

Le PropertyGrid

- Seconde Partie -

Sébastien FERRAND [MVP C#], 10 décembre 2005

Droit de diffusion :

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

1. Introduction.

Nous avons vu dans l'article précédent qu'il est possible de personnaliser et de faciliter l'attribution de valeurs aux propriétés d'un objet avec le `PropertyGrid`. Dans cette suite, nous allons revenir rapidement sur un point qui aurait pu être traité précédemment puis nous verrons comment personnaliser le nom des propriétés en .net 1.1.

Avant tout, vous trouverez le code source associé à cet article à l'adresse suivante : <http://www.csharpfr.com/code.aspx?ID=34356>.

2. UITypeEditor... la suite.

Nous avons qu'il est possible de définir des éditeurs personnalisés facilitant l'utilisation de certaines propriétés. Nous allons voir ici qu'il est aussi possible d'ajouter une icône à la gauche de notre valeur, celle-ci va permettre de la visualiser d'un coup d'œil.

Pour cela, nous devons surcharger 2 méthodes de plus lorsque nous créons notre éditeur personnalisé : `GetPaintValueSupported` et `PaintValue`.

La méthode `GetPaintValueSupported` permet d'avertir que nous voulons afficher une icône, et la méthode `PaintValue` la dessinera.

Nous allons donc reprendre l'un des éditeurs que nous avons construit dans l'article précédant : `SliderEditor`, pour y afficher une icône qui affichera une nuance de gris correspondant à la valeur que nous aurons sélectionnée.

```
public override
    bool GetPaintValueSupported(ITypeDescriptorContext context)
    {
        return true;
    }
```

```
public override
    void PaintValue(PaintValueEventArgs e)
    {
        if (e.Context == null) {
            base.PaintValue(e);
            return;
        }
        object obj = e.Context.Instance;
        int value = (int)e.Context.PropertyDescriptor.GetValue(obj);
        int valueAsByte = Math.Min(((value << 8) / 100), 255);

        Color color = Color.FromArgb(valueAsByte, valueAsByte,
            valueAsByte);
        e.Graphics.FillRectangle(new SolidBrush(color), e.Bounds);
    }
```

```
|| }
```

Ce code est relativement simple, il récupère la valeur de la propriété et à partir de celle-ci crée une nouvelle couleur dont chacune des composantes est comprise entre 0 et 255.

3. Désignons nos propriétés.

Cette fonctionnalité existe déjà dans le Framework .net 2.0 et permet de personnaliser l'affichage du nom d'une propriété dans le PropertyGrid. Il suffit simplement d'ajouter l'attribut `[DisplayName("nom personnalisé")]` à notre membre.

Dans cette partie, nous allons découvrir pas-à-pas comment procéder pour en faire de même avec le Framework .net 1.1.

a. Création d'un nouvel attribut.

Pour commencer, nous allons créer un nouvel attribut qui nous permettra de définir le nom que nous voulons voir afficher. Ce nom apparaîtra à la place du nom de la propriété dans le contrôle. Pour cela, nous dérivons la classe `System.Attribute` comme ci-dessous :

```
public class CustomDisplayNameAttribute : Attribute
{
    string displayName = string.Empty;
    public CustomDisplayNameAttribute(string displayName) {
        this.displayName = displayName;
    }

    public string CustomDisplayName {
        get {return this.displayName;}
        set {this.displayName = value;}
    }
}
```

Nous pourrions ainsi ajouter à nos propriétés l'attribut `[CustomDisplayName("le nom que je veux voir afficher")]`.

Cependant, l'ajout de cet attribut ne modifie en rien l'affichage de votre classe dans le PropertyGrid, ce dernier ignorant complètement sa présence et son utilisation. Pour gérer notre affichage personnalisé, nous devons à présent dériver 1 autre classe (`System.ComponentModel.PropertyDescriptor`) et implémenter une interface (`System.ComponentModel.ICustomTypeDescriptor`).

b. Dérivons la classe *PropertyDescriptor*.

Le rôle de cette nouvelle classe sera de surcharger la propriété `DisplayName` de la classe mère afin de récupérer l'attribut que nous venons de créer et l'afficher à la place du nom par défaut. Nous créons donc une classe nommée `CustomPropertyDescriptor` dérivant de `PropertyDescriptor` et surchargeons la propriété `DisplayName`.

```
public override string DisplayName {
    get {
        string displayName = basePd.DisplayName;
        // Recherche parmi les attributs de la propriété
        // "CustomDisplayNameAttribute", s'il est présent et
        // non vide, on l'affiche à la place du nom de la
        // propriété
    }
}
```

```

        foreach(Attribute attr in basePd.Attributes) {
            if (attr is CustomDisplayNameAttribute) {
                displayName =
                ((CustomDisplayNameAttribute)attr).CustomDisplayName;
                if (string.IsNullOrEmpty(displayName)) {
                    displayName = basePd.DisplayName;
                }
                break;
            }
        }
        return displayName;
    }
}

```

Dans cette surcharge, nous recherchons l'attribut que nous avons créé précédemment dans la collection des attributs de la classe de base (*basePd* étant renseigné dans le constructeur avec une instance de la classe de `PropertyDescriptor`). Si nous le trouvons et qu'il est non *null* et non vide nous en renvoyons la valeur. Dans le cas contraire, la valeur par défaut (c'est-à-dire le nom de la propriété) sera retournée.

