



Concevoir des applications pour Windows XP - 1ère Partie

(Avec code source à télécharger)

Mis à jour le 20/04/2005
Par Sébastien FERRAND

[Version PDF à télécharger](#)
[Code Source](#)

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.

Avant Propos :

S'il y a bien une chose que l'on critique très rarement, c'est l'interface de Windows, mieux même, elle est souvent prise comme modèle. Avec .NET, nous avons la possibilité de créer des interfaces ayant un rendu professionnel en écrivant peu de lignes, encore faut-il savoir quelles sont-elles.

Sommaire:

- [1. Généralités](#)
- [2. Les TabControls](#)
 - [2.1. 1ère solution](#)
 - [2.2. 2ème Solution](#)
 - [2.3. 3ème solution](#)
- [3. Performances](#)
- [4. Allons plus loin](#)

1. Généralités

1.1. 1ère solution

Pour que votre application s'intègre correctement à Windows XP, vous aurez besoin de lui préciser qu'elle en utilise les thèmes. Il existe pour cela au sein de l'assembly System.Windows.Form la classe Application et sa méthode EnableVisualStyle(). Vous devrez y faire appel avant l'ouverture du premier formulaire, et puisqu'elle concerne toute notre application, l'intégration dans la méthode Main est appropriée.

```
1:  /// <summary>
2:  /// Point d'entrée principal de l'application.
3:  /// </summary>
4:  [STAThread]
5:  static void Main() {
6:      Application.EnableVisualStyles();
7:      Application.DoEvents();
8:      Application.Run(new frmMain());
9:  }
10:
```

L'utilisation de la méthode Application.DoEvents() n'est pas obligatoire mais conseillé car il arrive parfois que l'application rende une erreur dès le démarrage.

Ces instructions demandent à votre application d'utiliser pour ses composants le thème d'XP, mais certains éléments resteront toutefois inchangés, ceux-ci dérivent des classes Button-Base, GroupBox et Label et possèdent tous une propriété FlatStyle que vous devrez mettre à System.

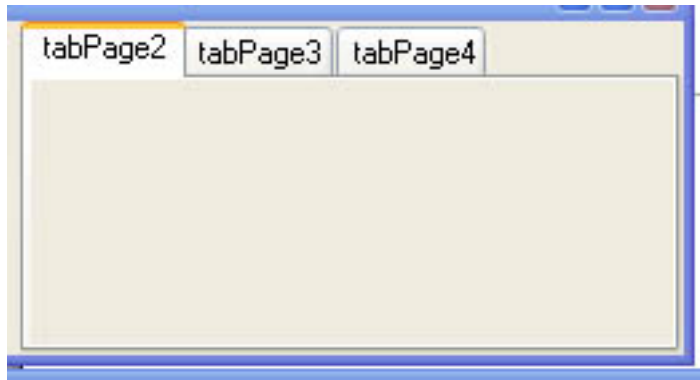
Vous êtes maintenant prêt à développer des applications qui se fondent dans Windows XP.

Remarque : L'ajout de l'appel à la méthode EnableVisualStyle() vous dispense de la création du fichier manifest correspondant à votre application puisque celui-ci est directement compilé en tant que ressource.

2. Les TabControls

Les objets TabControls permettent d'afficher plusieurs pages dans un espace réduit, ils permettent aussi de réunir beaucoup d'informations sans avoir besoin d'ouvrir un grand nombre de fenêtres. Malheureusement, en standard dans le framework .NET 1.1, j'ai l'impression que l'implémentation des styles ait été un peu bâclée, car comme vous pouvez le constater sur la capture le corps de la page est de la même couleur que le fond de la fenêtre, alors que normalement il devrait y apparaître un dégradé.

Nous allons voir comment il est possible, à l'aide de 3 méthodes différentes, de rendre ce contrôle plus en accord avec le reste de votre interface.



