Formations .NET – Audit, Conseil, Développement

Articles gratuits à télécharger www.e-naxos.com
Dot.Blog, le blog www.e-naxos.com/blog

Dix raisons de choisir WPF

[Une introduction à WPF par l'exemple]

Version 1.04

Sommaire

Table des figures

Introduction : Choisir WPF ?

La question se pose-t-elle vraiment aujourd'hui ? Car finalement il semble une évidence que l'avenir du développement est graphique et que les boutons rectangulaires et tristes autant que les affichages « plats » hérités de Win32 ont fait plus que leur temps. La seule plateforme graphique cohérente de l'instant est WPF et ses dérivées comme Silverlight pour le Web. Alors est-il légitime de poser la question du choix aujourd'hui ?

Je crois que oui et pour plusieurs raisons. La première est que l'adoption de WPF est loin d'être encore totale et qu'une large majorité de projets .NET sont encore débutés en Windows Forms au moment même où j'écris ces lignes. Je peux en comprendre les raisons, mais il s'agit à mon sens d'une erreur commise par manque de compréhension des véritables avantages de WPF.

La seconde raison est que WPF implique un changement assez grand des habitudes de développement, une autre façon de concevoir les interfaces, d'autres outils (Expression Blend, Expression Design), d'autres compétences (celle des graphistes notamment) et que cela peut créer une certaine appréhension bien légitime qu'il convient de dissiper.

Pour de nombreuses personnes WPF apparaît ainsi comme un « luxe » voire une « débauche d'effets spéciaux » bien trop « en avance » pour leurs utilisateurs. Je l'ai entendu, et je l'entends encore trop souvent…

Cela est bien entendu une vision très subjective et surtout faussée de la réalité. Une façon de se protéger contre le changement ? Peut-être mais comme Vista aura été une génération d'OS sacrifiée, alors que ceux qui n'en veulent pas aujourd'hui sauteront sur Seven (qui n'est autre qu'un Vista avec des aménagements), WPF et son étiquette « pur graphique hyper looké » pourrait pâtir de cette image, excuse facile pour qui refuse la nouveauté ou pour tous ceux qui, trop méfiants de nature, attendent qu'une technologie soit presque morte et dépassée pour commencer à l'adopter…

Il est vrai que WPF permet enfin de développer des applications **aussi belles que fonctionnelles**. Il est tout aussi vrai que changer un bouton rectangulaire et triste en un bouton ovale animé ne réclame plus de sous-classer un contrôle Windows en s'empêtrant dans les messages de l'OS et les mécanismes complexes de la librairie sous-jacente puisque WPF repose sur un format descriptif, **Xaml**, *conçu pour supporter toutes les fantaisies graphiques*.

Mais WPF est-il pour autant un environnement uniquement dédié aux jeux ou aux éditeurs de logiciels (chez qui la concurrence est si vive que le look peut faire la vente plus que la fonctionnalité) ?

Je ne le crois pas. WPF représente aussi *une fantastique avancée* en termes de modèle de développement d'interface et **c'est bien techniquement que cette solution est de loin préférable** à Windows Forms.

Je ne suis pas le seul à le penser, par exemple Josh Twist dans une série de billets écrits entre 2007 et 2008 avait choisi de donner 10 bonnes raisons d'utiliser WPF. Bien plus qu'un long discours, chacun des points qu'il soulève dans cette série est en soi une raison valable de préférer WPF. Évidemment, 10 raisons cela sonne arbitraire et ses choix ne semblent pas tous aussi essentiels, de même il s'en tient au Xaml écrit à la main en faisant l'impasse sur des outils

comme Blend. Des raisons il y en a certainement plutôt 50 ou 100 ! Mais l'exercice de style que représente le choix de 10 *bonnes* raisons est une gageure.

Je n'aime pas traduire des articles, c'est un exercice servile dans lequel je ne me sens pas à l'aise, j'ai toujours mon grain de sel à ajouter et dans ce cas je trahis l'auteur. Traduttore, Traditore ! (Traducteur, traître !) disent les italiens avec raison… Alors plutôt que de traduire et trahir les billets de Josh je renvoie le lecteur intéressé à la source (www.thejoyofcode.com). En revanche j'ai trouvé que certaines des dix raisons qu'il propose sont très pertinentes et que la logique de certains de ses exemples était intéressante. Je vais donc ici, telle une abeille qui mâche la goutte de pollen laissée par sa sœur pour mieux la recracher à son tour dans un long cycle qui en fera un miel subtil, faire mien certains des arguments de Josh et certains de ses exemples pour mieux les développer à ma façon et vous fournir ma propre goutte de miel.

Let's go !

Raison 1 : Le modèle de contenu riche

Le modèle Windows Forms (comme celui des MFC ou de la VCL Delphi) est un cadre stricte et peu évolutif. Il existe des composants commerciaux permettant d'améliorer le « look and feel » des applications mais tout cela est si peu standard que beaucoup de ces composants programmés de façon plus ou moins habile posent des problèmes à un moment ou un autre, en dehors de créer une dépendance à un fournisseur tiers quand ce n'est pas à plusieurs (et dans ce cas s'ajoute l'incompatibilité entre ces diverses bibliothèques).

Les développeurs Web ont depuis longtemps pris l'habitude de pouvoir adapter le contenu graphique des pages grâce à l'utilisation d'images, de séquences vidéo, des feuilles de styles, de JavaScript et autres artifices venus compléter Html. **Mais jamais autant de souplesse n'a été permise aux développeurs d'applications Windows**. **Pourtant le besoin est le même**, **les utilisateurs sont les mêmes**, alors où est le problème ? … C'est le modèle Win32 qui est coupable. Trop rigide, obligeant à créer ses propres contrôles pour modifier l'aspect d'un simple bouton, imposant une mise en page rectangulaire et plate, ce modèle créé un *couplage bien trop fort entre le code et la représentation des informations*. La création de composants réclame, de plus, des compétences techniques très largement au dessus du niveau du développeur moyen, quel que soit l'environnement Win32 (VC++, VB, Delphi…).

Windows Forms, système temporairement mis en place sous .NET en attendant WPF, est directement l'héritier du modèle Win32 auquel il n'ajoute rien de plus, en dehors d'une programmation plus cohérente (grâce à la qualité du Framework).

Créer des contenus riches cela ne veut pas dire transformer une application en jeu vidéo pour autant !

WPF est conçu pour simplifier la personnalisation des interfaces sans impacter sur le code. C'est une avancée énorme. Quelque chose que nous attendions tous depuis les premiers environnements de développement sous Windows. Pouvoir facilement **rendre une interface plus gaie et plus ergonomique n'est pas un luxe, c'est un progrès** sur lequel aucun modèle de programmation ne reviendra dans le futur. Autant choisir d'emblée cette voie au lieu de

rester du mauvais côté de l'aiguillage ! Ceux qui penseraient que WPF et ses affichages très graphiques sont un « luxe inutile » disaient la même chose de Windows, de ses fenêtres et de sa souris il y a plus de vingt ans et pourtant ils ne programmeraient plus aujourd'hui en mode console ! Mais plutôt que de les montrer du doigt, expliquons leur, par l'exemple, que leurs craintes sont infondées.

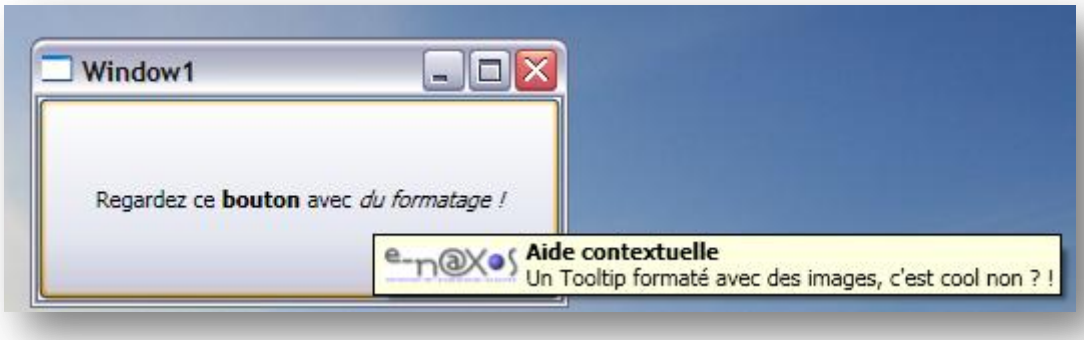


Figure 1 - Projet Wpf10-1

Sur cette illustration vous noterez la présente d'un bouton (occupant toute la fiche, c'est juste un exemple) contenant un texte formaté. Mieux, il dispose d'une aide contextuelle, elle aussi formatée et intégrant en plus une image. Combien de lignes de code pour réaliser cela en Windows Forms ?

Ne comptez pas. Ce n'est pas directement possible, tout simplement. Un bouton supportant du texte RTF cela existe, mais il faut l'acheter ou le programmer. Un `ToolTip` supportant à volonté du RTF et des images, cela existe aussi, mais à quel prix et ces deux composants seront-ils compatibles entre eux, évolueront-ils assez vite pour suivre le Framework ? J'arrête les questions faussement candides, vous vous doutez que les réponses sont embarrassantes…

Sous WPF tout cela est naturel, direct et immédiat. Si nous regardons le code Xaml de la fiche voici ce que nous trouvons (si, vraiment, il n'y a que ça pour cette application, pas de C#) :

```
<Button>
    <Button.ToolTip>
      <StackPanel Orientation="Horizontal">
        <Image Source="E-NAXOX LOGO 2006 mini.jpg" Margin="3"/>
        <TextBlock>
      <Run FontWeight="Bold">Aide contextuelle</Run>
      <LineBreak/>
      Un Tooltip formaté avec des images, c'est cool non ? !
        </TextBlock>
      </StackPanel>
    </Button.ToolTip>
    <TextBlock>Regardez ce <Run FontWeight="Bold">bouton</Run> avec <Run FontStyle="Oblique">du formatage
!</Run></TextBlock>
</Button>
```

Ces quelques lignes de *code parfaitement standard et sans librairie externe* ne valent-elles pas mieux que la création ou l'achat d'une bibliothèque de composants qui seront obsolètes ou mal maintenus ? A vous de voir…

Le modèle riche proposé par WPF avec Xaml est la première bonne raison de choisir cet environnement pour vos développements, j'en suis convaincu.

Raison 2 : La liaison aux données

Le Data Binding du Framework .NET est l'un de ses grands points forts si on se rappelle que sous les MFC ou la VCL il fallait une gymnastique complexe pour créer des composants liés aux données. Étendre la notion de « données » à tout objet a été un progrès immense.

Les Windows Forms tirent profit de ce Data Binding et bien que similaires aux autres modèles Win32 cités elles leur sont supérieures ne serait-ce que grâce à la grande qualité du Framework sur ce point.

Mais le Data Binding des Windows Forms reste au niveau de ce qui a été introduit dans le Framework 1.x, rien de plus. Et pourtant il a tant évolué jusqu'à aujourd'hui ! Certes Windows Forms 2.0 profitait des avancées du Framework pour aller un cran plus loin, mais on restait très en deçà de ce que le Framework lui-même permettait et laissait espérer de faire tellement **la puissance du Data Binding va bien au-delà des bases de données** (idée première et fausse qu'on s'en fait, autant que sur LINQ mais c'est une autre sujet !).

Certains n'hésitent d'ailleurs pas à dire que si vous n'utilisez pas aujourd'hui de Data Binding dans votre application c'est que vous avez probablement fait une erreur de développement…

Mais faisons fi des débats idéologiques et laissons parler le code !

Pour illustrer le Data Binding de WPF nous avons besoin d'une source de données. Une liste de synthétiseurs vintage fera parfaitement l'affaire :

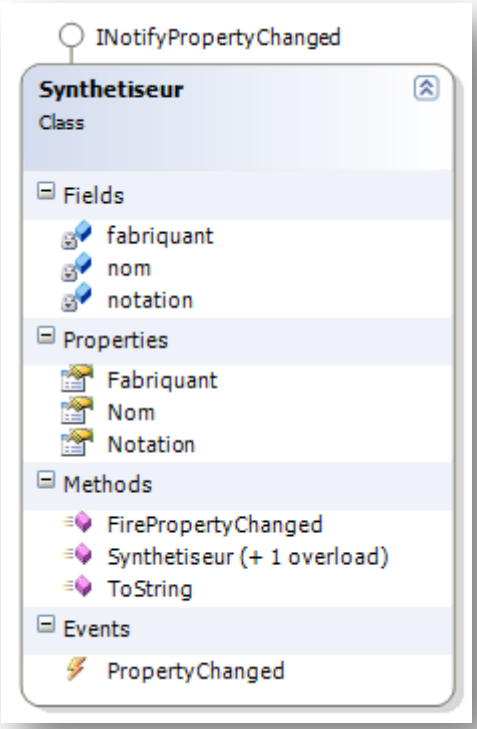


Figure 2 - La classe Synthétiseur

La classe est ici fort simple et expose trois propriétés. Comme vous le noterez sur le diagramme ci-dessus elle implémente l'interface `INotifyPropertyChanged` ce qui n'est pas un besoin propre à WPF et qui s'impose pour tout objet bien écrit.

Les données elles-mêmes seront créées à la volée par le constructeur de la page sous la forme d'une liste `ObservableCollection` qui sera affectée au `DataContext` de cette même page.

C'est là que se joue le nouveau Data Binding WPF : une page, comme d'autres objets (tous ceux descendant de `FrameworkElement`), possède un **contexte de données** (le fameux `DataContext`). Les objets contenus dans la page peuvent se référer à ce dernier de façon extrêmement simple puisqu'il suffit de nommer les propriétés pour s'y « accrocher ».

Ainsi, un `TextBox` sera relié à la propriété `Fabriquant` du `DataContext` (la liste de synthétiseurs) de la page sous la forme suivante dans le code Xaml :

```
<TextBox Grid.Column="1" Grid.Row="1" Margin="3" Text="{Binding Fabriquant}" />
```

C'est tout…

La liste `ObservableCollection` contenant les instances est accrochée au `DataContext` de la page comme cela :

```
public Page1()
    {
      InitializeComponent();
```

```
DataContext = new ObservableCollection<Synthetiseur>
        {
            new Synthetiseur("MiniMoog", "Moog", 5),
            ...
```

Ajoutons ainsi trois boites de saisie et une liste qui sera reliée au même `DataContext` de la page comme suit :

```
<ListBox Name="lbSynth" ItemsSource="{Binding}" Margin="3" />
```

Vous noterez que ce morceau de Xaml ainsi que le précédent ne se chargent pas seulement du Data Binding, ils créent aussi l'objet, le nomme et en fixe certains paramètres de placement comme la marge. Sans autre indication sur le champ à afficher, le binding utilisera ici la méthode `ToString` des objets pour afficher le texte. Il est toujours important de surcharger cette méthode dans une classe métier, c'est une bonne pratique à appliquer systématiquement.

Xaml peut être plus verbeux que cela et certains extraits de code que vous pouvez avoir vus ici ou là peuvent décourager d'autant que certains développeurs ont la manie du mysticisme technique… Les mêmes voulaient hier qu'un « bon » développeur Web écrive le Html à la main sous notepad par exemple. Laissons ces dinosaures dans leur enclos et sachez que l'éditeur de Visual Studio et plus encore celui de Expression Blend sont des aides précieuses évitant d'avoir à produire du code Xaml à la main. Le code montré ici ne sert qu'à illustrer la puissance descriptive de Xaml et la simplicité du Data Binding WPF. Ne l'oubliez donc pas en lisant cet article, Xaml se travaille majoritairement de façon visuelle et graphique avec les outils *ad hoc*.

