte de connexion")]
public string Titre {
    get { return lbTitre.Text; }
    set { lbTitre.Text = value; }
}
```

Nous allons donc réécrire cette méthode afin d'aller récupérer ces informations par réflexion sur l'objet.

Pour commencer, la technique qui permet de récupérer chaque attribut étant la même, nous n'allons pas l'écrire 3 fois, mais factoriser notre code et écrire une méthode spécialisée :

```
private T GetPropertyAttribute<T>(object o, string propertyName) where T : Attribute
{
    PropertyInfo p = o.GetType().GetProperty(propertyName);
    try
    {
        T attribute = (T)p.GetCustomAttributes(typeof(T), false)[0];
        return attribute;
    }
    catch {
        return null;
    }
}
```

Cette méthode retourne donc l'attribut de type *T* de la propriété *propertyName* de l'objet *o*.

 Pour en connaître un peu plus sur les *Generics*, vous pouvez vous référer à cette page de la documentation MSDN :

<http://msdn2.microsoft.com/en-us/library/512acb7t.aspx>.

Ensuite pour chaque attribut, on crée la méthode qui retournera la bonne valeur ou, le cas échéant, une valeur par défaut :

```
private string GetPropertyDisplayName(object o, string propertyName)
{
    DisplayNameAttribute att =
        GetPropertyAttribute<DisplayNameAttribute>(o, propertyName);
    if (att == null) return propertyName;
    return att.DisplayName;
}
```

```
private string GetPropertyDescription(object o, string propertyName)
{
    DescriptionAttribute att =
        GetPropertyAttribute<DescriptionAttribute>(o, propertyName);
    if (att == null) return propertyName;
    return att.Description;
}
```

```
private string GetPropertyCategory(object o, string propertyName) {
    CategoryAttribute att =
        GetPropertyAttribute<CategoryAttribute>(o, propertyName);
    if (att == null) return CategoryAttribute.Default.ToString();
    return att.Category;
}
```

Pour terminer, voici une méthode qui créera un objet *DesignerActionPropertyItem* en fonction du nom d'une propriété.

```
private DesignerActionPropertyItem
CreateDesignerActionPropertyItemFromPropertyName(string propertyName) {
    return new DesignerActionPropertyItem(propertyName,
        GetPropertyDisplayName(this.connexionBox, propertyName),
        GetPropertyCategory(this.connexionBox, propertyName),
        GetPropertyDescription(this.connexionBox, propertyName));
}
```

Comme vous pouvez le constater, nous appelons les méthodes que nous avons créées précédemment.

Cette fois, nous pouvons réécrire la méthode *GetSortedActionListItems* ainsi :

```
public override DesignerActionItemCollection GetSortedActionItems()
{
    DesignerActionItemCollection actionItems;
```

```
// Création de l'objet
actionItems = new DesignerActionItemCollection();

// on ajoute les actions possibles
actionItems.Add(
    CreateDesignerActionPropertyItemFromPropertyName("Titre"));
actionItems.Add(
    CreateDesignerActionPropertyItemFromPropertyName("CouleurFondEntete"));
actionItems.Add(
    CreateDesignerActionPropertyItemFromPropertyName("CouleurTextEntete"));
actionItems.Add(
    CreateDesignerActionPropertyItemFromPropertyName("ImageEntete"));

actionItems.Add(
    CreateDesignerActionPropertyItemFromPropertyName("LibelleNom"));
actionItems.Add(
    CreateDesignerActionPropertyItemFromPropertyName("LibelleMotDePasse"));

actionItems.Add(
    new DesignerActionMethodItem(
        this, "ToggleDockStyle", ToggleStyleText(), true));
return actionItems;
}
```

Ce qui est pour moi beaucoup plus propre.

Bien sûr pour que ça marche, il faut que votre classe soit correctement documentée avec les attributs adéquats.

## 7. Conclusion.

---

Cet article expose une nouvelles fonctionnalité du Framework .net 2.0 et de Visual Studio 2005, les *Smart Tags* ne sont bien sûr pas obligatoires pour vos composants... par contre, ils s'inscrivent dans une logique de professionnalisation de vos outils et des vos développements.

En bref, si vous souhaitez vendre ou distribuer vos contrôles, je ne saurais vous conseiller d'y adjoindre des *Smart Tags*.