2.1. 1ère solution

La première méthode qui vient à l'esprit est d'utiliser les fonctions de base de GDI+ pour re-peindre le fond de notre contrôle. Mais pour cela, il faut savoir de quelles couleurs nous allons le repeindre, je vous laisse donc utiliser le logiciel de votre choix. Pour ma part, j'ai relevé les couleurs suivantes (en RGB) :

- 252, 252, 254
- 244, 243, 238

À partir de celles-ci, il sera possible de remplir notre fond correctement en interceptant l'évènement Paint des pages du contrôle. Le code suivant est un exemple de la manière dont vous pouvez vous y prendre :

```
1: private void tabPage1_Paint(object sender, Sys-
tem.Windows.Forms.PaintEventArgs e) {
2:     e.Graphics.FillRectangle(
3:         new LinearGradientBrush(
4:             new Rectangle(new Point(0,0), tabPage1.Size),
5:             Color.FromArgb(252, 252, 254),
6:             Color.FromArgb(244, 243, 238),
7:             LinearGradientMode.Vertical),
8:         e.ClipRectangle);
9: }
```

À l'aide de l'objet Graphics que nous passe l'évènement en paramètre nous allons remplir le rectangle ClipRectangle qui correspond à la zone à redessiner (cela évite de redessiner l'ensemble si seulement quelques pixels le nécessitent).

La méthode FillRectangle a besoin comme premier paramètre d'un objet dérivant de la classe Brush (pinceau), nous utiliserons ici LinearGradientBrush (pinceau dégradé linéaire).

À l'exécution, la différence entre le TabControl standard et celui que nous venons de repeindre est flagrante. À droite se trouve le rendu par défaut et à gauche le TabControl dont la page a été repeinte.



Nous venons de voir qu'en quelques lignes et quelques secondes, qu'il était possible de modifier rapidement l'apparence d'une TabPage mais cette solution ne me satisfait pas entièrement. D'accord, le rendu est similaire à ce qu'on trouve au sein de Windows XP... mais que se passe-t-il si demain l'utilisateur change le thème de son Windows ? Mes couleurs seront-elles encore correcte ?

2.2. 2ème Solution

En partant du principe que Windows doit aller chercher les informations quelque part, je suis parti à la recherche d'informations sur la gestion des thèmes et j'ai trouvé une API toute dé-dîée : UXTheme.dll. C'est très simple... tout ce qui touche à la personnalisation de Windows se trouve dans cette librairie.

Reprenant, l'idée précédente, j'ai recherché les couleurs qu'utilise Windows XP pour dessiner le fond des TabPages.

Nous allons donc commencer par déclarer les méthodes, les structures et les constantes qui nous servirons pour cette exemple :

```
1: [DllImport("uxtheme.dll", ExactSpelling=true,
 CharSet=CharSet.Unicode)]
2: public static extern IntPtr OpenThemeData(IntPtr hWnd, String
classList);
3:
4: [DllImport("uxtheme.dll", ExactSpelling=true)]
5: public extern static Int32 CloseThemeData(IntPtr hTheme);
6:
7: [DllImport("uxtheme", ExactSpelling=true)]
8: public extern static Int32 GetThemeColor(IntPtr hTheme,
9:     int iPartId, int iStateId, int iPropId, out COLORREF pColor);
10:
11: public struct COLORREF {
12:     public byte R;
13:     public byte G;
14:     public byte B;
15:
16:     public Color ToColor() {
17:         return Color.FromArgb((int)R, (int)G, (int)B);
18:     }
19:
```

```

20:     public override string ToString() {
21:         return string.Format("{0},{1},{2}",R,G,B);
22:     }
23: }
24:
25: public const int TMT_FILLCOLOR = 3821;
26: public const int TMT_EDGEFILLCOLOR = 3808;
27: public const int TABP_BODY = 10;

```

Les 3 méthodes permettent d'ouvrir un canal de communication entre votre application et la gestion de thèmes (OpenThemeData), de récupérer les couleurs des éléments (GetTheme-Color) et de refermer ce canal (CloseThemeData).