Avec un peu de mise en page (très peu) notre application se présente comme cela :

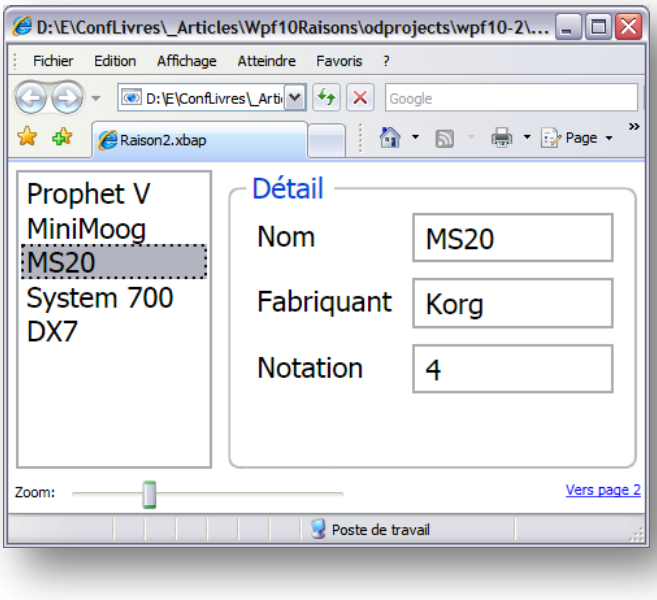


Figure 3 - Projet Wpf10-2

L'application de test est construite ici selon le mode `Xbap`, un modèle particulier d'application WPF s'exécutant dans Internet Explorer. Ne pas confondre avec Silverlight, basé aussi sur WPF

et s'exécutant aussi via le Web mais au travers d'un plugin léger multi plateforme et multi browser. L'exemple précédent était d'ailleurs développé en mode WPF desktop, plus traditionnel. Le choix de Xbap est donc ici purement arbitraire.

La liste ainsi que les `Textbox` sont synchronisés automatiquement au `DataContext` de la page, cette liaison étant par défaut à double sens il est possible de modifier par exemple le nom du synthétiseur à l'écran, cette modification étant portée directement dans l'instance sous-jacente. De même la liste affichant la propriété « `Nom` » sera mise à jour en temps réel. Il en irait de même si l'objet changeait de façon interne, l'affichage serait alors mis à jour (grâce au fait que l'objet supporte `INotifyPropertyChanged`).

Pour que cette mise à jour de l'affichage soit instantanée il nous suffit d'améliorer le binding du `TextBox` représentant la propriété `Nom` :

```
<TextBox Grid.Column="1" Grid.Row="0" Margin="3" Text="{Binding Nom,
UpdateSourceTrigger=PropertyChanged}" />
```

Vous noterez l'ajout d'un `UpdateSourceTrigger` connecté à l'événement `PropertyChanged` de l'objet. Grâce à ce trigger la propagation de la modification du nom sera immédiate (sinon elle interviendrait lorsque le champ est validé, sur la perte de focus par exemple).

Pour vous prouver que le Data Binding de WPF est d'une grande versatilité, regardons de plus près le `scroller` servant à gérer le zoom. En le bougeant tout le contenu de la fiche est zoomé en avant ou en arrière. Par quelle magie, quelle quantité de code ?

```
<Grid.LayoutTransform>
        <ScaleTransform ScaleX="{Binding ElementName=_sliderZoom, Path=Value}"
        ScaleY="{Binding ElementName=_sliderZoom, Path=Value}" />
</Grid.LayoutTransform>
```

Dans cet exemple tout le contenu de la page est accroché à un conteneur de type grille et dans les propriétés de cette dernière nous avons ajouté le code ci-dessus. Regardez de plus près : ce code affecte *une transformation* de mise en page (`Grid.LayoutTransform`) de type changement d'échelle (`ScaleTransform`). Et tout naturellement l'échelle des deux axes X et Y est relié à la valeur courante (`Value`) du `slider` gérant le zoom (`_sliderZoom`). C'est tout, une fois encore le reste est automatique !

Mais il y encore un petit bonus … Regardez en base à droite de la page, vous trouvez un lien hypertexte affichant « `Vers page 2` ». En cliquant sur ce dernier la page 2 de l'exemple vient remplacer la page en cours.

Le code Xaml définissant ce lien et son action est le suivant :

```
<TextBlock Grid.Column="2" Margin="3" HorizontalAlignment="Right" >
        <Hyperlink NavigateUri="Page2.xaml">Vers page 2</Hyperlink>
</TextBlock>
```

Même sans connaître Xaml il n'est guère difficile de voir à quel point tout cela est limpide et direct… et sans code !

La page 2 ressemble à cela :

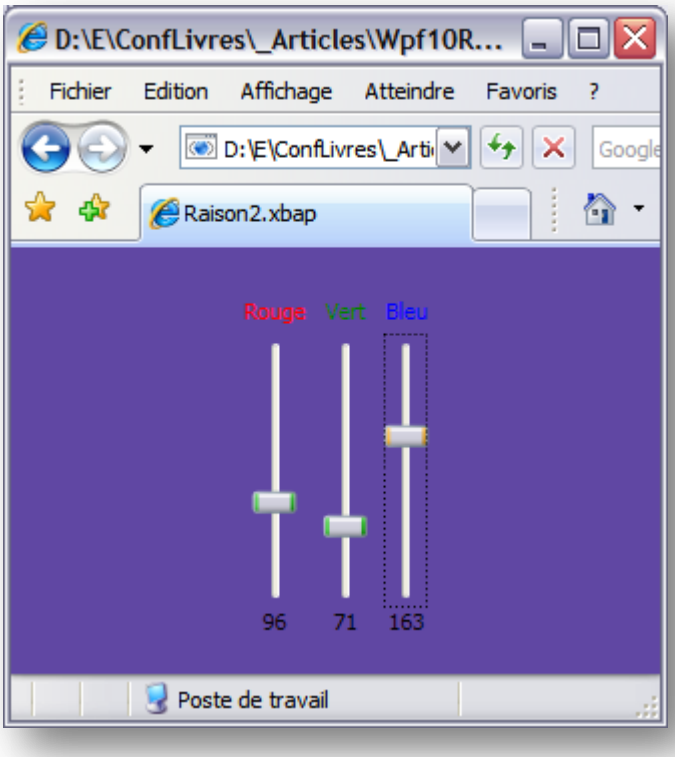


Figure 4 - La page 2 avec ses trois sliders

Trois sliders permettent de faire varier les valeurs R, G et B de la couleur appliquée au fond de page.

La magie s'opère de la même façon, grâce au Data Binding WPF :

```
<Page.Background>
        <MultiBinding Converter="{StaticResource colorConv}">
                <Binding ElementName="_sliderRed" Path="Value"/>
                <Binding ElementName="_sliderGreen" Path="Value"/>
                <Binding ElementName="_sliderBlue" Path="Value"/>
        </MultiBinding>
</Page.Background>
```

Les trois sliders voient leur propriété « `Value` » connectée par Data Binding au fond de la page (`Page.Background`). Le mécanisme est ici un peu plus sophistiqué car nous devons résoudre deux problèmes spécifiques : d'une part le fait que ce binding implique trois sources différentes pour une même propriété (le fond) et que la valeur des sliders ne retourne pas directement une couleur mais un entier.

Pour régler le premier point WPF nous offre une solution simple, le `MultiBinding`. Sans entrer dans les détails vous en comprenez immédiatement l'intérêt, ce mode de liaison permet de regrouper plusieurs valeurs pour en produire une seule en sortie.

Pour le second point la logique WPF se sophistique un peu : vous remarquez que le `MultiBinding` fait référence à un convertisseur. Ce dernier provient d'une ressource que nous avons créée (`StaticResource`) et s'appelle `colorConv`.

Sa définition dans la page :

```
<Page.Resources>
        <src:ColorConverter x:Key="colorConv" />
</Page.Resources>
```

Ici la ressource « `colorConv` » est créée à partir de la classe « `ColorConverter` ». Cette dernière est instanciée automatiquement par cette déclaration. Pas de code C#, pas besoin de faire un « `new` ».

Quant au code du convertisseur lui-même cela sort un peu du cadre de cet article, je vous laisse le soin de lire le code source. Disons juste que ce mécanisme est très utilisé sous WPF et Silverlight car il permet d'adapter des propriétés à la base incompatibles entre elles et ce de façon automatique. Comme vous le verrez d'ailleurs, ce code ne fait que quelques lignes.

Nous avons ici tout juste effleuré le Data Binding de WPF mais vous devez maintenant mieux comprendre à quel point il est **une brique essentielle de ce modèle de développement** et combien sa souplesse modifie en profondeur le codage ainsi que la séparation, enfin réelle, entre interface et code applicatif.

Il s'agit bien d'une vraie bonne raison de choisir WPF pour développer, pas de doute là-dessus.

Raison 3 : les DataTemplates

Les `DataTemplate` sont des modèles permettant de contrôler le formatage des données. En réalité les `DataTemplate` s'inscrivent dans une logique plus vaste, celle du templating tout cours, nous verrons cela au point suivant.

Les templates sont même à la base de la programmation visuelle de WPF et Silverlight (avec les Styles). Grâce à ces techniques il est possible de modifier l'aspect de toute une classe d'éléments visuels, de stocker les modèles en ressource locale ou partagée par toute l'application, voire de les stocker dans des fichiers de ressources particuliers, les dictionnaires, partageables entre plusieurs applications.

Reprenons pour commencer la `ListBox` de l'exemple précédent. Sa définition était la suivante :

```
<ListBox Name="lbSynth" ItemsSource="{Binding}" Margin="3" />
Ou dans une version spécifiant le champ à afficher :
<ListBox Name="lbSynth" ItemsSource="{Binding}" Margin="3" DisplayMemberPath="Nom"/>
```

Imaginons maintenant que nous souhaitions gérer un `ToolTip` pour chaque ligne affichée par la listbox et que ce tooltip contienne le nom du fabriquant du synthétiseur survolé par la souris. Comment feriez-vous cela sous Windows Forms ? Sous WPF il suffit de modifier légèrement la balise Xaml (ce qui peut se faire non par code mais visuellement sous Blend) :

```
<ListBox Name="lbSynth" ItemsSource="{Binding}" Margin="3">
      <ListBox.ItemTemplate>
         <DataTemplate>
            <TextBlock Text="{Binding Nom}" ToolTip="{Binding Fabriquant}"/>
         </DataTemplate>
      </ListBox.ItemTemplate>
</ListBox>
```

Xaml suit un formalisme XML, il est donc verbeux, ne vous laissez pas submerger par le texte, regardez plutôt la logique des balises. Tout d'abord la balise de définition du listbox a été transformée pour que sa fermeture se trouve détachée, ce qui permet d'ajouter d'autres balises entre l'ouverture et la fermeture. C'est du XML tout simplement.

Ensuite nous avons introduit un `ItemTemplate`, c'est-à-dire un modèle qui s'applique à chaque item de la liste. Ce template est lui-même définit sous la forme d'un `DataTemplate` qui, *in fine*, contient une `TextBlock` pour afficher le nom de l'objet et un `ToolTip` connecté par Data Binding au champ `Fabriquant` de l'objet.

Et voilà. Rien de plus. Pas de programmation en code behind ou autre. Juste des balises décrivant simplement ce que nous voulons obtenir. Xaml autorise un mode de programmation totalement descriptif, **vous expliquez ce que vous voulez faire, et non comment il faut le faire**.

Bien entendu cet exemple est simplificateur, l'aspect visuel ne sera pas forcément digne de la Guerre des Etoiles, certes. Mais c'est le principe qui compte et créer des sapins de Noël n'est d'ailleurs pas forcément votre objectif ! Pour aller plus loin visuellement il est préférable d'utiliser Expression Blend qui écrit à votre place le code Xaml lorsque vous modifiez visuellement un template. Choisir les bons outils c'est comme savoir choisir ses amis, ça change la vie.

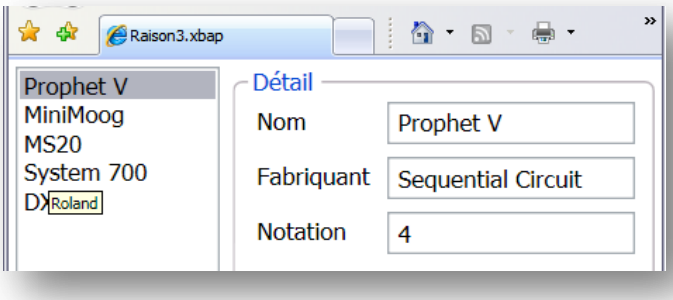


Figure 5 - Le ToolTip en action

L'image ci-dessus montre l'effet de notre `DataTemplate` (la souris étant placée sur « System 700 » ce qui ne se voit pas sur cette capture).

Ce n'est pas mal, mais on peut aller un peu plus loin en utilisant ce principe très riche et ouvert des `DataTemplate`. Imaginons alors que nous souhaitions afficher l'image de chaque synthétiseur à gauche du nom dans la listbox et que cette image provienne d'Internet…

