Formations .NET – Audit, Conseil, Développement

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

© Copyright 2009 Olivier DAHAN
MICROSOFT MVP C#

Reproduction, utilisation et diffusion interdites sans l'autorisation de l'auteur. Pour plus d'information contacter odahan@e-naxos.com

Les Propriétés de dépendance et les propriétés jointes

[WPF / Silverlight]

Version 1.01

Sommaire

Les propriétés de dépendance et les propriétés jointes sous WPF et Silverlight....3
Qu'est-ce ?....3
Des propriétés avant tout....3
... et des dépendances....3
Et les propriétés jointes ?....4
Comment ça marche ?....5
Utiliser une propriété jointe....5
Les variantes syntaxiques....6
Des noms qui changent.......7
Propriété de dépendance ou propriété jointe ?....8
Le service d'animation....8
Les autres services WPF pour les propriétés de dépendance....10
Valeur par défaut....10
Les expressions....10
Data Binding....10
L'héritage des valeurs....11
Les styles....11
Validation des valeurs....11
Précédence des valeurs....11
Notification de changement....11
Point intermédiaire....12
Définir des propriétés dépendantes....12
Le composant exemple : une jauge....12
Créer un user control WPF....13
Le code du contrôle....15
DependencyProperty....15
Le callback de notification de changement de valeur....16
Le callback de validation de la valeur....16
Utiliser une propriété de dépendance....17
Fixer des valeurs....18
A la main....18

En XAML 18
Par Data Binding 19
Animer la propriété 20
La définition des propriétés jointes 22
Enregistrement 22
Setter et Getter statiques 23
Partager une propriété de dépendance 23
Précédence et évaluation 24
Conclusion 25

Table des figures

Figure 1 - Création d'un TextBlock ____ 6
Figure 2 - Animation d'une propriété attachée ____ 9
Figure 3 - Création d'un UserControl Jauge ____ 13
Figure 4 - La conception visuelle du contrôle ____ 14
Figure 5 - La propriété de notre contrôle sous Blend ____ 18
Figure 6 - La propriété manipulée en XAML sous VS ____ 19
Figure 7 - L'animation de la jauge ____ 21
Figure 8 - Création de l'animation sous Blend ____ 21

Les propriétés de dépendance et les propriétés jointes sous WPF et Silverlight

Qu'est-ce ?

Après m'être trituré les méninges j'avoue ne pas avoir trouvé de traduction française élégante et éloquente pour le terme « *dependency property* ».

Dans les traductions Microsoft on trouve l'appellation « *propriétés de dépendance* » qui me parait assez peu évocateur, j'aurais plutôt penché pour « propriétés dépendantes », plus proche du fonctionnel, mais finalement cet article est là pour éclaircir ce concept alors ne chipotons pas sur la terminologie !

Quant aux propriétés jointes, traduction Microsoft de « *attached properties* », j'aurais pensé assez logique de les appeler « propriétés attachées » pour rester plus proche de l'expression américaine. Mais là encore, jointes ou attachées, tant que vous n'aurez pas lu ce que va suivre, je ne suis pas convaincu que le choix de l'adjectif change grand-chose à la compréhension *a priori* de ce qui se cache derrière.

> *Le lecteur notera que nombreux concepts propres à WPF et Silverlight évoqués ici ne sont pas forcément expliqués pour maintenir une taille raisonnable au présent article qui reste ciblé sur son sujet principal. En revanche je l'invite à se reporter à mon article précédent « **10 bonnes raisons de préférer WPF** ».Il a été conçu pour être à la fois un plaidoyer en faveur de WPF et une introduction aux principaux concepts de cette technologie. Vous trouverez cet article sur mon site, rubrique des téléchargements.*

So, let's go !

Des propriétés avant tout

Les propriétés, de façon générale, sont des attributs définissant des caractéristiques et des états d'un objet (ou d'une classe pour les propriétés statiques). En ce sens les propriétés de dépendance ne diffèrent pas des propriétés « habituelles », celles qu'on déclare avec un *getter* et un *setter*. Propriétés que nous appellerons ici « propriétés CLR » pour ne pas les confondre, non pas avec les fraises des bois mais avec les propriétés de dépendance et puisque le mécanisme des propriétés « habituelles » est justement défini dans le CLR.