Vous remarquerez que tous les autres membres de la classe ont été surchargés, cela s'explique par le fait que tous ces membres sont marqués *virtual* dans la classe `PropertyDescriptor` et donc doivent être surchargés. Cependant, il suffit de rappeler le membre de base dans notre nouvelle implémentation pour que notre classe fonctionne correctement.

Bien que nous travaillons semble satisfaisant, il nous manque encore une chose à faire : nous devons 'dire' au `PropertyGrid` d'utiliser notre nouveau *descriptor*. C'est là qu'intervient l'interface `ICustomPropertyDescriptor` que nous allons implémenter maintenant.

c. Implémentons l'interface `ICustomPropertyDescriptor`.

Cette interface permet de définir un descripteur d'objet, celui-ci sera utilisé par le `PropertyGrid` pour afficher les informations de l'objet.

Nous continuons donc en créant une classe `CustomDescriptor` implémentant cette interface et concentrons-nous sur ce que nous intéressons : ce que nous voulons, c'est permettre l'utilisation de notre descripteur avec le `PropertyGrid`... et bien, pour cela observons les membres que nous devons implémenter :

- `GetConverter` : pas intéressant
- `GetEvents` : pas intéressant
- `GetComponentName` : pas intéressant
- `GetPropertyOwner` : pas intéressant
- `GetAttributes` : pourrait-être intéressant... mais en fait retourne la liste des attributs de la classe... et non de la propriété, donc pas intéressant
- `GetProperties` : là, nous touchons au but... cette méthode nous retourne une collection de `PropertyDescriptor`, justement ce que nous avons dévirié tout à l'heure !
- `GetEditor` : pas intéressant
- `GetDefaultProperty` : pas intéressant
- `GetDefaultEvent` : pas intéressant
- `GetClassName` : pas intéressant

Tous les points que nous avons marqués « pas intéressant » doivent quand même être implémentés, c'est le principe même d'une interface, c'est pourquoi nous utiliserons la classe `TypeDescrip-`

tor qui propose justement des membres *static* portant les mêmes noms que les nôtres. C'est ce que vous pouvez voir dans le code fourni en exemple. (Je ne m'attarde pas dessus car cela ne correspond pas au sujet de cet article)

Il nous reste plus qu'une seule chose à faire : implémenter la méthode `GetProperties()` !

```
public PropertyDescriptorCollection GetProperties()
{
    if ( pdc == null )
    {
        PropertyDescriptorCollection defaultProp =
            TypeDescriptor.GetProperties(this, true);

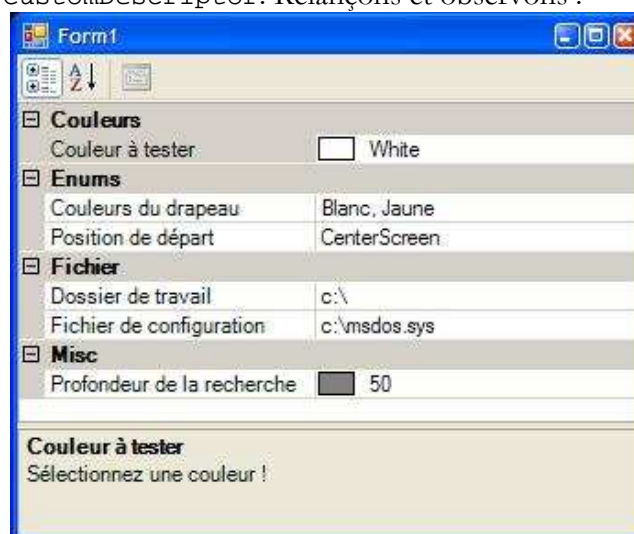
        pdc = new PropertyDescriptorCollection(null);

        foreach( PropertyDescriptor pd in defaultProp )
        {
            pdc.Add( new CustomPropertyDescriptor(pd) );
        }
    }
    return pdc;
}
```

Dans l'exemple ci-dessus, la variable `pdc` est un membre privé de la classe `CustomDescriptor`.

Explications : comme nous avons pu le voir plus précédemment, la méthode `GetProperties` est chargée de retourner une collection de `PropertyDescriptor`, celle-ci permettra à notre `PropertyGrid` de faire son travail et d'afficher les informations nécessaires. L'idée ici est de remplacer cette collection par une autre contenant notre descripteur de propriétés, celui-ci héritant de la classe `PropertyDescriptor`, ce qui veut dire qu'il peut être ajouté de façon transparente à cette collection.

Voilà, le tour est joué, notre travail est presque terminé, à présent il faut hériter notre classe d'exemple de la classe `CustomDescriptor`. Relançons et observons :



Nous constatons avec plaisir que les efforts que nous avons fait ne sont pas restés vains et que toutes les propriétés ont bien été renommées.

4. Conclusion.

Nous avons vu dans cet article qu'il est simple ajouté une icône représentant une valeur dans un `PropertyGrid` en surchargeant 2 membres dans la class `UITypeEditor`, puis nous avons démontré qu'il est aussi possible de modifier le nom des propriétés à l'affichage. Nous pourrions presque écrire un livre tellement il reste de choses à découvrir sur ce contrôle. Il est ainsi possible de localiser les noms des propriétés, les descriptions, etc...

J'espère que ces 2 articles vous auront convaincu que ce contrôle vaut la peine d'être utilisé dans vos applications et qu'il est 'facilement' personnalisable.