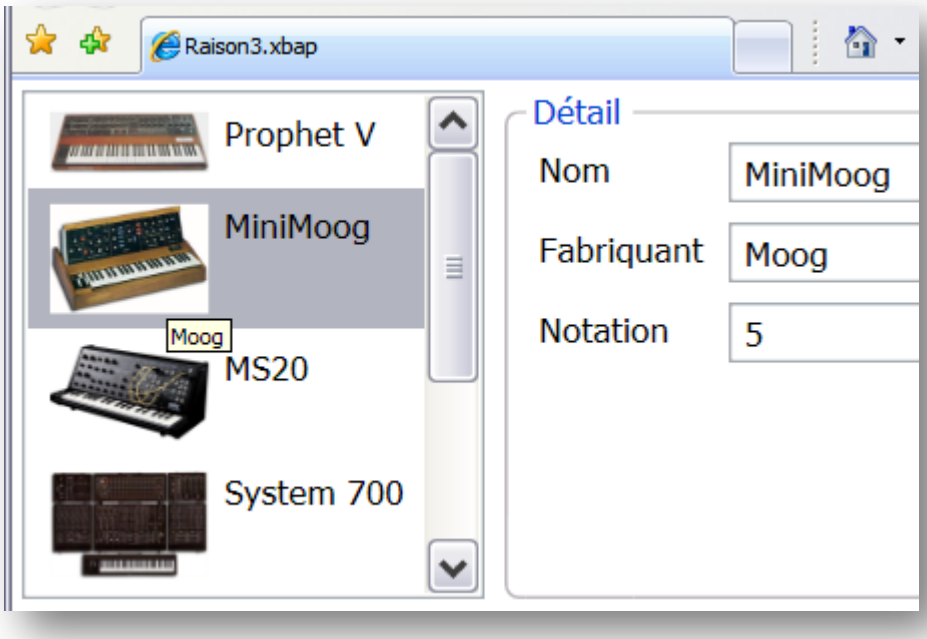


Figure 6 - Projet Wpf10-3

C'est nettement mieux maintenant… Le code ? quel code ? C# ? Nop. Il n'y en a aucun. En revanche nous avons un peu complété la balise du `DataTemplate` de la `ListBox`, et comme le montre les lignes ci-dessous nous n'avons pas modifié grand-chose :

```
<ListBox Name="lbSynth" ItemsSource="{Binding}" Margin="3">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal" ToolTip="{Binding Fabriquant}">
                <Image Source="{Binding ImageUrl}" Width="50" Margin="5" />
                <TextBlock Text="{Binding Nom}" Margin="0,5,0,0"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Ici nous avons remplacé le `TextBlock` par un ensemble d'autres composants. Tout d'abord nous avons placé un `StackPanel` orienté horizontalement. Il s'agit d'un conteneur mettant ses objets enfants les uns derrière les autres, soit verticalement (comme une listbox) soit horizontalement, ce que nous avons choisi ici. Le `ToolTip` a été déplacé pour être accroché à ce conteneur. À l'intérieur nous avons placé deux composants, une image et l'ancien `TextBlock`. Ce dernier est toujours lié à la propriété `Nom` du `DataContext` de la page, et l'image est liée à une nouvelle propriété de la classe `Synthetiseur`, `ImageUrl`. Une propriété de type `string` qui contient tout simplement l'Url de l'image à afficher :

... new Synthetiseur("System 700", "Roland",
5,"http://www.vintagesynth.com/roland/images/roland_system700.jpg") ...

Pas de composant spécifique pour se lier à Internet ou télécharger l'image en mémoire, rien de tout cela, juste une simple chaîne de caractère liée par Data Binding à un composant image. Pas mal non ?

Et voici comment une poignée de lignes de Xaml peuvent remplacer des composants spécialisés et du code plus ou moins complexe. Il est facile de comprendre comment les coûts de maintenance et même de conception sont limités par une telle approche, d'autant que l'aide d'un graphiste n'est pas forcément indispensable pour des interfaces simples.

Nous pourrions complexifier l'exemple encore longtemps, ajoutant ceci ou cela pour rendre la démonstration encore plus éclatante, mais je pense sincèrement que les 10 lignes de Xaml définissant la `ListBox` de cet exemple remplacent tous les longs discours.

Nous n'avons fait que survoler de très haut les `DataTemplate` WPF. Mais on voit clairement qu'ils autorisent **un style de développement nouveau** et d'une **puissance sans équivalent** dans les anciens modèles de programmation.

La 3ème bonne raison de choisir WPF ce sont les `DataTemplate`. Indiscutable.

Raison 4 : Le templating des contrôles

Il est peut être un peu spécieux de revenir ainsi sur le templating pour créer une quatrième raison qui risque d'apparaître artificielle. Mais en réalité nous avons vu peu de chose, juste les `DataTemplate`. Rien à propos du templating des contrôles. Et cette possibilité est **l'un des piliers de WPF et de Silverlight** grâce à qui une simple fiche peut devenir une magnifique application offrant une expérience utilisateur unique.

La raison 3 montrait les `DataTemplate`, un moyen puissant de créer des modèles pour des données à afficher, par exemple dans une `ListBox`.

Mais en réalité le templating va beaucoup loin et il est tout à fait légitime de présenter les templates de contrôles totalement à part. Revenons ainsi sur l'exemple utilisé au point précédent. Pour agrémenter encore plus l'affichage de la listbox nous pourrions souhaiter montrer une barre qui indique visuellement la notation attribuée à chaque synthétiseur. Cela donnerait la chose suivante :

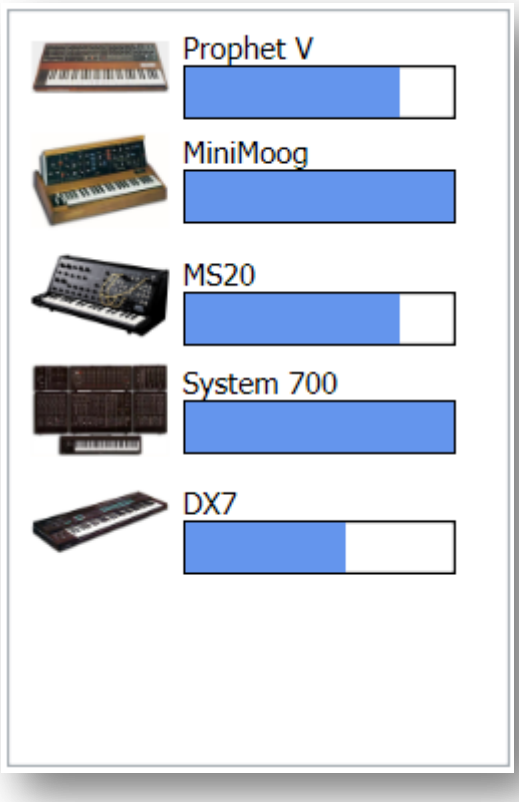


Figure 7 - Projet Wpf10-4

Tout comme l'ajout de l'image, l'affichage de la barre de notation donne un aspect bien plus fini, bien plus « pro » à notre listbox qui reste malgré tout une `ListBox` **standard**, ne l'oublions pas.

Ici la barre a été réalisée en imbriquant un autre conteneur qui lui-même contient le nom à afficher ainsi qu'une `ViewBox` (surface de dessin) dans laquelle est dessiné un rectangle dont la largeur est couplée, par Data Binding, au champ entier `Notation` de chaque objet de la liste.

La méthode n'est pas parfaite et frôle le bricolage pour être franc. Car il y a bien plus simple sous WPF ! En effet, il existe un contrôle qui sait déjà afficher une jolie barre, c'est le `ProgressBar`. Utilisons-le en place et lieu du dessin du rectangle :