> *Les propriétés de dépendance sont ainsi avant toute chose des propriétés comme les autres, c'est la façon de s'en servir qui justifie la mise en place d'un mécanisme et d'une implémentation qui diffèrent de ceux utilisés par les propriétés CLR.*

… et des dépendances

En quoi ces propriétés « de dépendance » sont-elles différentes des autres propriétés ? Et surtout en quoi consistent ces « dépendances » ?

Sous WPF (et Silverlight), qui repose sur XAML, les objets, surtout ceux intervenants au niveau de l'interface, sont manipulés de telle sorte que certaines de leurs propriétés dépendent de plusieurs sources différentes qui viennent les alimenter : valeur par défaut, Data Binding, animations, styles, ressources, etc… Gérer de façon efficace les multiples sources connectées à une même propriété est assuré par un mécanisme particulier de WPF.

Les propriétés de dépendance peuvent ainsi se définir comme étant les propriétés gérées par ce mécanisme.

Il ne s'agit pas de remplacer ou de modifier la façon dont les propriétés sont gérées par le CLR ni de créer une nouvelle « espèce » de propriétés, choix conceptuellement valides mais dont les impacts ont certainement été jugés disproportionnés par l'équipe de développement de WPF et par celle du Framework. Ainsi, les propriétés de dépendance sont basées à la fois sur une couche de gestion implémentée dans WPF et sur des guidelines pour les mettre en œuvre.

Il faut noter que les propriétés de dépendance permettent aussi d'enrichir le mécanisme standard des propriétés CLR par la possibilité de définir des valeurs par défaut, la notification de changement et la validation des valeurs. Toutes ces choses sont possibles avec les propriétés CLR mais WPF unifie la façon dont elles sont déclarées et utilisées.

Et les propriétés jointes ?

Elles sont définies comme des propriétés de dépendance qui permettent de décorer d'autres classes.

Pour être plus clair, prenons l'exemple de la propriété `Dock` sous Windows Forms : on la retrouve dans tous les composants visuels bien écrits. C'est-à-dire que tous les composants susceptibles d'être un jour dockés doivent posséder un champ interne privé pour stocker la valeur de `Dock` et ils doivent tous exposer la propriété en question. Malheur au développeur qui ne suivrait pas cette logique ! Son ou ses composants ne pourraient tout simplement pas être dockés…

Chaque instance de composant visuel consomme ainsi de la mémoire pour stocker une valeur qui, en réalité, n'est que rarement utilisée. Et si cette propriété est oubliée dans la définition d'une classe, elle ne pourra pas participer, le jour venu, à une mise en page impliquant l'utilisation du docking.

Fâcheuse situation s'il en est… Surtout que la propriété `Dock` n'est pas la seule de ce genre ! Sous Windows Forms, qui fixe un cadre très limité à l'environnement visuel, c'est encore jouable, mais sous WPF où le visuel est totalement libre utiliser une telle logique est tout bonnement irréalisable. Que dire de `Top`, `Bottom`, `Right` et `Left` qui, sous WPF, n'ont de sens que pour les composants posés sur un `Canvas`. Ces propriétés sont inutiles si le composant est placé dans un conteneur de type `StackPanel` ou `Grid`. Dans ce dernier cas il faut en revanche des propriétés ligne et colonne qui n'ont aucun intérêt si le composant est placé dans un `Border`. Etc.

On voit bien que dans un environnement de type WPF qui laisse une très grande liberté aux développeurs pour définir les interfaces visuelles il n'est simplement plus possible d'imposer la présence d'une longue liste figée de propriétés.

D'une part il est impossible de prévoir toutes les propriétés qui « seraient » utiles un jour, et, d'autre part, même si cela était possible, il faudrait que toutes les instances de chaque composant utilisé réservent dans la mémoire de nombreux espaces de stockage pour tout un tas de propriétés qui seraient inutilisées dans la majorité des cas.

