Présentation de WPF

Les applications Windows sont des applications fenêtrées présentant des données d'une manière bien plus graphique et agréable que la fenêtre de ligne de commande. Pour développer ce type d'application, .NET propose depuis ses débuts la technologie **Windows Forms**. La version 3.0 du framework .NET, sortie en 2006, est arrivée avec une nouvelle technologie de présentation : **WPF** (*Windows Presentation Foundation*). Celle-ci a apporté plusieurs améliorations notables tant au niveau technique qu'à celui du développement :

- La définition de l'interface est effectuée à l'aide du langage de balisage **XAML** (*eXtensible Application Markup Language*) et non plus avec du code C#.
- Le **découplage** entre l'interface et le code métier peut être effectué plus facilement grâce à la notion de **binding**.
- L'**affichage** ne s'appuie plus sur la brique logicielle GDI mais sur **DirectX**, ce qui implique que certains calculs peuvent être déportés sur le **GPU**.
- Tous les composants WPF utilisent le **dessin vectoriel**.
- Au sein d'un composant, l'aspect graphique et l'aspect fonctionnel sont très peu liés, ce qui permet une modification graphique poussée des composants.

Malgré sa robustesse, la technologie Windows Forms est aujourd'hui fréquemment délaissée au profit de WPF. Les raisons principales de ce choix sont liées d'une à part à l'expérience utilisateur, puisque **WPF** permet de créer simplement des **applications fluides** et à l'**aspect moderne**, et d'autre part à des motifs purement techniques, car les possibilités de découplage offertes par WPF améliorent la **testabilité** et la **maintenabilité** des applications.

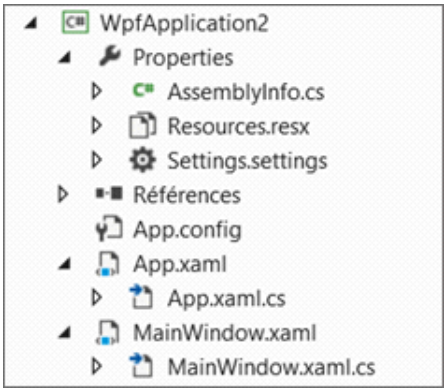
Microsoft mise aujourd'hui sur WPF et les technologies utilisant XAML. Une équipe de développeurs du géant du logiciel a d'ailleurs mentionné à la conférence BUILD 2014 que la technologie **Windows Forms** est aujourd'hui en **phase de maintenance**, ce qui signifie qu'elle n'aura plus de mises à jour au niveau des fonctionnalités, bien que les problèmes détectés soient corrigés.

1. Structure d'une application WPF

Au moment de sa création, un projet WPF est déjà composé de plusieurs fichiers :

- Les fichiers `App.xaml` et `App.xaml.cs` contiennent la déclaration et l'implémentation de la classe principale de l'application : c'est en effet elle qui contient le point d'entrée du programme. Le fichier `.xaml` contient généralement toutes les ressources nécessaires au bon fonctionnement de l'application, tandis que le fichier de code-behind associé contient les gestionnaires d'événements relatifs au cycle de vie de l'application ou les portions de code nécessaires à son démarrage.
- Les fichiers `MainWindow.xaml` et le fichier de code-behind `MainWindows.xaml.cs` contiennent respectivement la description de la fenêtre principale de l'application et le code C# qui lui est associé.
- Le fichier `app.config` contient les paramètres de configuration de l'application.
- `AssemblyInfo.cs` contient les métadonnées décrivant l'assembly résultant de la compilation du projet : titre, version, informations de copyright... Ces informations sont décrites sous la forme d'attributs.
- Le fichier `Resources.resx` contient généralement des chaînes de caractères utilisables en plusieurs endroits de l'application, mais il peut aussi encapsuler des ressources de type binaire.
- Le fichier `Settings.settings` permet quant à lui de stocker des paramètres liés à l'application ou à un utilisateur de l'application : tailles de polices, couleurs ou emplacement des barres d'outils par exemple.