```
<ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal" ToolTip="{Binding Fabriquant}">
        <Image Source="{Binding ImageUrl}" Width="50" Margin="5" />
        <StackPanel Margin="0,5,0,0" HorizontalAlignment="Stretch">
            <TextBlock Text="{Binding Nom}"/>
            <ProgressBar Value="{Binding Notation}" Minimum="0" Maximum="5" MinHeight="16"/>
        </StackPanel>
      </StackPanel>
    </DataTemplate>
</ListBox.ItemTemplate>
```

Le code reste très simple, et sans autre arrangement, cela donne le résultat suivant :

Figure 8 - Projet Wpf10-4b

C'est un peu mieux, mais d'une part le composant n'a pas une largeur fixe, ce que nous pouvons arranger facilement, et d'autre part cela ressemble surtout bien trop à une `ProgressBar` justement !

Faut-il sous-classer la `ProgressBar` et écrire un nouveau composant s'affichant différemment ? Faut-il chercher un autre composant ou en acheter un ?

Non et non. **WPF sait répondre à ces problématiques de façon native et simple par le templating des contrôles**.

Plutôt que de suivre les exemples habituels montrant encore et encore du code Xaml, je vais vous faire voir comment relooker la barre de progression sous **Expression Blend** qui est l'outil conçu pour ce travail.

Notre projet peut rester ouvert sous Visual Studio, cela n'est pas gênant, Blend et VS savent se synchroniser quand on passe de l'un à l'autre. Nous allons maintenant ouvrir Expression Blend et charger le projet exemple :

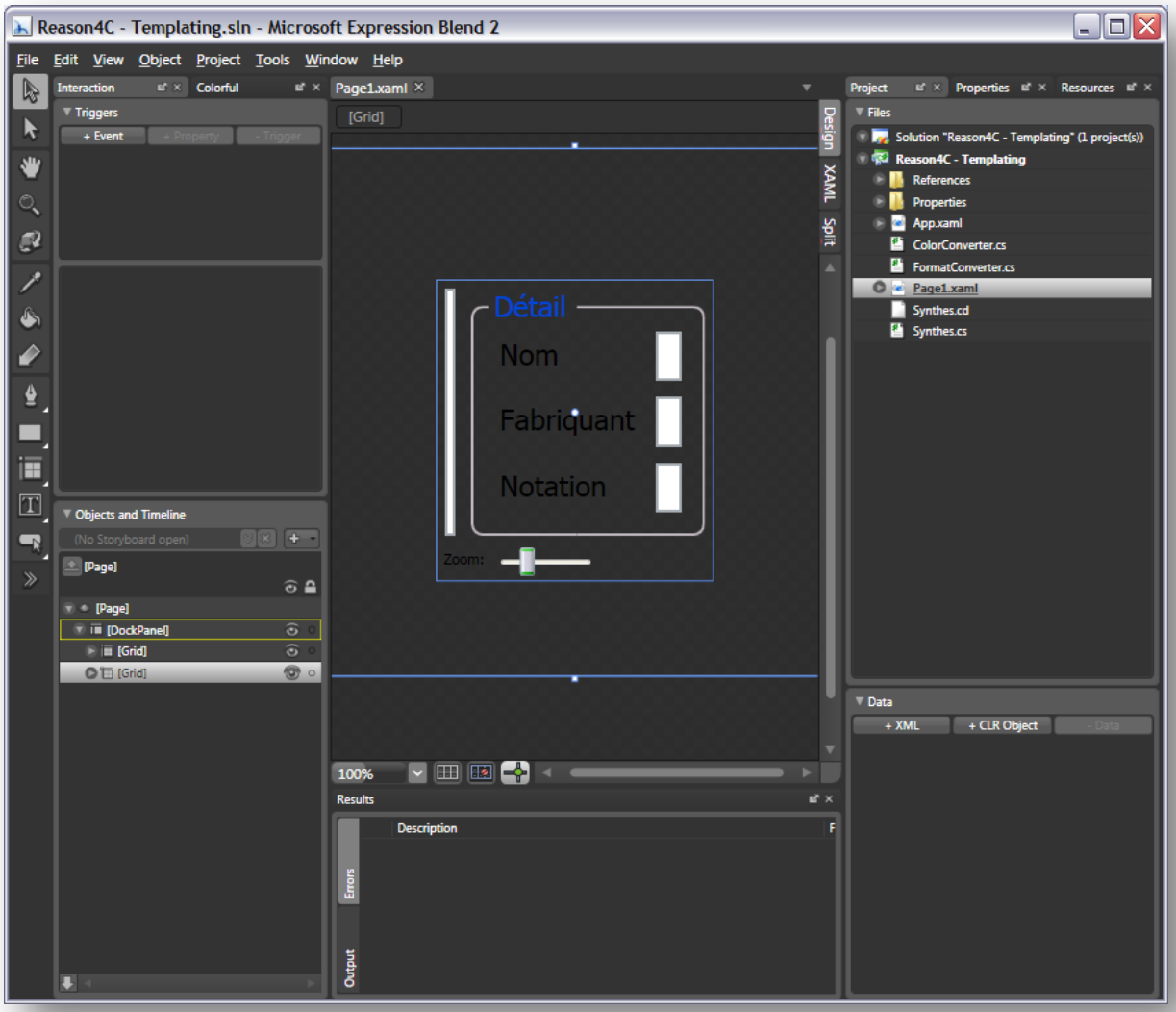


Figure 9 - projet Wpf10-4c

La page semble un peu compressée, ce n'est rien, en l'étirant un peu nous retrouverons rapidement le même affichage que sous VS.

Les mécanismes sous Blend sont assez différents de ceux du code Xaml tapé à la main. Par exemple, et pour simplifier le Data Binding, Blend permet de créer des sources de données à partir de classes contenue dans le projet (le cadre « Data » en bas à droite sur la capture ci-dessus). Ensuite il suffit d'un clic droit sur la listbox pour définir une liaison à ces données. Le `DataTemplate` peut être créé immédiatement en quelques clics :

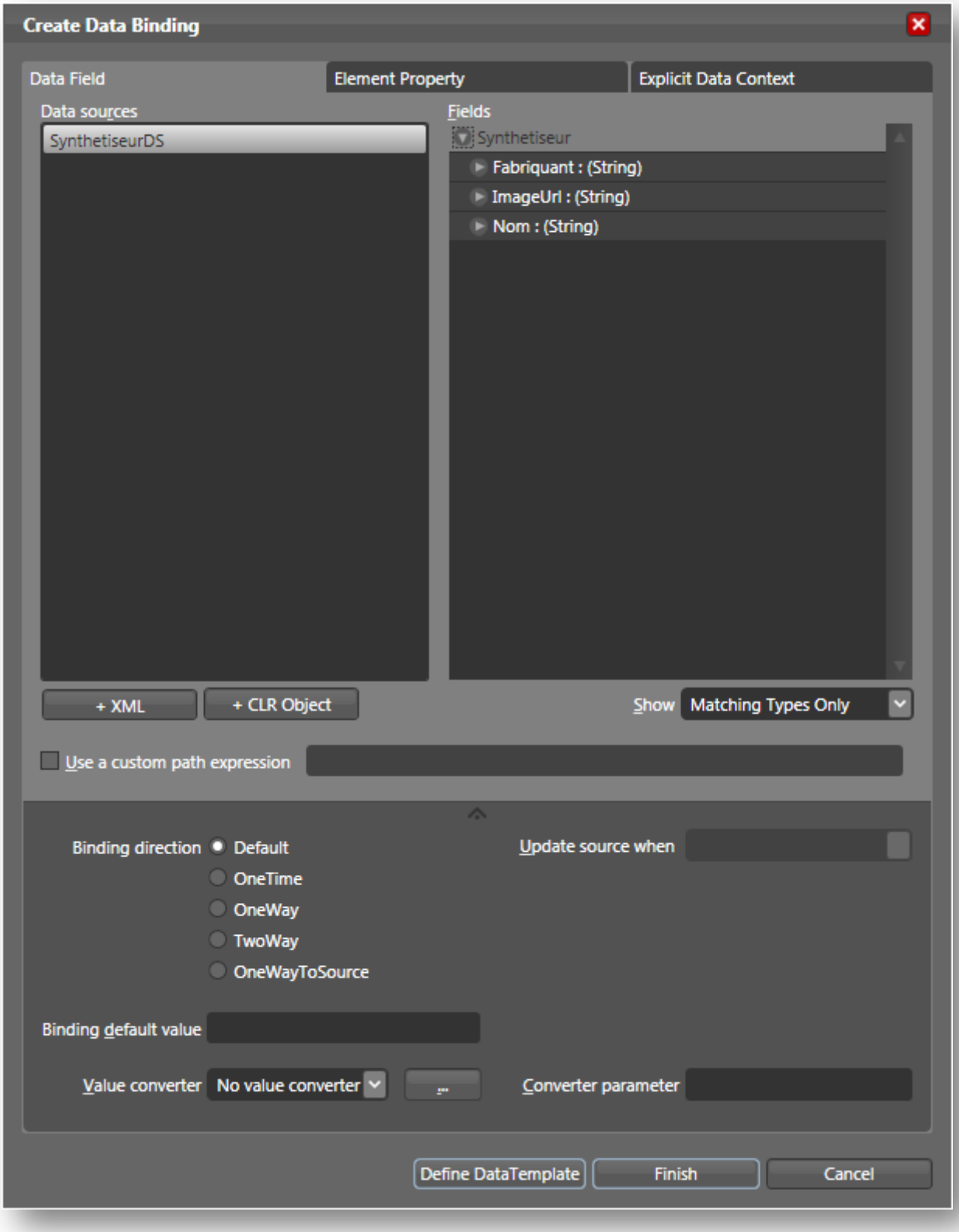


Figure 10 - La création d'un Data Binding sous Blend

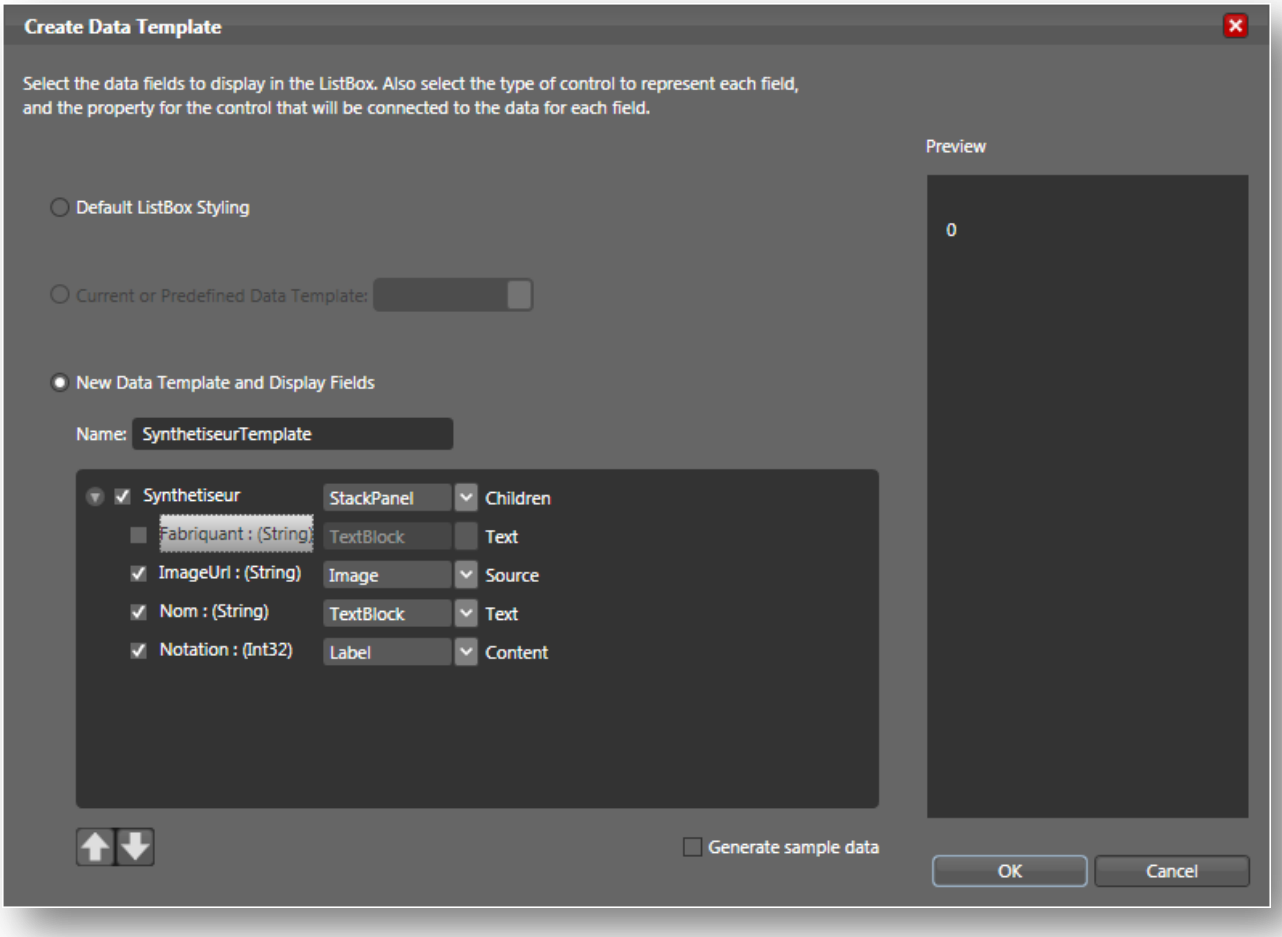


Figure 11 - la création d'un Data Template sous Blend

La capture ci-dessus montre la définition rapide d'un `DataTemplate` sous Blend. On choisit les champs de l'objet à afficher et le type de composant à utiliser (choix qu'on pourra modifier et affiner ensuite).

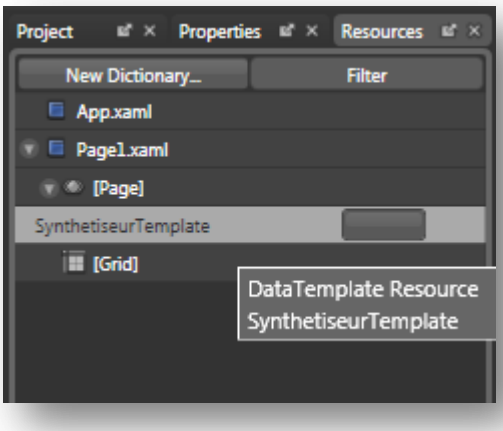


Figure 12 - L'onglet Ressources sous Blend

Ci-dessus nous voyons le `DataTemplate` « `SynthetiseurTemplate` » qui apparaît dans les ressources de la page. Nous pourrions le déplacer au niveau projet par exemple pour le réutiliser dans plusieurs pages. En cliquant sur le template nous entrons en mode édition de celui-ci, c'est là que vous pouvons visuellement faire toutes les adaptations possibles :

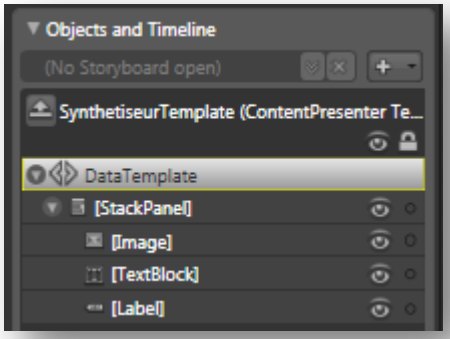


Figure 13 - Le panneau Objets et Animations de Blend

Le cadre « *objects and Timeline* » montre l'arborescence des objets de notre template dont le nom s'affiche en haut. Il ne nous reste plus qu'à mettre tout cela en forme selon nos vœux.

En utilisant Blend **nous travaillons de façon plus structurée et plus visuelle**, le `DataTemplate` est nommé, il est placé dans des ressources facilement accessibles et il est modifiable d'un simple clic en utilisant un mode d'édition totalement visuel. Même s'il s'agit d'un logiciel de plus à acquérir (il n'y a pas de « Blend Express » pour l'instant), il ne faut pas se voiler la face : **pas de développement sérieux WPF ou Silverlight sans Expression Blend**. N'utiliser que Visual Studio pour ce type d'application ce serait aussi productif que de vouloir développer tout un site ASP.NET avec le bloc-notes.

Maintenant que nous avons recréé proprement le `DataTemplate` et la connexion aux données sous Blend, revenons à nos moutons : le templating d'un contrôle visuel, en l'occurrence le `ProgressBar`. Pour ce faire, rien de plus simple : nous ouvrons en édition le `DataTemplate` et nous faisons un clic droit sur le composant en question. Blend nous offre plusieurs options dont la modification / création de son template. Nous pourrons aussi créer un style qui engloberait ce template mais ici nous irons droit au but :

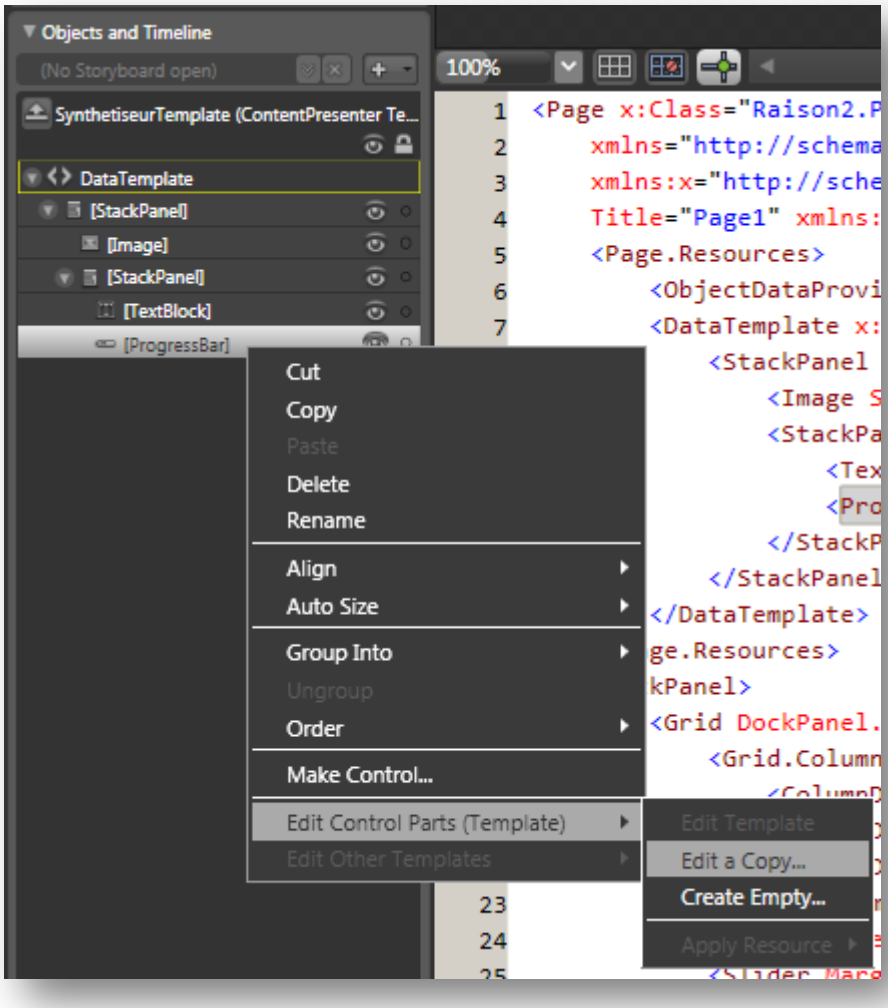


Figure 14 -Création d'un template (mode copie)

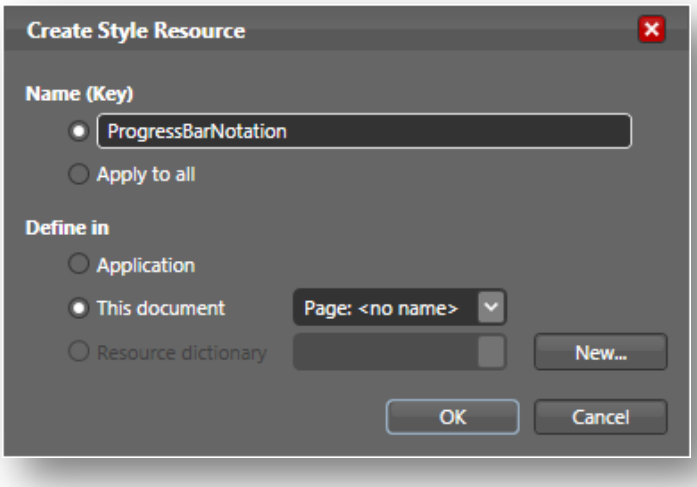
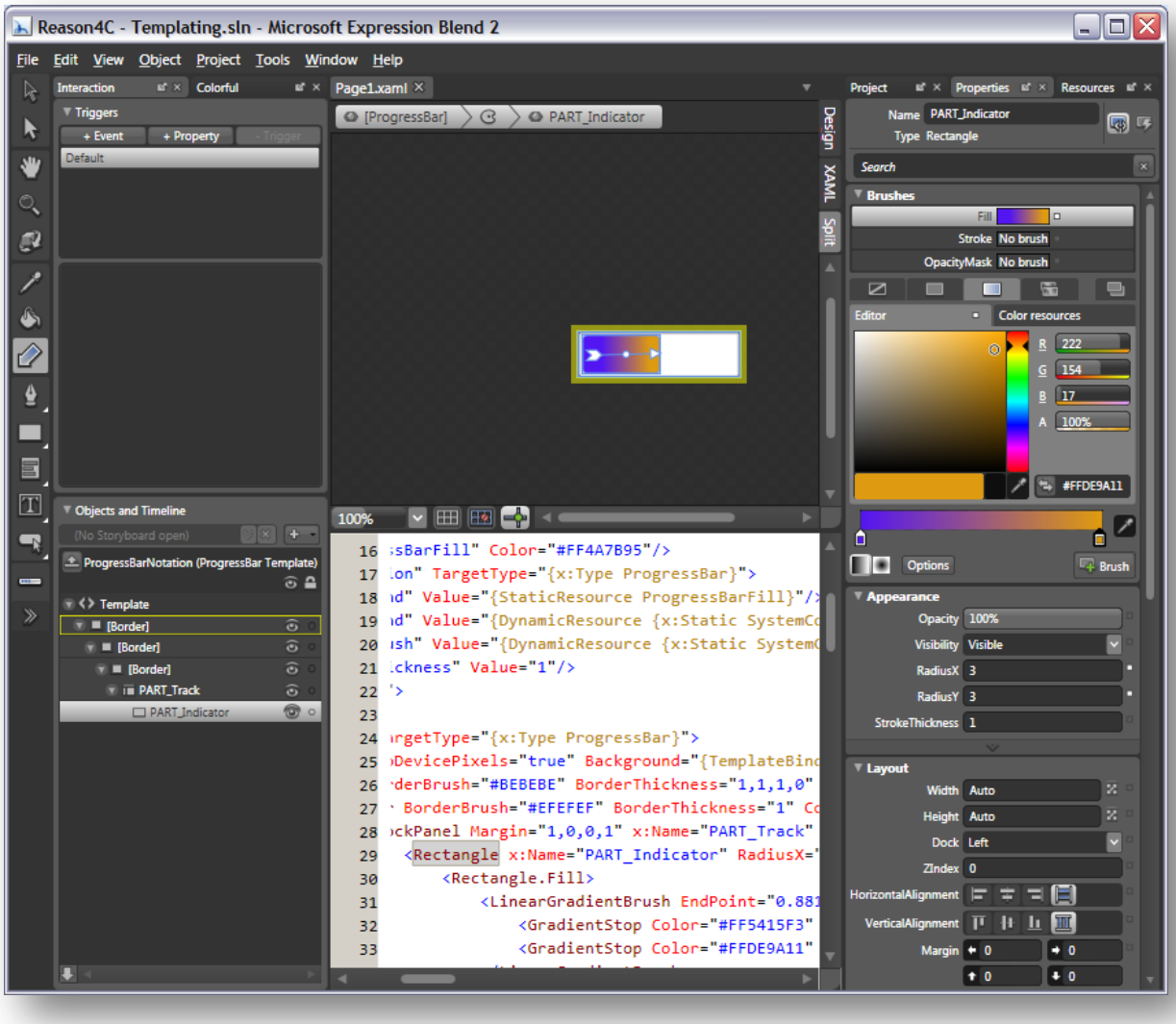


Figure 15 - Création d'une ressource de style

Les deux captures ci-dessus :

- Localisation du composant à templater dans l'arborescence du `DataTemplate`, clic droit et modification d'une copie du template (les composants possèdent un template par défaut qui dépend de l'OS sur lequel on travaille, XP ou Vista). On pourrait partir d'un template totalement vide, ce qui est parfois plus rapide.
- Création d'une nouvelle ressource de style dont le nom sera `ProgressBarNotation`. Elle sera défini dans les ressources du document en cours. On voit qu'il serait possible de placer la ressource au niveau de l'application ou dans un dictionnaire facilement partageable entre plusieurs applications.



La capture ci-dessus montre le templating du `ProgressBar` en cours de travail. Pour cette démonstration nous avons fait au plus simple en remplaçant l'image de remplissage typique des barres de progression de XP par un dégradé plus chatoyant. Nous avons aussi arrondi les coins du rectangle dégradé pour plus de douceur visuelle. (vous déduirez donc que cet article a été créé sous XP et non Vista… La raison est toute bête sur ma machine en dual boot XP/Vista j'ai commencé l'article un jour où j'étais sous XP et non Vista, et je l'ai terminé en restant sous XP ! Aucun rejet de Vista que je préfère de loin à XP ☺).

Figure 16 - Affichage de la ProgressBar relookée par templating

Visuellement le résultat est plus agréable non ? Bon je n'ai pas fait dans la dentelle non plus et je n'ai pas été déranger l'infographiste de service pour produire une interface ultra léchée. Mais justement c'est cela qui compte : avec les bons outils, sous WPF même un informaticien pas doué pour le dessin peut produire une application riche. De plus, la séparation nette entre interface et code permettra, si cela est nécessaire, de faire travailler un infographiste sur le visuel sans remettre en cause une seule ligne de programmation.

Le templating des contrôles est bien la 4ème bonne raison de choisir WPF !

Raison 5 : Les triggers et le VisualStateManager

La gestion des événements est l'un des piliers de la programmation sous Windows et autres environnements du même type. Le modèle est d'ailleurs appelé *programmation événementielle*. Le principe est simple : les objets peuvent déclencher des événements auxquels d'autres objets peuvent « s'abonner » et être ainsi directement prévenus. Le plus célèbre, l'événement `Click` d'un bouton permet à une fiche d'implémenter une méthode qui sera appelée directement lorsque l'utilisateur cliquera sur le bouton. Procédé appelé « délégation » (un objet délègue l'exécution d'une action à un autre objet).

C'est simple, pratique, mais un peu limitatif. Le Framework .NET va ajouter au modèle Win32 classique de type MFC ou VCL la possibilité que plusieurs objets écoutent le même événement. La design pattern Observer devient native ce qui autorise bien plus de souplesse.

Sous WPF ce principe est bien entendu repris mais il est amplifié. La notion de **Trigger**, si elle est proche de celle des événements, complète cette dernière. Il devient ainsi possible de déclencher des actions non seulement en réponse à des événements classiques mais aussi à **des changements d'état** des objets.

Cet ajout a été rendu indispensable pour simplifier la création des interfaces graphiques riches. Par exemple un bouton peut déclencher une animation qui le rendra plus lumineux dans le cas où la souris le survole, ou bien l'opacité d'un élément visuel peut être modifiée en fonction de l'importance de l'information qu'il véhicule. Si certains de ces comportements (et bien d'autres impossibles à lister) peuvent se résoudre avec des événements « classiques », d'autres nécessitent la prise en compte de valeurs particulières ou d'états singuliers. Et c'est là que les triggers prennent leur intérêt. Pour rappel : les états d'un objet sont souvent représentés par des énumérations ou des propriétés booléennes (`IsEnabled`, `IsMouseOver`, etc.) .

Revenons à l'exemple de code que nous faisons évoluer au fil de cet article. Il existe un état particulier que nous aimerions mettre en évidence, celui d'une fiche synthétiseur lorsque sa notation est égale à 5, le maximum de points autorisé.

Encore une fois sous WPF pour répondre à ce type de situation nul besoin de dégainer son compilateur favori. Xaml et sa puissance descriptive permettent de trouver des solutions beaucoup plus originales et moins coûteuses en lignes de code. On notera que l'état qui nous intéresse n'est pas représenté directement dans l'objet cible, ni énumération ni booléen spécifique, pourtant WPF va permettre de traiter ce cas particulier.

Pour faire très simple et dans un premier temps avec juste quelques lignes de Xaml, nous allons faire passer en gras le nom du synthétiseur lorsque la valeur de sa notation est égale à 5 :