Les propriétés de dépendance ou jointes nécessitent, on le comprend mieux maintenant, la mise en place d'un mécanisme un peu plus subtile que d'imposer des listes de propriétés impossible à prévoir en réalité et consommant en pure perte de la mémoire.

Ce mécanisme justifie à lui seul la mise en place des propriétés dépendantes sous WPF.

Comment ça marche ?

Une fois l'intérêt des propriétés de dépendance exposé, la question qui vient immédiatement est de savoir comment les utiliser et les définir.

Utiliser une propriété jointe

Prenons une simple `Window` avec un conteneur de type `Canvas`. Ajoutons un bouton qui servira à déclencher la démonstration et tapons le code suivant dans le gestionnaire de l'événement `Click` :

```
private void btnCreateText_Click(object sender, RoutedEventArgs e)
{
    TextBlock text = new TextBlock(new Run("Texte de démo"));
    text.SetValue(Canvas.TopProperty, 100.0);
    text.SetValue(Canvas.LeftProperty,50.0);
    canvas1.Children.Add(text);
}
```

Dans ce code nous créons un bloc de texte (première ligne). Comme nous voulons l'afficher à un endroit précis du `Canvas` nous devons fixer deux propriétés, les coordonnées `X`,`Y` de son coin supérieur gauche. Hélas (et tout à la fois heureusement !) le composant `TextBlock` ne possède aucune propriété de ce type. En revanche, la classe `Canvas` expose deux *propriétés jointes* servant à décorer ses composants enfants pour en gérer correctement l'affichage.

Il nous est donc possible (les deux lignes suivantes) d'utiliser la méthode `SetValue` de l'objet texte pour lui indiquer que nous souhaitons initialiser les propriétés `TopProperty` et `LeftProperty` qui proviennent de la classe `Canvas`, respectivement avec les valeurs `100` et `50`.

En réalité nous ne stockons pas ces valeurs dans l'instance du `TextBlock` (il ne sait rien à propos des propriétés en question) mais dans les `DependencyProperty` déclarés dans la classe `Canvas` pour gérer `Top` et `Left`… Par convention on notera que les propriétés de dépendance (et donc les propriétés jointes) ont leur nom qui se termine par le suffixe « `Property` ». En tout cas c'est de cette façon que le champ statique et public contenant la propriété doit être déclaré. Nous reviendrons là-dessus plus tard.

Il ne reste plus (dernière ligne) qu'à ajouter l'instance du `TextBlock` à la liste des enfants du composant `canvas1`. Et l'affaire est jouée (voir la figure 1) !

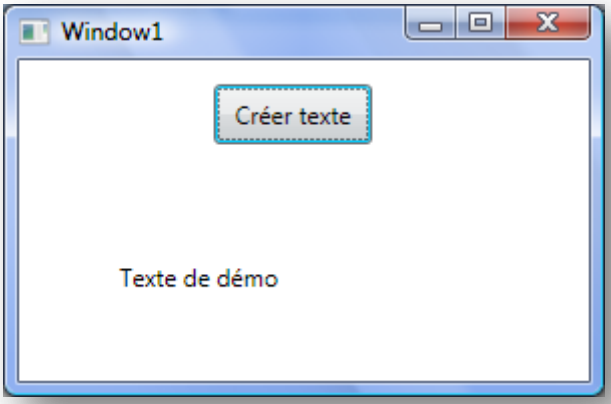

Figure 1 - Création d'un TextBlock

Les variantes syntaxiques

La façon d'utiliser les propriétés jointes que nous venons de voir permet de comprendre le mécanisme, mais il existe d'autres façons d'obtenir le même résultat.

Dans un premier temps, au lieu de partir du `TextBlock` et d'utiliser sa méthode `SetValue` nous pouvons faire l'inverse, c'est-à-dire partir du `Canvas` :

```
private void btnCreateText_Click(object sender, RoutedEventArgs e)
{
    TextBlock text = new TextBlock(new Run("Texte de démo"));
    Canvas.SetTop(text,100);
    Canvas.SetLeft(text,50);
    canvas1.Children.Add(text);
}
```

Comme on le voit ici, c'est en partant du `Canvas` qu'on attribue les valeurs directement aux propriétés en indiquant quel composant est concerné. Exactement l'inverse du premier exemple. Les deux approches sont valides, question de point de vue, et en termes de stylistique on obtient quelque chose d'équivalent avec malgré tout un code un peu plus court et l'avantage du typage de la valeur de la propriété (c'est pourquoi je préfère cette approche personnellement).