L'idée est simple : nous ouvrons le canal, nous récupérons les couleurs et nous le refermons.

Regardez le code suivant :

```

1: IntPtr hTheme;
2: Color color1;
3: Color color2;
4:
5: hTheme = OpenThemeData(this.Handle, "TAB");
6: COLORREF c;
7:
8: GetThemeColor(this.hTheme, TABP_BODY, 1, TMT_FILLCOLOR, out c);
9: color1 = c.ToColor();
10:
11: GetThemeColor(this.hTheme, TABP_BODY, 1, TMT_EDGEFILLCOLOR, out c);
12: color2 = c.ToColor();
13:
14: CloseThemeData(hTheme);

```

Nous utilisons la méthode OpenThemeData pour accéder aux propriétés du thème en lui précisant le handle (identifiant système) de la fenêtre courante et une liste de classes d'objets séparées par des points-virgules (ici, nous n'avons qu'une seule classe). Ensuite nous récupérons les 2 couleurs de TABP_BODY qui nous intéressent (le corps du contrôle) : TMT_FILLCOLOR et TMT_EDGEFILLCOLOR. CloseThemeData referme canal spécifié.

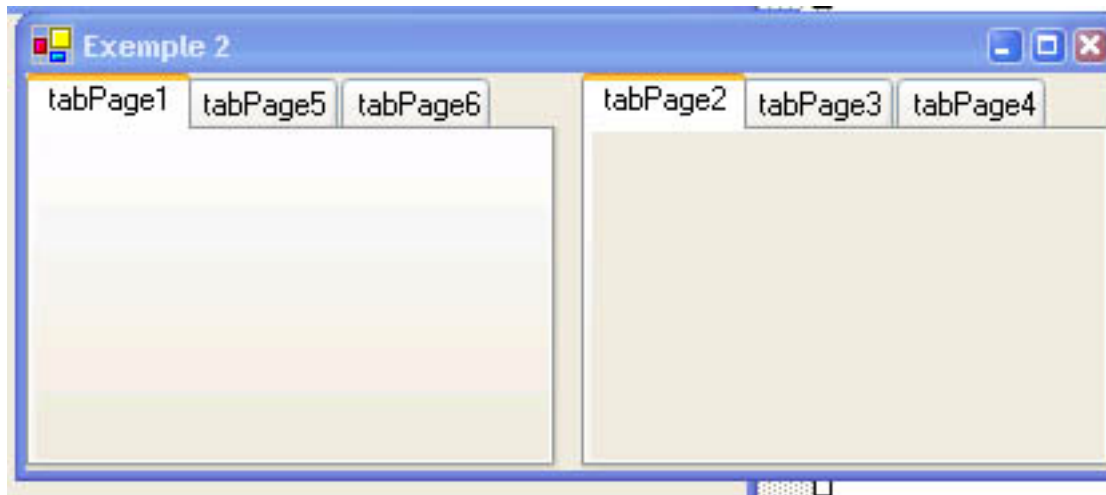
Après l'adaptation de l'évènement Paint du tabPage, nous obtenons le code suivant :

```

1: private void tabPage1_Paint(object sender, Sys-
tem.Windows.Forms.PaintEventArgs e) {
2:     e.Graphics.FillRectangle(
3:         new LinearGradientBrush(
4:             new Rectangle(new Point(0,0), tabPage1.Size),
5:             color1,
6:             color2,
7:             LinearGradientMode.Vertical),
8:         new Rectangle(new Point(0,0), tabPage1.Size));
9: }
10:

```

La capture suivante montre que le résultat ainsi obtenu est similaire (ou presque) à la capture précédente, les couleurs n'étant apparemment pas tout à fait juste.



Cet exemple pourrait donc satisfaire nos besoins en terme de qualité, et puis maintenant si vous changez l'apparence de Windows XP votre contrôle aura toujours le bon 'look'.

Pourtant, cette solution n'est pas la bonne... Nous utilisons jusqu'à présent le framework pour dessiner, plus précisément GDI+... mais Windows n'est pas basé dessus, Windows utilise ses propres APIs, pourquoi ne le ferions-nous pas ?