```
<DataTemplate x:Key="SynthetiseurTemplate">
   <DataTemplate.Triggers>
      <DataTrigger Binding="{Binding Notation}" Value="5">
         <Setter TargetName="tbNom" Property="TextBlock.FontWeight" Value="Bold"/>
      </DataTrigger>
   </DataTemplate.Triggers>
         <StackPanel Orientation="Horizontal" Margin="0,5,0,5" ToolTip="{Binding Path=Fabriquant}">
                  <Image Source="{Binding Path=ImageUrl}" Width="50" Margin="0,0,5,0"/>
                  <StackPanel>
                           <TextBlock x:Name="tbNom" Text="{Binding Path=Nom}"/>
                           <ProgressBar Maximum="5" Value ="{Binding Path=Notation}"
Style="{DynamicResource ProgressBarNotation}" Width="100" Height="20"/>
                  </StackPanel>
         </StackPanel>
</DataTemplate>
```

Le code ci-dessus reprend le `DataTemplate` créé sous Expression Blend. Nous avons ajouté une section `DataTemplate.Trigger` qui contient un `DataTrigger` lié au champ `Notation` du `DataContext` et qui se déclenchera lorsque la valeur de ce champ sera exactement égale à `5`.

Ensuite nous décrivons ce qui doit arriver lorsque cette condition se déclenche. Pour cela nous utilisons une section « `Setter` ». Comme leur nom l'indique de telles sections permettent de modifier la valeur d'une propriété d'un objet. Ici nous modifions l'objet `tbNom` (qui est le

`TextBlock` contenant le nom du synthétiseur dans la liste, on le voit défini un peu plus bas). Dans cet objet nous ciblons la propriété `FontWeight`, le poids de la fonte et nous passons sa valeur à `Bold`.

Voilà, deux ou trois lignes de Xaml permettent, toujours sans code C# ou autre, de modifier le comportement visuel de l'application. C'est simple et descriptif, peu de chances de faire une « erreur de programmation ». Au pire l'effet rendu ne sera pas celui escompté mais nous ne risquons pas de bouleverser le code applicatif qui n'est absolument pas concerné ici.

Visuellement nous obtenons l'effet suivant :

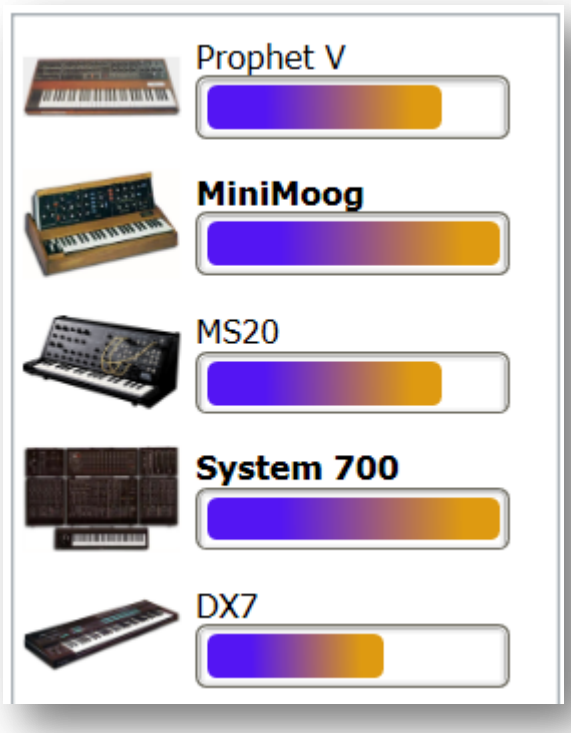


Figure 17 - Projet Wpf10-5

Les deux synthétiseurs notés 5 dans la liste ont leur nom qui apparaît bien en gras !

Pour l'instant nous nous sommes intéressés à des embellissements visuels, l'aspect d'un logiciel est aujourd'hui une chose essentielle et il est naturel de lui donner tant d'importance. Mais WPF et ses nombreuses possibilités, comme les triggers, servent aussi à gérer des situations plus fonctionnelles.

Dans notre exemple les boîtes de saisie à droite de la liste servent à modifier le contenu de la fiche sélectionnée. Mais à l'entrée de l'application aucune fiche n'est sélectionnée et il reste malgré tout possible de saisir dans ces boîtes. Notre application exemple ne supporte pas la création d'objets pour l'instant et il nous faut interdire cette situation.

Ici encore nul besoin de code en C#, un trigger placé sur l'objet de groupe contenant les `TextBox` permettra de les rendre inopérant (*disabled*).
Pour ce faire nous localisons la `GroupBox` « `Détail` » et nous lui ajoutons les lignes suivantes :

```
<GroupBox.Style>
    <Style>
        <Style.Triggers>
            <DataTrigger Binding="{Binding ElementName=lbSynth, Path=SelectedIndex}" Value="-1">
                <Setter Property="GroupBox.IsEnabled" Value="False"/>
            </DataTrigger>
        </Style.Triggers>
    </Style>
</GroupBox.Style>
```

Le principe reste le même, au sein d'une section de style de la `GroupBox` nous ajoutons une section `Triggers` qui contient un `DataTrigger`. Ce dernier est lié à la listbox (`ElementName`) et plus spécifiquement à sa propriété `SelectedIndex`. C'est lorsque la valeur de cette propriété est égale à `-1` que le trigger se déclenchera (aucun élément sélectionné dans la liste).

Il suffit alors de préciser ce que nous désirons faire une fois le trigger déclenché. Ici nous passons simplement à `False` la propriété `IsEnabled` du `GroupBox`.

Visuellement nous obtenons ceci au lancement de l'application :

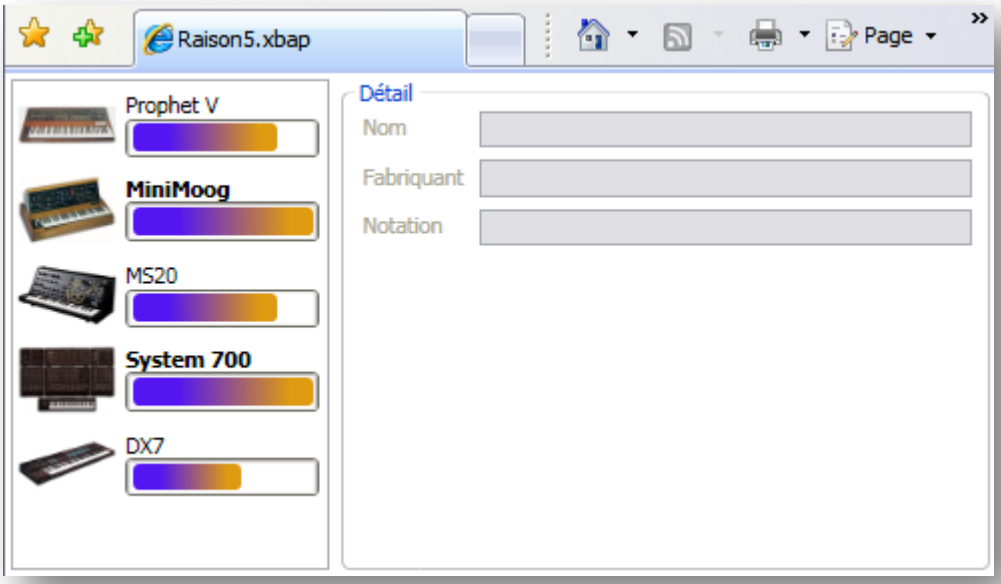


Figure 18 - Le cadre "détail" n'est pas actif

Il n'est plus possible de saisir du texte dans le cadre « Détail » si aucune ligne n'est sélectionnée dans la liste.

La gestion des triggers est bien une 5ème bonne raison de choisir WPF !

Mais revenons un instant sur le titre de cette section « Triggers et VisualStateManager ». Si vous voyez maintenant ce que sont les triggers, qu'est-ce donc que le « `VisualStateManager` » ?

Originellement cette fonction n'était disponible que sous Silverlight, mais en raison de sa popularité méritée le concept a été porté sous WPF.

Le problème qui est résolu est celui des différents changements d'état d'un objet et surtout de leurs transitions. Par exemple un bouton bien développé devra réagir au `MouseEnter`, puis au `MouseLeave` (pour reprendre son aspect original), de même il aura certainement un affichage légèrement différent durant un `Click`, etc. Tout cela peut se programmer sous WPF avec des événements ou des triggers. Blend facilite énormément le travail et permet de créer des animations (timelines) qui seront jouées en réponse à certains déclencheurs. Tout cela peut malgré tout devenir un peu compliqué à gérer.

Imaginons que le bouton devient vert quand la souris le survole et qu'il redevient gris une fois la souris sortie de la zone du bouton. Mais il doit devenir rouge durant le `Click`. En fin de `Click` de quel couleur est-il ? En gérant le `MouseUp` et le `MouseDown` séparément plutôt que le `Click` on peut réaffecter le vert en fin de `Click` (si on vient de finir de cliquer on est toujours au dessus du bouton donc il est vert…). Vous voyez que sur des actions graphiques aussi simples les choses s'embrouillent assez vite. Il faut beaucoup de méthode à l'infographiste pour planifier tous les états visuels sans s'emmêler… les pinceaux virtuels !

C'est la que le `VisualStateManager` de Silverlight, et désormais de WPF, vient au secours de l'artiste. Comme son nom l'indique ce système est un « gestionnaire des états visuels ». Il regroupe tous les groupes d'états qu'un composant peut présenter et permet d'indiquer pour chaque transition l'effet ou les effets à appliquer. C'est le `VisualStateManager` qui s'occupe ensuite de basculer d'un état visuel à l'autre en respectant la cohérence visuelle et celle des objets impliqués. On peut même indiquer une durée pour ces transitions, créant ainsi des animations automatiques sans programmer de timelines.

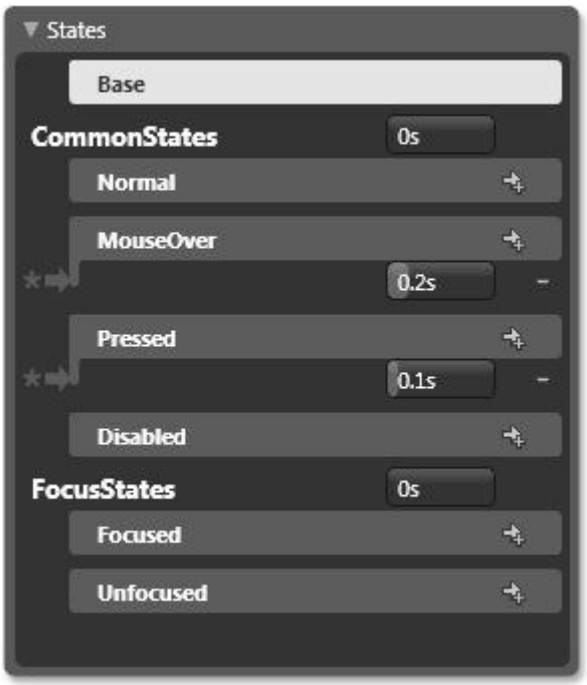


Figure 19 - Le VisualStateManager sous Blend

Faire une démonstration sous Blend prend quelques instants car tout ceci est purement visuel et interactif, mais dans un article tout de suite cela devient plus lourd et moins attractif. Je

n'entrerai donc pas les détails du `VisualStateManager` mais sachez qu'il simplifie grandement la notion de trigger.

Vous voyez, tout cela méritait largement d'être une raison de plus de choisir WPF !

Raison 6 : Les Styles

Il est vrai que nous les avons évoqués, mais nous ne leur avons pas consacré la place qu'ils méritent. Les styles sont une des fonctionnalités les plus importantes de WPF et Silverlight. En effet ce sont eux qui permettent, à l'instar des feuilles de style CSS, de créer un *look and feel* complet pour une application facilement réutilisable dans une autre application (à noter que l'équivalent stricte d'une feuille de style CSS sous WPF s'appelle un dictionnaire, fichier contenant des définitions de ressources, dont les styles).

Quelle différence y-a-t-il entre les styles les templates ?

Les `DataTemplate` permettent de formater des données à afficher, les templates de contrôles permettent de modifier l'aspect graphique de tout contrôle.

Toutefois, lorsque vous créez un template pour un contrôle certaines propriétés ne doivent pas être fixées. Non par limitation de WPF mais pour des raisons liées aux bonnes pratiques sous cet environnement.

Reprenons l'exemple précédent et regardons ce que nous avons fait dans le template de la `ProgressBar`. Hélas, en figeant dans le template le dégradé qui remplit le contrôle nous avons « cassé » en quelque sorte « la chaîne des propriétés ». La plupart des composants possèdent par exemple une propriété `Background` ou `Foreground` pour fixer leur couleur d'arrière et avant plan. Si le template fige de telles couleurs l'utilisateur du composant n'en verra pas moins dans la fenêtre de propriété les fameux `Background` et `Foreground`. Tout naturellement, si la couleur que nous avons choisi ne lui convient pas il voudra, et c'est légitime, la changer en modifiant la propriété concernée. Malheureusement la propriété est « débranchée » elle « n'arrive nulle part » puisque nous ne nous en servons plus…

Créer des templates de cette façon n'est donc pas une bonne approche. Vous allez dire, oui mais comment faire si je veux justement, en dehors de l'aspect et de la forme, fixer un thème (couleurs, fontes…) ?

C'est justement à cela que servent les styles (entre autres). Le template se contente de figer la forme, l'aspect graphique, et il met en place une liaison entre les propriétés du composant (comme la couleur de fond, la marge, etc) et les éléments graphiques qui permettent de rendre compte de ces propriétés. De fait, c'est à l'intérieur d'un style et non d'un template qu'on pourra modifier ces valeurs, par souci de cohérence. Un style englobe donc des valeurs pour les propriétés d'un composant alors que le template prépare le terrain pour recevoir ces valeurs (par exemple en contenant un rectangle dont la couleur de remplissage sera connectée à la propriété `Background` du contrôle hôte par un Data Binding, ce qui se fait très simplement sous Blend). De plus, un style peut parfaitement définir un template auquel il fixera des valeurs.

Les styles WPF se placent ainsi un cran plus haut dans la hiérarchie que les templates.

Pour donner corps à tout cela il est tant de donner un look à notre application !

Pour cela nous allons créer un nouveau dictionnaire de ressources, c'est là seule chose que nous allons ajouter au projet, sans rien modifier du code C# existant (le peu qu'il y en a d'ailleurs).

Sous Blend cela se fait simplement en cliquant sur « *New Dictionary* » dans l'onglet *Resources* en haut à droite de la page. Une fois le nom saisi Blend ajoute le fichier au projet et enregistre celui-ci automatiquement au niveau de `App.xaml`, un des fichiers clé de toute application WPF.

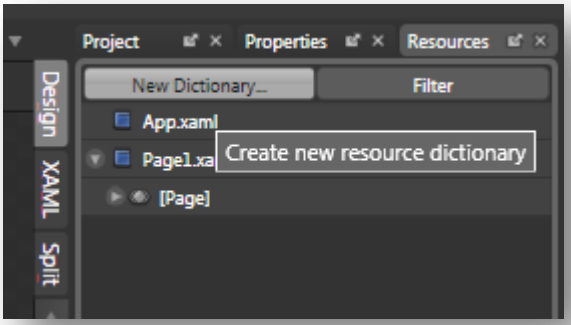


Figure 20 - La création d'un nouveau dictionnaire de ressources

Sous VS il suffit d'ajouter un nouveau fichier, par exemple `MesStyles.xaml`, au projet (ajouter un nouvel item / WPF / fichier de ressource). Toutefois VS n'intègre pas automatiquement le dictionnaire à `App.xaml`, il faut donc penser à le faire manuellement code le montre le code ci-dessous :

```
<Application x:Class="Raison6.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="Page1.xaml">
   <Application.Resources>