Ces différents fichiers sont organisés de la manière suivante :



2. XAML

La technologie WPF utilise le langage XAML pour la **création d'interfaces** graphiques. Ce langage est un **dialecte XML** qui permet l'instanciation d'objets .NET de manière déclarative, c'est-à-dire par l'utilisation de balises, à la manière de HTML. Les différents objets sont imbriqués à partir d'un élément racine pour former un arbre logique.

La définition d'un bouton à l'aide de XAML ressemble donc au code suivant :

```
<Button Height="30" Width="120">Ceci est un bouton</Button>
```

La balise `<Button>` instancie un objet de type `Button`, tandis que les attributs XML valorisent les propriétés `Height` et `Width` du bouton. Comme tout élément XML, une balise XAML peut contenir d'autres balises ou une valeur textuelle.

Il est important de savoir que tout ce qui peut être codé en XAML peut être codé avec C#. L'équivalent impératif du code XAML ci-dessus est le suivant :

```
Button bouton = new Button();
bouton.Height = 30;
bouton.Width = 120;
bouton.Content = "Ceci est un bouton";
```

On voit ici apparaître une propriété `Content` pour laquelle aucune valeur n'était explicitement fournie en XAML. Cette propriété est en fait la **propriété par défaut** pour le type `Button`, ce qui signifie que toute valeur textuelle ou XAML placée à l'intérieur de la balise `Button` valorise la propriété `Content` de l'objet `Button`.

La propriété par défaut d'un type est définie en plaçant sur le type un attribut `ContentPropertyAttribute` spécifiant le nom de la propriété.

La **syntaxe élément-propriété** permet d'assigner un contenu XAML complexe à une propriété. Elle autorise l'utilisation d'une propriété en tant qu'élément XML imbriqué. Le nom de l'élément doit faire apparaître le nom du type sur lequel on agit, suivi par un point puis le nom de la propriété concernée. Suivant cette syntaxe, le code XAML de création d'un bouton vu précédemment peut être écrit de la manière suivante :

```
<Button Height="30" Width="120">
    <Button.Content>Ceci est un bouton</Button.Content>
```

```
</Button>
```

Tout le code C# n'est en revanche pas transposable en XAML car il est notamment impossible d'appeler une méthode directement de manière déclarative. Bien que XAML soit très puissant grâce aux nombreux objets utilisables, les scénarios avancés nécessitent donc très souvent l'utilisation de code .NET impératif.

Bien qu'il ait été initialement introduit avec WPF, le langage de description XAML est aujourd'hui largement utilisé pour le développement d'applications n'utilisant pas la technologie WPF (comme les applications Windows Store ou Windows Phone).

a. Templates

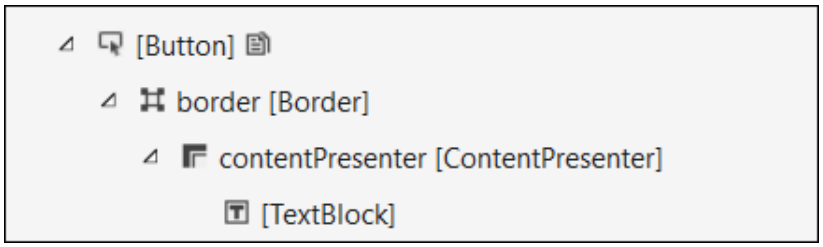
Les objets sont définis sous la forme d'un graphe composant l'arbre logique d'un écran. Cette arborescence est différente de l'arbre visuel, qui est le résultat de la transformation de l'arbre logique après application de différents *templates* sur chaque contrôle.

La partie visuelle d'un contrôle graphique est définie en XAML par la création d'un objet de type `ControlTemplate`. Celui-ci est un modèle de dessin représentant le résultat souhaité à l'aide d'une arborescence constituée d'objets plus simples. Il est appliqué au contrôle lors de son chargement.