Mais on peut faire plus court en se passant totalement de code C# !

> *Il ne faut jamais oublier en effet que WPF est une technologie qui englobe un compagnon indissociable : XAML. Même s'il n'est pas à mon sens plus judicieux de taper du XAML à la main que de vouloir écrire une application Web en tapant du HTML dans le bloc-notes de Windows, XAML permet de décrire totalement le visuel d'une application ainsi que bon nombre de comportements et cela sans faire appel à du code-behind. Savoir intervenir dans le code XAML est une connaissance indispensable même si des outils comme Expressions Blend et Visual Studio (dans une moindre mesure) permettent d'automatiser son écriture à l'instar de ce que peut être Expression Web pour créer des pages HTML.*

Néanmoins pour l'instant restons en mode « manuel » pour mieux illustrer le propos de cet article et voyons comment nous pourrions nous passer totalement de code C# :

```
<Canvas Name="canvas1">
   <TextBlock Canvas.Top="140" Canvas.Left="170">Autre bloc de
texte</TextBlock>
</Canvas>
```

On voit ici que XAML simplifie grandement la création de l'instance du `TextBlock` (pas de « `new` » notamment). Les deux propriétés jointes `Canvas.Top` et `Canvas.Left` sont directement modifiées au sein de la balise du `TextBlock`.

Des noms qui changent…

Les plus attentifs auront noté qu'il existe une légère variation dans le nom des propriétés selon le contexte, ce qui peut être source de confusion, avouons-le.

Ainsi, dans le premier exemple de code nous utilisons la propriété `Canvas.TopProperty`, dans le second nous utilisons `Canvas.SetTop` et dans le troisième (en XAML) nous utilisons `Canvas.Top`. Pour le coup c'est pas … top !

Clarifions immédiatement les choses :

Lorsque nous utilisons la méthode statique qui joue le rôle de *setter* de la propriété `Top` de la classe `Canvas`, il est normal (et c'est une convention d'ailleurs) d'appeler cette méthode `Set<nom de propriété>`. D'où le `Canvas.SetTop`.

Lorsque la propriété de dépendance (ou jointe) est définie, et nous verrons plus loin comment le faire, elle est enregistrée avec son nom, ici `Top`. Il est donc normal d'utiliser `Canvas.Top` dans le code XAML qui se base sur le nom enregistré à la création de la propriété (la classe `Canvas` étant compilée et fonctionnelle le designer visuel peut utiliser l'instance pour accéder aux propriétés de dépendance comme au reste de l'objet).

Enfin, quand par code C# nous utilisons `TextBlock.SetValue` il nous faut passer le nom de la propriété. Or, le nom de la propriété (`Top`) n'est qu'un enregistrement effectué par code dans la classe `Canvas`, au même titre qu'on peut ajouter un élément à une liste générique par sa méthode `Add`. L'item ajouté n'existe pas pour autant de façon autonome dans la classe liste ni dans celle qui héberge la liste. C'est la même chose pour les propriétés de dépendance et jointes : leur nom n'est jamais qu'un bout de texte ajouté à une liste gérée par WPF, le nom n'est pas un identificateur codé en C#. Le code XAML sait retrouver ce nom (le `Canvas` posé sur la fiche en mode design est une instance bien réelle dont les propriétés sont initialisées) mais en revanche le compilateur C# ne sait pas le faire, il ne travaille pas avec le designer visuel ni avec des objets instanciés, il ne peut accéder qu'à une « vraie » propriété CLR ou un champ déclaré dans le code.