         <ResourceDictionary>
                  <ResourceDictionary.MergedDictionaries>
                           <ResourceDictionary Source="MesStyles.xaml"/>
                  </ResourceDictionary.MergedDictionaries>
         </ResourceDictionary>

   </Application.Resources>
</Application>
```

Maintenant que nous disposons d'un dictionnaire de ressources qui est visible au niveau de l'application nous allons pouvoir définir des styles. Commençons par quelques brosses fixant le nuancier à utiliser. Procéder de cette façon est une bonne pratique, on ne fixe jamais les couleurs dans un style ou un modèle, mieux vaut définir des brosses auxquelles les styles et templates pourront faire référence. Il sera bien plus aisé de modifier la charte couleur ultérieurement !

Voici par exemple comment est définie une brosse :

```
<LinearGradientBrush x:Key="MaBrosse" EndPoint="0.081,0.011" StartPoint="0.919,0.989">
    <GradientStop Color="#FF000000" Offset="0"/>
    <GradientStop Color="#FF949494" Offset="1"/>
</LinearGradientBrush>
```

Il s'agit ici d'une brosse de type dégradé utilisant deux couleurs. Bien entendu définir les valeurs à la main sous Visual Studio n'est vraiment pas pratique, c'est là que Expression Blend d'une aide précieuse se transforme tout simplement en un passage obligé.

Regardons maintenant la définition d'un style :

```
<Style TargetType="{x:Type Raison6:Page1}">
    <Setter Property="FontSize" Value="16" />
    <Setter Property="Background" Value="{StaticResource _brshHorizon}"/>
</Style>
```

Ici un nouveau style est créé avec pour cible la `Page1` de l'application. La taille de la fonte et la couleur de fond sont fixés par des balises `Setter` que nous avons déjà vues. Cette simple entrée dans le dictionnaire change les deux propriétés concernées sans avoir à connaître ni posséder le code source applicatif. Si le principe est connu depuis les feuilles de style CSS dans le monde du Web l'application de ce principe aux applications desktop est nouveau. On notera que la définition d'un style peut cibler un élément particulier (ici la `Page1` du namespace `Raison6`) ou bien un type comme `TextBlock` ou bien rien du tout (tout élément supportant les propriétés indiquées sera modifié).

Une fois une poignée d'autres styles définis notre application exemple (à laquelle la suppression et la création d'items ont été ajoutés) ressemble à cela :

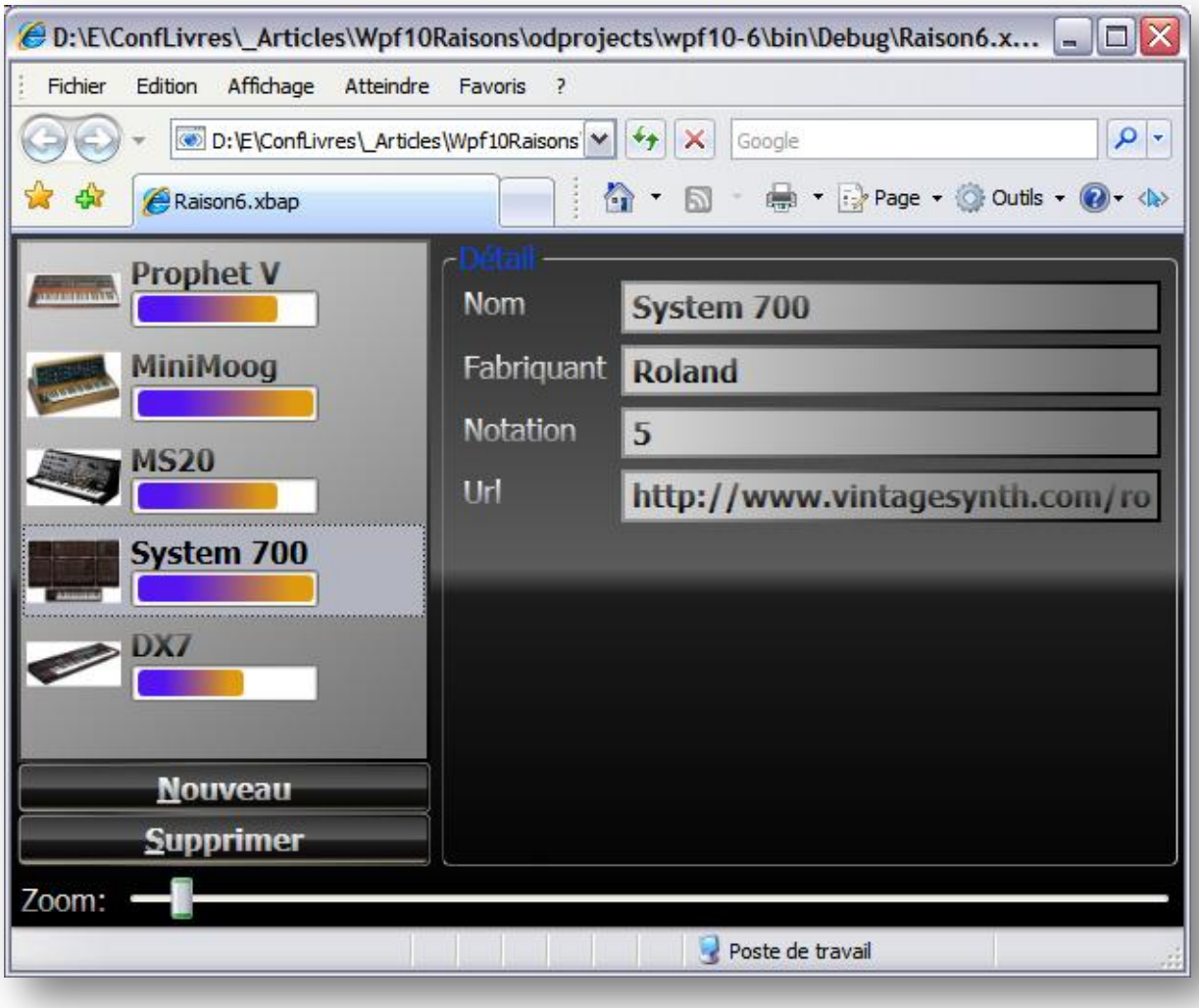


Figure 21 - Projet Wpf10-6

Pas mal pour du zéro code, zéro librairie tierce et quelques lignes de Xaml non ?

La gestion des styles est bien une 6ème bonne raison de préférer WPF !

Raison 7 : Les validations

Valider les données et produire un feedback intelligible pour l'utilisateur est l'un des points clé d'une interface ergonomique réussie. La faiblesse des modèles Win32 sur ce point est flagrante et il aura fallu attendre ASP.NET ou les Windows Forms pour que des mécanismes de validation cohérents soient implémentés de façon native.

Le Data Binding de WPF, comme nous l'avons vu, est d'une grande souplesse, on place le nom d'une propriété dans une accolade de `Binding` et l'affaire est jouée. Comment se passe alors les validations dans un modèle si tolérant ?

Par défaut il ne se passe rien… En effet l'interface étant totalement libre tout affichage intempestif de la part du Framework dénoterait avec l'allure générale de l'application en cours qui n'est plus prévisible comme sous Windows Forms. Avec ce dernier modèle, si le

Framework ouvre une boîte de dialogue signalant une exception cet affichage peut passer inaperçu (d'un point de vue aspect visuel), une fenêtre rectangulaire grise avec un bouton rectangulaire gris marqué « Ok » ne dépare pas avec une application où tout est aussi rectangulaire et gris…

En revanche sous WPF impossible de tromper l'utilisateur. Tout affichage doit se faire en cohérence avec le look & feel global de l'application qui n'est pas prévisible. Donc par défaut WPF est « oublieux », c'est-à-dire que les erreurs de saisie par exemple sont purement et simplement ignorées.

Dans l'application exemple qui nous suit depuis le début de cet article nous disposons de plusieurs champs de saisie, tous sont de type `string` mais la notation est de type `integer`. Que se passe-t-il si nous tapons du texte ? Rien. Le texte tapé est bien affiché mais en revanche l'objet sous-jacent n'est pas mis à jour (bien naturellement puisque la conversion a échoué).

Si vous regardez la fenêtre de sortie de Visual Studio vous verrez tout de même un message du type :

```
System.Windows.Data Error: 7 : ConvertBack cannot convert value '4m' (type 'String').
BindingExpression:Path=Notation; DataItem='Synthetiseur' (HashCode=1120636206); target element
is 'TextBox' (Name='tbNotation'); target property is 'Text' (type 'String')
FormatException:'System.FormatException: Le format de la chaîne d'entrée est incorrect.
   à System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number,
NumberFormatInfo info, Boolean parseDecimal)
   à System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   à System.String.System.IConvertible.ToInt32(IFormatProvider provider)
   à System.Convert.ChangeType(Object value, Type conversionType, IFormatProvider provider)
   à MS.Internal.Data.SystemConvertConverter.ConvertBack(Object o, Type type, Object
parameter, CultureInfo culture)
   à System.Windows.Data.BindingExpression.ConvertBackHelper(IValueConverter converter, Object
value, Type sourceType, Object parameter, CultureInfo culture)'
```

Ici un « `m` » minuscule a été tapé à la suite du « `4` » se trouvant déjà dans la zone. Le message est clair, le Framework nous indique qu'une valeur de type `string` n'a pu être convertie pour la propriété `Notation` de la classe `Synthethiseur` et que ce problème s'est posé dans le `TextBox tbNotation`. S'en suit le dévidage de la pile d'appel.

Le Framework fait donc bien son travail et de façon très précise (combien nous aurions rêvé d'avoir des traces aussi détaillées sous les environnements de développement Win32 !).

Le décor est donc déjà en place, il ne manque que le feedback visuel.

Pour l'instant le `TextBox` en question est défini de la façon suivante dans la page Xaml :

```
<TextBox Grid.Column="1" Grid.Row="2" Margin="3" Text="{Binding Path=Notation,
UpdateSourceTrigger=PropertyChanged}" Name="tbNotation" />
```

Une première réécriture de la balise permet de dégager les différentes sections pour mieux intervenir :

```
<TextBox Grid.Column="1" Grid.Row="2" Margin="3"  Name="tbNotation">
    <TextBox.Text>
        <Binding Path="Notation" UpdateSourceTrigger="PropertyChanged" />
    </TextBox.Text>
</TextBox>
```

Ici nous n'avons fait que remettre en forme la balise originale, le comportement restant identique. En détaillant la balise de `Binding` nous pouvons maintenant agir sur les règles de validation. Dans un premier temps nous allons ajouter un comportement par défaut :

```
<TextBox Grid.Column="1" Grid.Row="2" Margin="3"  Name="tbNotation">
    <TextBox.Text>
        <Binding Path="Notation" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <ExceptionValidationRule/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Le Data Binding supporte la notion de règle de validation, ici nous utilisons simplement la règle la plus simple gérée automatiquement par WPF. Si une erreur apparaît elle sera signalée à l'utilisateur par un cadre rouge dessiné autour du champ fautif :

Figure 22 - Comportement par défaut d'une règle de validation

L'image ci-dessus montre la zone « `Notation` » durant une erreur de saisie. La lettre « `p` » a été tapée à la suite du « `5` » ce qui déclenche une erreur de conversion. WPF entoure automatiquement la zone d'un filet rouge.

Cela est simple, parfois suffisant, mais n'oublions pas que si nous utilisons WPF c'est pour sa souplesse et la richesse visuelle des applications qu'on peut concevoir. On doit donc pouvoir faire mieux, toujours sans code C#.

Avant d'aller plus loin revenons sur ce qu'implique, techniquement, le code Xaml de validation que nous avons ajouté.

Le fait d'avoir spécifié une règle de validation met en route une série de mécanismes dont la chaîne est, en simplifiant : si une règle échoue le Framework place la valeur `True` dans la propriété `Validation.HasError` qui a été attachée au contrôle. Le basculement de cette valeur entraîne l'affichage du template `ErrorTemplate` associé au contrôle. Ce template est spécifié dans `Validation.ErrorTemplate`. Par défaut le template fourni s'occupe de mettre un filet rouge autour du champ.

Rien de magique donc, une chaîne logique avec beaucoup de choses « par défaut » pour simplifier l'utilisation de WPF, mais aussi beaucoup de points sur lesquels le développeur, s'il le souhaite, peut intervenir pour personnaliser le comportement global.

De fait nous pouvons parfaitement fournir notre propre template d'erreur pour le faire mieux correspondre au look & feel de notre application. A la différence de Windows Forms et des systèmes Win32 qui obligent le développeur sans cesse à « lutter » contre les mécanismes codés en dur pour obtenir un résultat moins terne, **WPF a été totalement conçu pour supporter la créativité**. On ne lutte pas contre WPF pour créer une interface riche, ergonomique et créative, c'est bien au contraire un allié précieux qui fournit toute l'infrastructure nécessaire pour atteindre ce but.

Cela est vraiment essentiel et n'est pas juste anecdotique. Sous les autres environnements on doit sans cesse « ruser », sous classer des composants, jongler avec les messages Windows, etc, tout cela pour rendre une interface moins terne, moins passe-partout. La tâche est d'ailleurs si difficile et réclame un talent de programmeur tellement au dessus de la moyenne que les bibliothèques tierces sont nombreuses. D'ailleurs cet énorme handicap a été, il fut un temps pas si lointain, un argument de vente ! Paradoxalement le fait, par exemple, que la communauté Delphi se trouva très active à l'apogée de ce langage consistait en soi un argument commercial : vous allez trouver facilement plein de composants gratuits et payants ! (ce raisonnement était le même pour les OCX sous VB ou VC++).
Chic ! Plein de code tiers programmé de façon non homogène, peu ou pas évolutif, souvent pas très bien ficelé et impossible à maintenir, que je vais pouvoir mélanger joyeusement à mon propre code !

Dis comme ça aujourd'hui avec le recul ça fait peur et ça frise le ridicule non ? Les temps changent…

WPF se moque de savoir si vous allez trouver sur le Web un composant « jauge » au look sympa ou une grille capable d'afficher une ligne sur deux avec une couleur différente. Il s'en moque parce qu'il sait le faire, vous n'avez qu'à ajouter les quelques lignes de Xaml qu'il faut … Ce qui n'interdit pas d'utiliser des librairies tierces, mais vous n'y êtes plus obligé, ce qui fait une grande différence.

Revenons à la validation de notre champ `Notation`.

Voici un template de contrôle très simple que nous ajoutons soit aux ressources locales de la page, soit plus généralement à celles de l'application (`App.xaml`) ou à un dictionnaire réutilisable :

```
<ControlTemplate x:Key="validationTemplate">
    <DockPanel>
      <AdornedElementPlaceholder/>
      <TextBlock Foreground="Red" FontWeight="Bold" FontSize="20">!</TextBlock>
    </DockPanel>
 </ControlTemplate>
```

Ce template permet de définir l'aspect visuel de tout contrôle auquel il sera lié. Le niveau de redéfinition offre une très grande liberté. Le contrôle décoré est appelé

`AdornedElementPlaceHolder`, ce qui permet de le manipuler sans le connaître. Dans le code Xaml ci-dessus cet emplacement réservé pour le contrôle décoré est enchâssé dans un `DockPanel` qui contient à la suite un morceau de texte affichant un point d'exclamation rouge.