2.3. 3ème solution

Dans cette solution, comme pour la précédente, nous allons utiliser l'API UXTheme.dll, mais cette fois-ci, nous appellerons la méthode DrawThemeBackground qui se chargera elle-même de dessiner le fond de notre contrôle. En réalité, elle se charge de récupérer le bitmap qui est défini dans le thème et l'adapte au rectangle du contrôle. Cette API simplifie véritablement le code, il n'est plus question maintenant de dessiner un dégradé mais d'afficher un bitmap.

Nous pourrions donc simplifier le code de la manière suivante :

```

1: private void tabPage1_Paint(object sender,      Sys-
tem.Windows.Forms.PaintEventArgs e) {
2:     RECT rect1 = new RECT(0,0,tabPage1.Width, tabPage1.Height);
3:     RECT rect2 = new RECT(e.ClipRectangle);
4:     IntPtr hdc = e.Graphics.GetHdc();
5:     DrawThemeBackground( this.hTheme,
6:         hdc,
7:         TABP_BODY,
8:         1,
9:         ref rect1,
10:        ref rect2);
11:    e.Graphics.ReleaseHdc(hdc);
12: }

```

Et comme vous pouvez le constater sur la capture ci-contre, le résultat est parfait, notre TabControl ressemble exactement à celui du système.



Nous sommes dorénavant sûr que notre application s'affichera toujours de la bonne manière quelque soit le thème utilisé par l'utilisateur.

3. Performances

Parmi les questions qui m'ont été posées lorsque que j'ai fait relire cet article, on retrouve des inquiétudes sur la rapidité d'exécution : « Quelles sont les performances ? », « Quelle est la solution la plus rapide ? ». J'ai donc décidé de mettre en place une méthode de calcul qui me permette de révéler les différences entre chaque solution.

Mode opératoire : Pour connaître le temps d'exécution, j'ai donc disposé sur chaque formulaire un bouton qui me permet de repeindre le contrôle. Cette méthode à l'avantage de pouvoir recommencer le test autant fois qu'on le désire et de calculer une moyenne des résultats.

Voici donc les résultats que j'obtiens :

	Solution 1	Solution 2	Sol
Test 1	10 ms	10 ms	1
Test 2	10 ms	0 ms	1
Test 3	10ms	30 ms	3
Test 4	10 ms	10 ms	3
Test 5	10 ms	10 ms	1
Moyenne :	10 ms	12 ms	11

Interprétation :
 Nous pouvons remarquer que la solution 3 est 10 fois plus lente que les 2 précédentes, je suis aussi surpris que vous. Paradoxalement, cette solution affiche un bitmap est non un dégradé, il faut donc l'adapter au rectangle du contrôle. Malgré tout, elle reste ma solution privilégié car le rendu est celui qui est le plus professionnel. La solution qui permettra d'accélérer sera sans doute de déporter ce dessin dans un User-Control dont le style comprendra la valeur `ControlStyle.DoubleBuffer`. Nous y reviendrons dans le prochain article.

4. Allons plus loin

Cet article et ceux qui suivront traiteront des astuces qui permettent à une application d'utiliser à plein le potentiel de

Windows XP (et de Windows Server 2003) pour la personnalisation. Malheureusement, en entreprise le parc n'est pas forcément composé uniquement de poste de dernière génération. Vous trouverez encore dans la plupart des cas des ordinateurs tournant sous Windows 2000 Pro voire même Windows 98.

Ces OS ne supportant pas ces APIs vous devrez obligatoirement vérifier que ce dernier est compatible. Une petite recherche sur [Google](#) nous retourne ce blog : <http://addressof.com/blog/archive/2004/02/15/400.aspx>



L'ensemble ou partie de ce document ainsi que le code mis à disposition, ne peut être diffusé ailleurs sans autorisation préalable

elise.dupont@europe.com - www.dotnet-tech.com - 2004