C'est pourquoi les propriétés de dépendances sont déclarées « en double » : une fois lors de leur enregistrement dans le système de gestion de WPF et une seconde fois en code C# sous la forme de propriétés statiques portant le nom de la propriété XAML. Mais pour les propriétés jointes on ne déclare pas de telles propriétés statiques, on accède alors au champ statique et publique qui définit la propriété et qui, par convention, s'appellent `<Propriété>Property`, d'où le nom `TopProperty` auquel on accède via la classe `Canvas`, `Canvas.TopProperty` n'étant rien d'autre qu'un champ statique et public de la classe `Canvas`.

J'espère avoir été clair mais il me semble percevoir comme un air dubitatif sur votre visage… Pas de panique ! Nous allons étudier comment définir des propriétés de dépendance et des propriétés jointes et les choses devraient devenir plus limpides, du moins je vais tout faire pour ça ☺

Propriété de dépendance ou propriété jointe ?

L'exemple que nous venons de développer montre des propriétés jointes (*attached properties*). En effet, `Top` et `Left` n'appartiennent pas au composant `TextBlock` mais à la classe `Canvas` qui, via les mécanismes de WPF, permet de décorer[1] ses enfants visuels[2] avec ces propriétés.

Si on comprend bien ici l'intérêt des propriétés jointes, qu'en est-il des propriétés de dépendance « simples » ? Prenons l'exemple de `Canvas.Height`. Elle n'est utilisée par aucun enfant visuel de `Canvas` et cela n'aurait d'ailleurs pas beaucoup de sens, la hauteur du `Canvas` n'est applicable qu'à un objet bien précis de type `Canvas`. Pourtant cette propriété est définie comme une propriété de dépendance et non comme une propriété CLR dans la classe `Canvas`.

Pourquoi les concepteurs de la classe `Canvas` se sont-ils compliqués l'existence ?
… Tout simplement pour bénéficier des services que WPF offre aux propriétés de dépendance !

Le service d'animation

Quels sont ces services que WPF offre aux propriétés de dépendance qu'il n'offre pas aux propriétés CLR ?

Il y en a plusieurs, mais l'un des plus marquant parce qu'il tranche avec tout ce qui était possible avant WPF c'est le service d'animation.

Toute propriété de dépendance (et donc une propriété jointe) peut être animée ce qui n'est pas possible avec une propriété CLR.

Prenons un joli cercle (une instance de `Ellipse`) que nous posons sur le `Canvas` de la fiche et ajoutons une animation très simple qui aura pour effet de faire bouger le cercle de droite à gauche et de gauche à droite éternellement. Voici le code XAML de la définition de cet objet et de son animation :

---

[1] « décorer » est un peu un abus de langage ici. Les propriétés jointes sont accessibles via les mécanismes présentés dans cet article mais ne « décorent » pas au sens strict du terme d'autres classe comme par exemple les class helpers décorent d'autres classes avec des méthodes ne leur appartenant pas. Disons que « décorer » est une bonne approximation pour se faire une idée du fonctionnement des propriétés jointes à la condition qu'on ne soit pas trop à cheval sur la précision terminologique…

[2] En réalité on peut utiliser une propriété attachée même sur une instance qui n'est pas enfant visuel de la classe offrant la dite propriété. Cela n'a souvent que peu d'intérêt puisque la valeur attribuée ne sera pas utilisée, mais WPF ne l'interdit pas (avec de l'imagination on peut certainement s'en servir dans ce contexte !).

```
        <Ellipse Canvas.Left="12" Canvas.Top="46" Height="27"
                 Name="ellipse1" Stroke="Black" Width="27" Fill="SkyBlue">
            <Ellipse.Triggers>
                <EventTrigger RoutedEvent="Ellipse.Loaded">
                    <BeginStoryboard>
                        <Storyboard RepeatBehavior="Forever">
                            <DoubleAnimation
                                Storyboard.TargetName="ellipse1"
                                Storyboard.TargetProperty="(Canvas.Left)"
                                From="12" To="300" AutoReverse="True"
                                RepeatBehavior="Forever"/>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Ellipse.Triggers>
        </Ellipse>
```

La propriété `Canvas.Left` qui est animée dans cet exemple est une propriété jointe à l'ellipse par le `Canvas`, mais nous aurions pu de la même façon animer la propriété `Height` du `Canvas` comme nous l'évoquions plus haut, propriété qui est une propriété de dépendance mais pas jointe. Il n'y aucune différence de comportement en réalité, en tout cas du point de vue des services WPF disponibles.