Visuellement le résultat est moins probant que le filet rouge par défaut mais le but ici n'est pas de faire une leçon d'infographie, juste de comprendre la richesse de WPF…

Pour accrocher ce template au `TextBox` Notation, et ce uniquement si une erreur de validation se produit, nous modifions sa balise pour qu'elle devienne la suivante :

```
<TextBox Grid.Column="1" Grid.Row="2" Margin="3"  Name="tbNotation"
                         Validation.ErrorTemplate="{StaticResource validationTemplate}">
    <TextBox.Text>
        <Binding Path="Notation" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <ExceptionValidationRule/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Le résultat visuel devient le suivant en cas d'erreur de saisie :

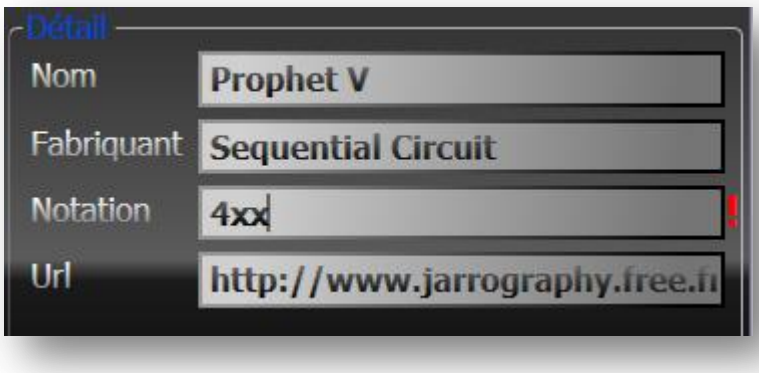


Figure 23 - L'effet du template de validation personnalisé

La gestion d'un template pour les erreurs de validation est une possibilité parmi d'autres. Nous pouvons aussi créer un style avec des triggers pour définir l'aspect des `TextBox` de notre application (avec des animations en plus si nécessaire) :

```
<Style x:Key="highlightValidationError" >
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="True">
            <Setter Property="Control.Background" Value="Pink" />
        </Trigger>
    </Style.Triggers>
</Style>
```

Ce qui donnera un effet visuel comme suit :

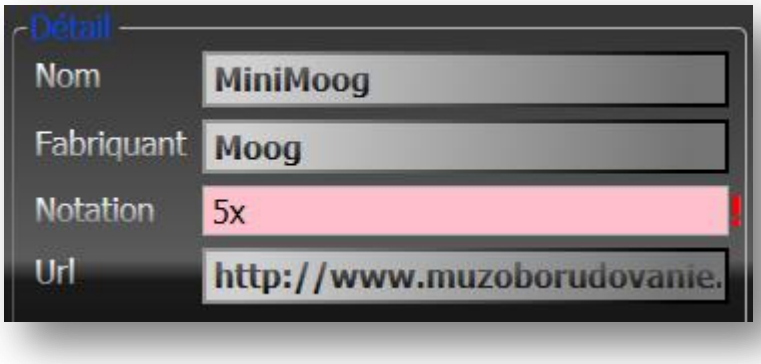


Figure 24 - Amélioration du template de validation

En poussant cette logique plus loin nous pouvons ajouter la prise en charge d'un `ToolTip` affichant le détail de l'erreur :

```
<Setter Property="Control.ToolTip" Value="{Binding RelativeSource={x:Static RelativeSource.Self}, Path=(Validation.Errors)[0].ErrorContent}" />
```

La ligne ci-dessus a été ajoutée au style précédent à la suite du `Setter` passant le fond en rose. On peut voir comment la propriété `ToolTip` du contrôle est modifiée en utilisant le Data Binding et la notion de source relative et de chemin de propriété. Nous n'entrerons pas dans ces détails ici surtout qu'une fois encore rappelons que la création d'un visuel pour une application ne se fait pas en tapant du Xaml mais plutôt en manipulant Expression Blend. Le code montré ici ne sert qu'à expliquer les principes et ne constitue pas une méthode de travail conseillée.

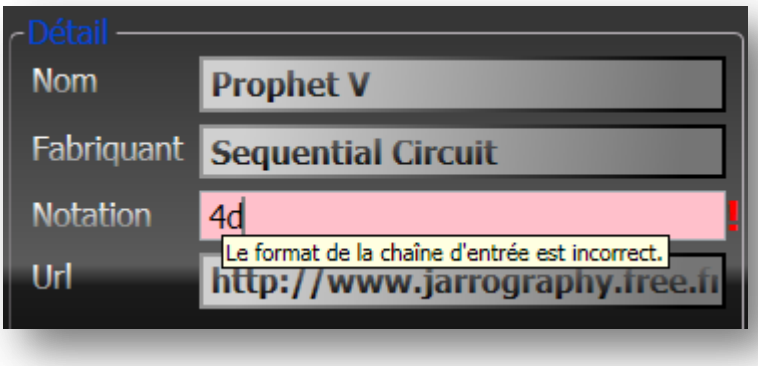


Figure 25 - projet Wpf10-7

Il est facile de voir sur l'image ci-dessus l'accumulation des effets, dont le dernier, le `ToolTip` affichant l'erreur renvoyée par le système de validation du Framework. On notera que chaque élément utilisé n'est pas en soi définitif et immuable puisque tout peut toujours être remplacé par quelque chose de plus sophistiqué. Si vous vous rappelez le premier point de cet article vous comprenez alors que le `ToolTip` jaune par défaut affiché ici peut être remplacé par n'importe quel objet visuel, contenir des images, du texte formaté, etc… L'imbrication des objets offre un champ infini à votre créativité ou au moins à celle de votre infographiste !

Poussons le raisonnement jusqu'au bout.

Puisque WPF est si riche et si souple, pourquoi nous satisfaire des validations par défaut ? Comment intégrer nos propres règles de validation à ce magnifique assemblage de Xaml ? Aurait-on touché les limites du modèle ?

La réponse est bien entendu non aux dernières questions. Et pour répondre à la première disons le sans hésiter, non, nous ne pouvons nous satisfaire des validations par défaut car une application bien faite possède forcément des règles de validation internes qu'il faut pouvoir prendre en compte.

Nous allons rester sur la zone `Notation`. Grâce aux validations par défaut nous savons signaler à l'utilisateur que la saisie n'est pas un entier valide. Mais le Framework ne sait pas que nous avons décidé que la note saisie doit être bornée entre 0 et 5.

Il existe plusieurs façons de prendre en charge cette contrainte. Parmi celles-ci se trouve l'écriture d'une classe héritant de `ValidationRule`, ce que nous allons faire maintenant.

Pour changer, le résultat visuel d'abord :

Figure 26 - La prise en charge de la règle de validation métier

La classe de validation :

```
namespace Raison7
{
   class NotationValidationRule : ValidationRule
   {
      public int MinNotation { get; set; }
      public int MaxNotation { get; set; }

      public override ValidationResult Validate(object value, System.Globalization.CultureInfo
cultureInfo)
      {
         try
         {
            if (MinNotation >= MaxNotation) return new ValidationResult(true, null);
            var i = Int32.Parse(value.ToString());
            if (i<MinNotation) return new ValidationResult(false,
               string.Format("La valeur ne peut pas être inférieure à {0}",MinNotation));
            if (i > MaxNotation) return new ValidationResult(false,
```

```
                    string.Format("La valeur ne peut pas être supérieure à {0}", MaxNotation));
            return new ValidationResult(true, null);
        }
        catch (Exception ex)
        {
            return new ValidationResult(false,ex.Message);
        }
    }
  }
}
```

Cette classe hérite de `ValidationRule`, une classe du Framework. Nous y définissons librement deux nouvelles propriétés, les valeurs mini et maxi autorisées, et nous surchargeons la méthode `Validate`. Cette dernière s'occupe des vérifications et retourne une instance de `ValidationResult` qui indique dans son premier paramètre si la zone est validée ou non. Le second paramètre est un objet, ici nous plaçons directement le message à afficher.

On le constate sur ces quelques lignes de code, la chose est vraiment simple à mettre en place. Mais comment utiliser cette classe dans la page ?

Si nous revenons à la définition du `TextBox` de notation, nous avons le code suivant :

```
<TextBox Grid.Column="1" Grid.Row="2" Margin="3"  Name="tbNotation"
                        Validation.ErrorTemplate="{StaticResource validationTemplate}">
    <TextBox.Text>
        <Binding Path="Notation" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <ExceptionValidationRule/>
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Pour activer la prise en charge de notre propre règle en place et lieu de la gestion par défaut des exceptions nous remplaçons la balise `ExceptionValidationRule` par la suivante :

```
<src:NotationValidationRule MinNotation="0" MaxNotation="5" />
```

Nous avons pris soin aussi d'ajouter le `namespace` de notre code en entête de la page :

```
xmlns:src="clr-namespace:Raison7"
```

Et c'est tout.

Il est bien entendu possible de définir tout cela dans un template ou un style afin de pouvoir appliquer les mêmes règles facilement à tous les `TextBox` d'une fiche ou d'une application ou bien uniquement à certains d'entre eux.

Mises bout à bout, toutes les possibilités de validation permettent de créer des applications riches et réactives et d'améliorer à la fois la fiabilité du code ainsi que ce qu'il convient d'appeler *l'expérience utilisateur*.

Le système de validation de WPF autorise bien d'autres choses, mais ce que nous venons d'en voir justifie pleinement d'en faire une 7ème bonne raison de choisir WPF !

Raison 8 : Le graphisme

Il est vrai que par peur de trop faire passer WPF pour un environnement de développement uniquement destiné à créer des applications hyper lookées j'ai attendu de vous faire voir des choses plus concrètes avant d'aborder les jolis dessins. Mais il ne faudrait pas que cette crainte de voir WPF associé à des interfaces « Guerre des Etoiles » ne fasse oublier ses fantastiques possibilités graphiques !

Après tout, chacun est libre de créer les interfaces qu'il veut. Celles que je vous ai fait voir jusqu'à maintenant étaient plutôt orientées « business form », des fiches de saisie comme on en retrouve dans toutes les applications de gestion.

Comme vous l'avez vu, WPF permet simplement de faire de telles business forms riches et élégantes en mettant à profit sa logique novatrice, qu'il s'agisse de la mise en page, du Data Binding, des validations ou de la gestion des styles. Et même pour ce type de développement WPF surclasse de loin Windows Forms et les autres approches (Win32 ou Java).

Mais WPF permet d'aller bien plus loin !

C'est un environnement résolument tourné vers le multimédia. Ce n'est pas hasard si Vista et demain Seven utilisent WPF comme moteur d'affichage, et ce n'est pas une coïncidence si des machines aussi révolutionnaires que Surface utilisent aussi WPF pour la gestion de leur interface.

Rendons ainsi à César ce qui lui appartient et parlons un peu de graphisme dans WPF.

Je ne vais pas vous refaire le coup du carrousel avec ses vidéos qui flottent dans tous les sens ni même celui des pages qui se tordent et se plient lorsqu'on les tourne (composant créé par Mitsu Furuta de Microsoft qu'il serait injurieux à sa réputation de présenter !). Vous avez certainement déjà vu ces démos et ce sont peut-être elles qui vous ont un peu « effrayé » ? Si on ne peut pas rire de tout avec tout le monde, c'est connu, on ne peut certainement pas tout faire voir à tout le monde non plus. Le bond entre les techniques Win32 ou Windows Forms et WPF est tel, que voir trop vite trop d'effets spéciaux peut créer un mouvement intellectuel de recul, une crainte légitime. A trop vouloir bien faire ces présentations de WPF très multimédia ont parfois loupé leur cible. Alors réaffirmons-le, WPF sert aussi et surtout à faire des vraies applications de tous les jours, juste mieux, plus vite, et pour un résultat plus ergonomique. Avoir des vidéos qui tournoient dans l'espace à grand renfort de musique Rock c'est très sympa, mais peu d'informaticiens peuvent y retrouver une consonance avec leur travail quotidien… Tout le monde ne fabrique pas des sites marchands pour Renault ou Dior au sein de grosses Web Agency parisiennes… L'informatique de gestion n'a que peu à voir avec le monde de la pub ou de la production de dessins animés en 3D, les connexions sont rares entre ces mondes tellement les premiers sont cloisonnés, élitistes et fermés. **Il faut donc réintégrer WPF dans le Daily Business pour démontrer que ce qu'il sait faire est utile à tout le monde**.

Je vais vous faire voir maintenant quelque chose de bien plus sophistiqué que tout ce que je viens d'évoquer et pourtant, je l'espère, de bien concret.

Imaginons en effet un logiciel qui présente des fiches produit, sous WPF desktop ou bien sous Silverlight. Voir des images ou même des vidéos fait partie des bases, mais pouvoir tourner autour du produit, le faire voir en 3D sous tous les angles, n'est-ce pas mieux ? Par force oui. L'utilisation d'images 2D ou 3D ne saurait d'ailleurs se limiter à ce genre de situation : sites marchands, application de gestion, point de vente, suivi boursier, dossier médical… **finalement toutes les applications méritent d'avoir une meilleure interface !**

Pour commencer prenons une image en 3D.

Expression Blend sait manipuler de la 3D mais ne sait pas en créer, pas plus que Visual Studio ou Expression Design. A l'heure actuelle il faudra débourser quelques dollars pour acquérir Zam3D de Electric Rain. Rien à voir avec un Maya ou 3D Studio Max d'AutoCad, c'est beaucoup plus simple à prendre en main (et moins onéreux aussi).

De façon rapide j'ai modélisé un paperclip, un trombone en plastic. Le voici en cours de réalisation sous Zam3D :

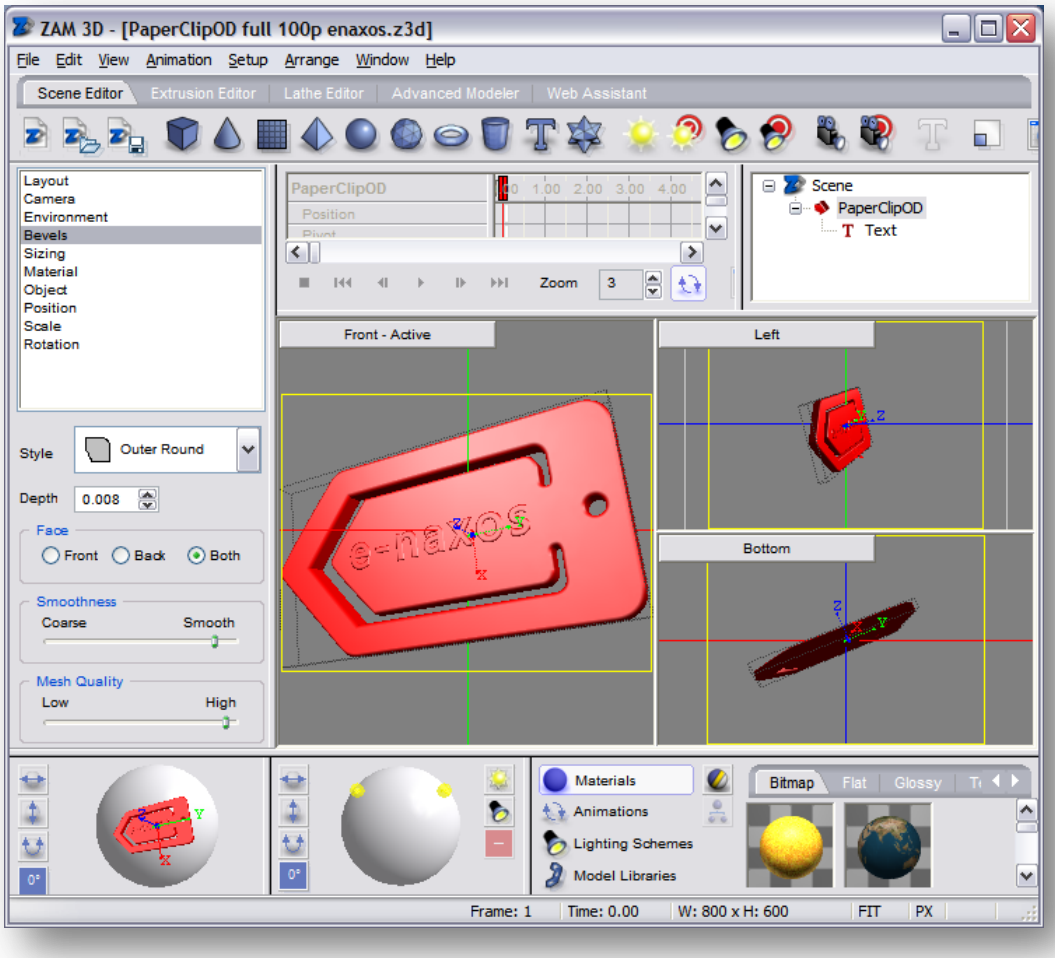


Figure 27 - ZAM3D en action

L'objet est très simple, il a été créé par extrusion et un texte en relief y a été attaché. La matière simule du plastique rouge avec une lumière d'ambiance pas trop forte mais suffisamment pour faire apparaître les ombres et les reflets, avantage de la 3D là où le graphisme 2D avec Design ou Illustrator impose au graphiste de travailler avec une grande précision.