L'arbre logique suivant est extrêmement simple : il consiste en une seule balise.

```
<Button Height="30" Width="120" Content="Ceci est un bouton" />
```

L'arbre visuel correspondant est un peu complexe :



Cette représentation de l'arbre visuel d'un bouton a été obtenue à l'aide de la fenêtre **Arborescence d'éléments visuels en direct** de Visual Studio. Celle-ci peut être ouverte lors du débogage d'une application WPF à partir du menu **Déboguer - Fenêtres - Arborescence d'éléments visuels en direct**.

Deux éléments ont été ajoutés : un objet de type `Border` et un objet de type `ContentPresenter`. Ceux-ci font partie du `ControlTemplate` associé la classe `Button`. L'objet de type `TextBlock`, ajouté par le `ContentPresenter`, a quant à lui encapsulé notre texte dans sa propriété `Text`.

b. Espaces de noms

Comme pour le code C#, il est nécessaire d'importer des espaces de noms dans le code XAML pour pouvoir utiliser les types qu'ils contiennent. Pour réaliser cette opération, il n'est évidemment pas possible d'utiliser une déclaration `using` de C#, mais les déclarations d'espaces de noms de XML sont tout indiquées pour cela. Ces déclarations prennent la forme d'attributs XML respectant une syntaxe bien particulière :

```
xmlns:<prefix>="clr-namespace:<espace de noms>"
```

Deux espaces de noms sont obligatoires dans chaque fenêtre WPF et doivent être présents au niveau de l'élément racine. Le premier importe le framework WPF et son implémentation de XAML, tandis que le second importe certains éléments globaux de XAML.

Ces deux déclarations permettent donc d'utiliser les bases de WPF dans l'ensemble du document en cours d'édition.

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Lorsqu'une fenêtre est créée à l'aide du modèle présent dans Visual Studio, sa déclaration inclut déjà ces espaces de noms.

Fort heureusement, l'import d'espaces de noms .NET est généralement plus simple. Pour l'espace de noms `Exemple.Controles`, la déclaration sera faite de la manière suivante :

```
xmlns:controlesExemples="clr-namespace:Exemple.Controles"
```

Le contrôle `Calendrier` situé dans cet espace de noms est ensuite utilisé en préfixant son nom de type par l'alias d'espace de noms `controlesExemple` suivi du symbole :, ce qui donne l'instanciation suivante :

```
<controlesExemple:Calendrier />
```

3. Contexte de données et binding

Le concept de **binding** est peut-être le plus important pour faire une utilisation optimale des capacités de WPF. Il correspond à une **liaison unidirectionnelle ou bidirectionnelle** entre un élément XAML et un **contexte de données** (`DataContext`) et permet la propagation de données du code C# vers le code XAML (et/ou inversement) sans qu'il ne soit nécessaire pour le développeur d'effectuer ce transfert d'informations à l'aide de code C#. Ce mécanisme est la clé du **découplage** entre le code métier et l'interface graphique.

Une liaison de données est définie dans le code XAML par une expression délimitée par des accolades et dont le premier terme est le mot-clé `Binding`. Ce dernier peut être suivi par une liste de paramètres définissant la source de données, le chemin de la propriété avec laquelle la liaison doit être faite ou encore le mode de liaison.

Par défaut, un binding transmet le résultat de l'appel de la méthode `ToString` de l'objet. Pour les objets complexes, le résultat affiché si cette méthode n'a pas été redéfinie est le nom complet de son type.

- Créez un nouveau projet WPF dans Visual Studio à l'aide de la boîte de dialogue **Nouveau projet** (menu **Fichier - Nouveau - Projet** ou [Ctrl][Shift] N) dans laquelle il faut sélectionner **Application WPF** et nommez-le **EssaisBinding**.
- Éditez le contenu du fichier `MainWindow.xaml` du projet nouvellement créé de manière à ce qu'il contienne le code suivant :

```
<Window x:Class="EssaisBinding.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <TextBox Height="30" Width="120" Text="{Binding Path=Nom,
Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
    </Grid>
</Window>
```

Ce code crée une zone de saisie dans une fenêtre et associe à sa propriété `Text` une liaison bidirectionnelle (`Mode=TwoWay`) vers la propriété `Nom` du contexte de données (`Path=Nom`). Le transfert des informations du contrôle vers le contexte de données sera effectué à chaque fois que la propriété sur laquelle est défini le binding est modifiée (`UpdateSourceTrigger=PropertyChanged`).