D'un point vue purement pratique, j'en suis convaincu, la 3D est bien plus accessible à un informaticien que la 2D ! L'art du dessinateur c'est de maîtriser l'ombre et la lumière et les softs de 3D le font tout seul. Les softs de 3D sont aussi plus « mathématiques » et plus « logiques », un informaticien s'y sent plus à l'aise que sous Illustrator par exemple. N'hésitez donc pas à vous lancer !

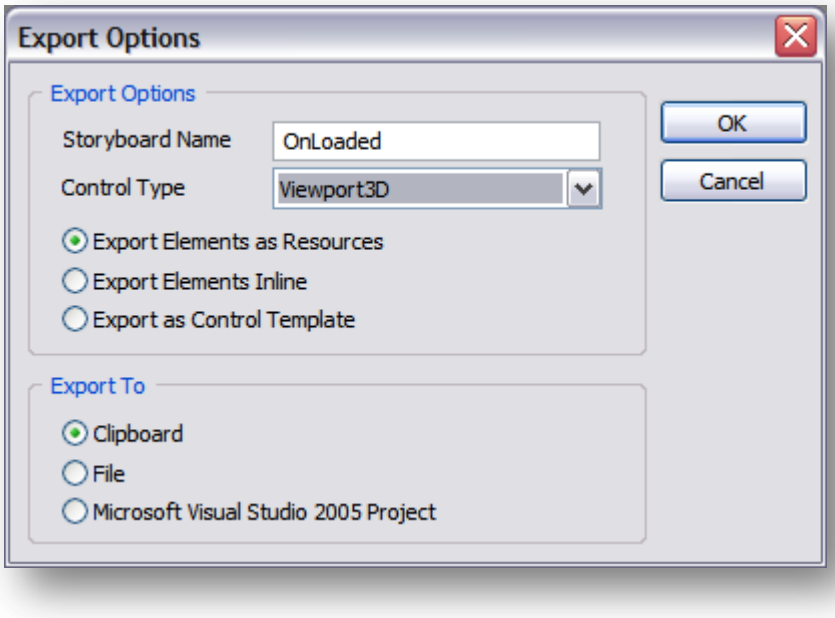


Figure 28 - Exportation XAML d'un projet 3D sous ZAM3D

Zam3D permet d'exporter une scène (avec ses lumières, ses caméras, son éventuelle animation) en Xaml selon plusieurs modes différents.

Passons maintenant sous Expression Blend et créons un nouveau projet WPF. En quelques secondes on place un dégradé en fond de fenêtre, quelques labels et une poignée de boutons. En ré-exploitant un dictionnaire de style on relook immédiatement les boutons et les autres éléments visuels. Enfin on place le `Viewport3D` contenant notre paperclip. En cours de travail nous obtenons ceci :

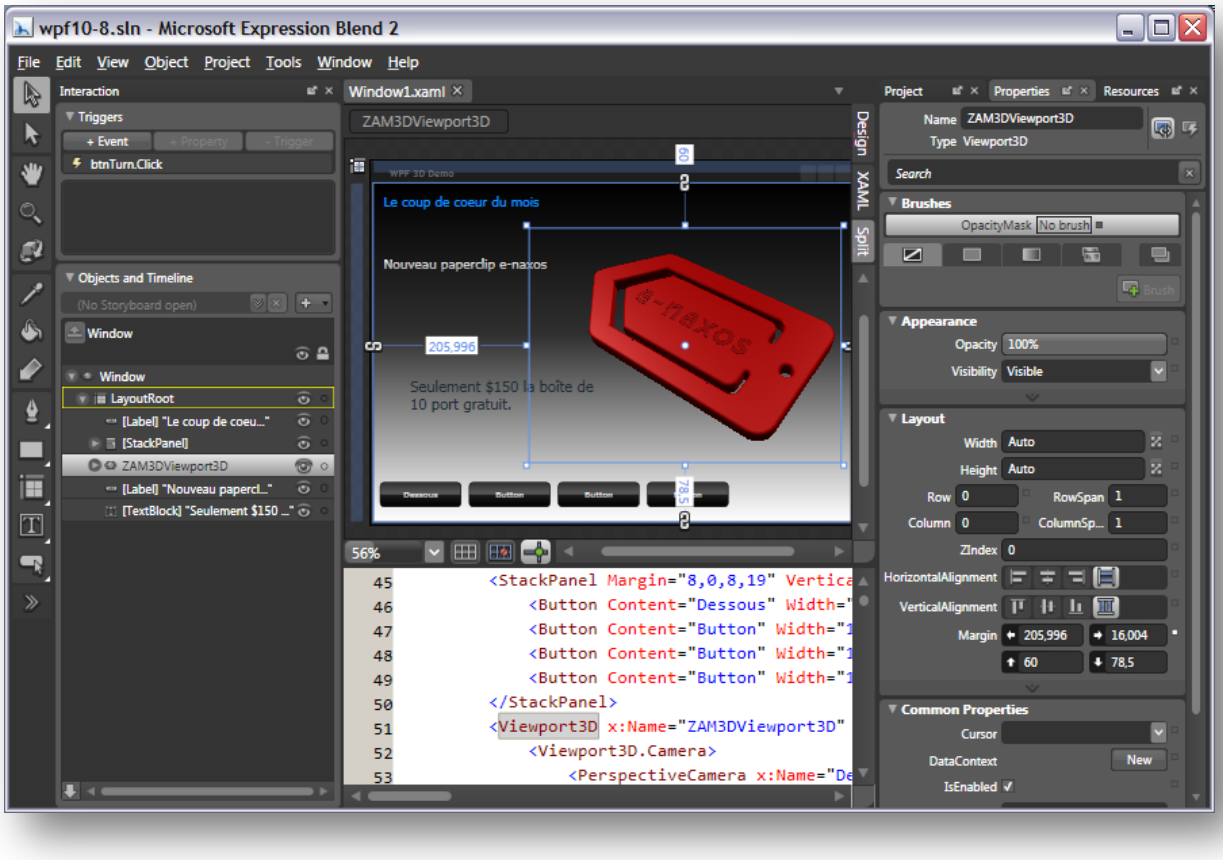


Figure 29 - L'intégration de la source 3D dans une page WPF sous Blend (Projet Wpf10-8)

Maintenant, pour la démonstration nous allons créer une timeline (une animation) assez simple qui va consister à faire pivoter l'objet sur son axe longitudinal.

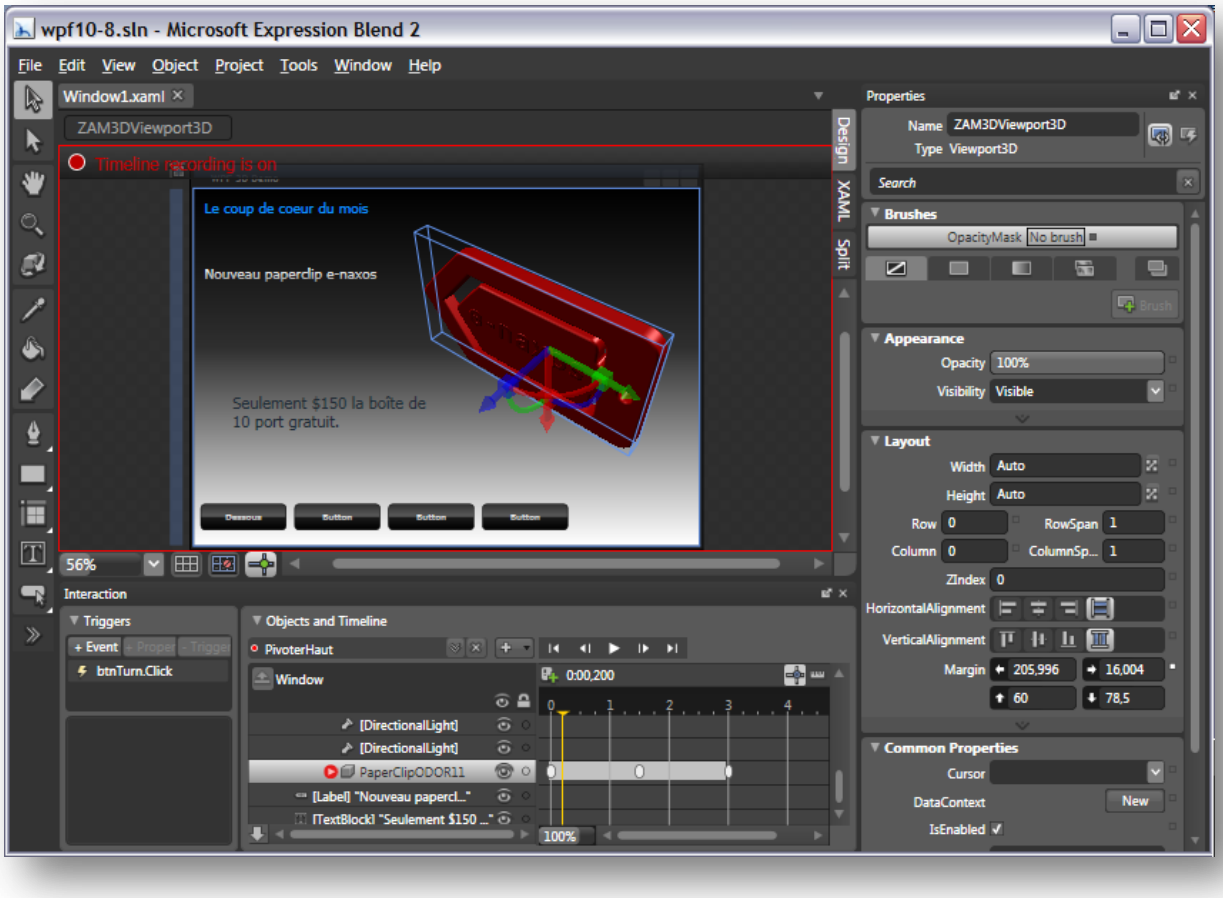


Figure 30 - Animation de l'objet 3D (timeline et storyboard sous Blend)

L'image ci-dessus montre la création de l'animation. On remarque que l'objet (en cours de pivotement) est dessiné dans une boîte 3D délimitant son emplacement et que nous disposons d'un système de trois axes permettant facilement par *drag'n drop* d'effectuer des translations, des changements d'échelle et des rotations suivant les 3 axes.

La timeline s'appelle « PivoterHaut » et on remarque qu'elle est constituée de 3 key frames (des images clé enregistrant les changements de propriétés) entre lesquelles WPF fera une variation automatique.

Pour lancer l'animation nous connectons, via un événement (visible en bas à gauche) le `Click` du bouton `btnTurn` à la méthode `Begin` de notre `StoryBoard` (un `StoryBoard` est une animation complète qui peut contenir plusieurs timelines).

C'est tout. F5 pour lancer en debug, comme sous VS 2008 :

Figure 31 - Une application WPF utilisant la 3D

En cliquant sur le bouton « Dessous » on déclenche l'animation qui fait pivoter le paperclip et en montre le dessous pour ensuite revenir à sa position.

Bien entendu, impossible sur le papier de vous faire voir le résultat visuel, exécutez l'application fournie avec l'article, cela sera bien plus parlant !

Un détail : bien que créé sous Blend, le projet peut être ouvert sous VS 2008 bien entendu. Vous ne pourrez pas créer de storyboard ni modifier visuellement l'animation de l'objet 3D, pour cela il faut Blend, mais vous pourrez voir qu'il n'y a pas une goutte de code C# dans cet exemple et vous pourrez le compiler et l'exécuter.

Bon, bien sûr cette démonstration est ultra simple, mais si vous en avez compris le principe et l'intérêt, c'est l'essentiel !

A noter que Silverlight 3 qui sera disponible en 2009 supportera les accélérations graphiques hardware et la 3D. L'exemple WPF que nous venons de voir pourra ainsi être porté à l'identique en réutilisant les mêmes objets pour créer une application Web riche…

Xaml, Blend et WPF vont devenir au fil du temps les meilleurs alliés des informaticiens travaillant sur des projets innovants, vous avez le choix de prendre ce train de luxe en première

classe maintenant, ou bien de le laisser passer et de vous contenter dans l'avenir d'une seconde classe encombrée. A vous de voir !

Dans tous les cas le support des graphiques 2D et 3D et des animations font bien une 8ème bonne raison de choisir WPF !

Raison 9 : Desktop et Web

Cette raison là ne sera pas accompagnée de code, elle s'énonce simplement et est évidente : WPF, Xaml, Visual Studio et Blend forment un ensemble permettant avec les mêmes outils et les mêmes langages de créer des applications riches ré-exploitant le même code métier, les mêmes objets graphiques au sein d'applications desktop et Web, voire mobile.

Cela est un tel avantage que nul besoin d'en rajouter. Et c'est pour le moins une 9ème raison parfaitement valable de choisir WPF !

Raison 10 : Tout le reste !

Choisir quelques raisons est par force arbitraire comme je le disais en introduction. J'en ai choisi quelques unes qui m'apparaissent décisives mais elles sont loin d'être les seules qui méritent un tel coup de projecteur.

Je n'ai pas montré ici l'interopérabilité Windows Forms / WPF qui existe et fonctionne dans les deux sens (utiliser un composants Windows Forms sous WPF ou le contraire) et qui peut certainement pour de nombreux projets en cours d'écriture ou de maintenance évolutive devenir une bonne raison de passer à WPF sans remettre en cause l'existant.

Je n'ai pas parlé non plus des effets bitmaps et de la richesse des possibilités graphiques ainsi que des performances (WPF est basé sur DirectX sous Windows). Pour certaines applications cela peut être décisif.

Je ne vous ai pas montré la puissance de composants comme le `ListView` ou le `DocumentViewer`, ni de la gestion de thème ou le `VisualStateManager` (que je n'ai fait qu'évoquer), ni même n'ai abordé le *WPF Toolkit* et tous les composants Open Source qui lui sont ajoutés au fil du temps par Microsoft. Pourtant tout cela peut largement faire différence avec les autres environnements.

Simplification de la navigation dans les applications, gestion fine de la sécurité, déploiement ClickOnce, interopérabilité Win32, etc, sont encore autant de domaines qui font de WPF une plateforme unique en termes de puissance et d'innovation.

Enfin, le modèle de développement proposé par WPF, le **découplage total entre code applicatif et présentation visuelle** faisant qu'une intervention dans l'un de ces domaines n'entraîne pas de régression ou de bogue sournois dans l'autre, et mieux, la possibilité de disposer d'outils pouvant travailler simultanément sur les mêmes projets en séparant clairement le travail de l'informaticien et de l'infographiste qui prend alors une place entière dans le processus de conception, ce modèle là est tellement puissant qu'à lui tout seul il justifie de choisir WPF…

Bref, tous ces aspects pourtant essentiels et que j'ai oubliés de vous montrer ici, faute de temps et de place, tous, et au moins collectivement, méritent de faire une 10ème bonne raison de choisir WPF !

Conclusion

Comme je le disais en introduction de cet article, choisir 10 raisons de préférer WPF est totalement arbitraire, il en existe bien d'autres comme l'évoque la 10ème raison ci-avant !

Comment dire parmi toutes les qualités et ouvertures de WPF lesquelles sont les plus essentielles ? La tâche m'a semblé impossible avant d'écrire cet article, arrivé à son terme je mesure à quel point j'étais encore loin d'en sonder la véritable difficulté !

Vous avoir converti à WPF, c'est bien entendu mon espoir secret, et si j'aime placer la barre assez haut je sais aussi faire preuve de modestie, ainsi, si j'ai au moins mis le doute dans votre esprit et si pour votre prochain développement vous hésitez à refaire du Windows Forms au profit de WPF, alors finalement je serai déjà heureux d'avoir pu y être un peu pour quelque chose dans ce changement d'état d'esprit…

Je remercie dans tous les cas les plus courageux d'entre vous, ceux qui auront atteint cette dernière ligne !

Thanks,

Et … Stay Tuned !

Olivier Dahan

Dot.Blog : www.e-naxos.com/blog

Les projets de type XBap nécessiteront une modification pour fonctionner : allez dans les propriétés du projet, onglet Debug et modifiez le chemin complet du fichier de la section « Start Browser with URL » pour l'accorder avec l'arborescence où vous aurez copié chaque exemple.

Les exemples fonctionnent sous Visual Studio 2008 avec les derniers SP ainsi qu'avec Blend 2 SP1.