- Lancez le projet en utilisant la touche [F5] ou le bouton **Démarrer** de la barre d'outils Visual Studio (en ayant au préalable défini le projet comme projet de démarrage).

En l'état, rien de notable ne se produit. En effet, aucun contexte de données n'est défini et aucune source de données n'est explicitement spécifiée. Pour définir le contexte de données de la fenêtre, il convient de valoriser sa propriété `DataContext`.

- Éditez le constructeur de la fenêtre, dans `MainWindow.xaml.cs`, de manière à définir le contexte de données de la fenêtre comme étant la fenêtre elle-même.

```
public MainWindow()
{
   this.InitializeComponent();
   this.DataContext = this;
}
```

Si vous lancez l'application à ce stade, l'interface se comportera exactement de la même manière. En revanche, la fenêtre **Sortie** de Visual Studio présente une nouvelle ligne d'information :

```
System.Windows.Data Error: 40 : BindingExpression path
error: 'Nom' property not found on 'object' ''MainWindow'
(Name='')'. BindingExpression:Path=Nom;
DataItem='MainWindow' (Name=''); target element is 'TextBox'
(Name=''); target property is 'Text' (type 'String')
```

Ce message indique qu'une erreur s'est produite dans la création de la liaison de données car la propriété `Nom` n'a pas été trouvée dans l'objet `MainWindow` utilisé comme contexte de données. Il fournit également des informations permettant d'identifier l'objet XAML et la propriété sur lequel la liaison erronée est placée.

- Ajoutez la définition de propriété suivante dans la classe `MainWindow` de manière à corriger l'erreur.

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

- Pour visualiser le bon fonctionnement de la liaison, ajoutez une affectation de valeur à la propriété `Nom` dans le constructeur de la classe.

```
public MainWindow()
{
    InitializeComponent();
    Nom = "Sébastien";
    this.DataContext = this;
}
```

Dans cette configuration, l'application affiche une zone de saisie dans laquelle la valeur de la propriété `Nom` est bien présente, mais les modifications apportées à cette valeur lors d'un clic sur un bouton, par exemple, ne seront pas répercutées dans le contrôle `TextBox` malgré l'existence d'une liaison de données.

- Pour le vérifier, ajoutez la ligne suivante juste après la déclaration du contrôle `TextBox` dans `MainWindow.xaml` :

```
<Button Height="30" Width="120" VerticalAlignment="Top"
Click="Button_Click" Content="Modifier le contenu de la zone de
saisie" />
```

- Ajoutez également le gestionnaire d'événements suivant à la classe `MainWindow` :

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Nom = "Nom modifié";
}
```

La non-propagation de la modification est tout à fait normale. La technologie WPF utilise l'événement `PropertyChanged` de l'interface `INotifyPropertyChanged` pour informer les éléments graphiques des modifications de valeur sur les propriétés. Il est donc nécessaire que l'objet utilisé comme source de données (ici, la classe `MainWindow`) implémente cette interface.

Ci-dessous le code complet de la classe `MainWindow` après implémentation de cette interface :

```
public partial class MainWindow : Window, INotifyPropertyChanged
{
    private string nom;
    public string Nom
    {
        get { return nom; }
        set
        {
            nom = value;
            OnPropertyChanged("Nom");
        }
    }

    public MainWindow()
    {
        InitializeComponent();
```

```
        this.DataContext = this;
        Nom = "Sébastien";
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        this.Nom = "Nom modifié";
    }

    private void OnPropertyChanged(string nomPropriete)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new
PropertyChangedEventArgs(nomPropriete));
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```