> *Le code XAML peut devenir très verbeux, comme tout format dérivant de XML et plus généralement de SGML qui a donné de nombreux enfants dont XML, HTML et XAML. Ainsi, le code de l'exemple est un peu lourd à mon goût. C'est pourquoi j'insiste toujours sur le fait que le visuel d'une application WPF ou Silverlight ne se travaille pas en XAML mais à l'aide d'outils visuels, qu'il s'agisse de l'éditeur WPF de Visual Studio ou mieux de Expression Blend qui permet bien plus de choses comme justement la création d'animations.*

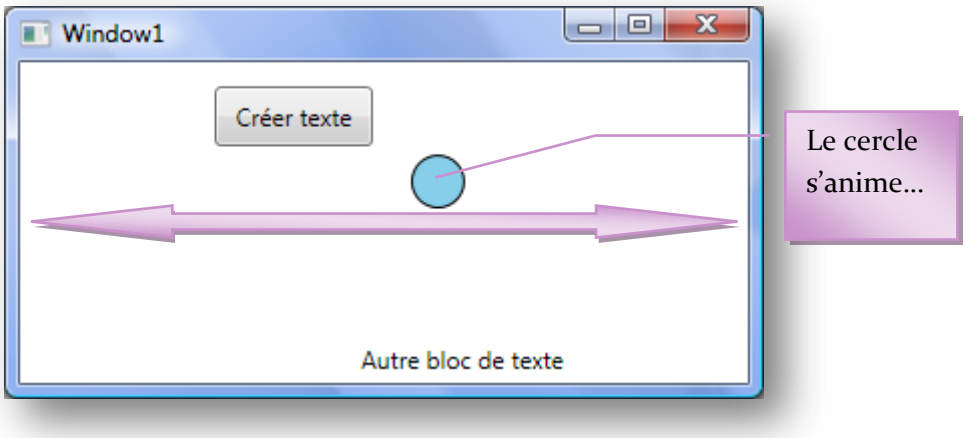

Figure 2 - Animation d'une propriété attachée

Le but de cet article n'étant pas de présenter XAML je n'entrerai pas dans les détails du code de l'exemple, mais pour les curieux en voici les grandes lignes :

Pour lancer l'animation nous nous connectons sur un déclencheur (*trigger*) de l'objet `Ellipse` et plus particulièrement sur l'événement `OnLoaded` qui est lancé une fois l'objet totalement chargé et initialisé. Directement dans le code du déclencheur nous créons un `StoryBoard`, un

objet permettant de regrouper des animations. Puis, à l'intérieur de ce `StoryBoard` nous déclarons une `DoubleAnimation`, c'est-à-dire une animation de propriété de dépendance prenant une valeur exprimée par un `Double`. Les paramètres de l'animation elle-même fixent la propriété ciblée, la durée de l'animation, l'étendue de la variation de la valeur etc… Rien de bien compliqué en réalité mais le côté verbeux de XAML rend très vite le code un peu indigeste dès qu'on dépasse la simple déclaration d'un bouton… En revanche, une fois cette impression dépassée, on s'aperçoit à quel point le pouvoir descriptif de XAML est grand. Au final on apprécie assez rapidement ce langage qui permet de décrire ce qu'on veut obtenir plutôt que de coder comment le faire ! Une approche fonctionnelle très à la mode, à juste titre.

Les autres services WPF pour les propriétés de dépendance

Certains pourraient se dire que tout ça pour des animations c'est peut-être un brin se prendre la tête pour quelque chose qui n'intéresse pas tout le monde… En se disant cela ils feraient en réalité deux erreurs grossières. La première étant de considérer les animations comme un exotisme réservé à certains développements particuliers, la seconde serait de croire que les services WPF se limitent au service d'animation.

Si je ne reviendrai pas sur le fait que les animations (même subtiles) sont aujourd'hui indispensables pour rendre attractives et ludiques toutes les applications, quel que soit leur type, et que rendre une application agréable à utiliser est au moins aussi important que de la rendre fiable et pertinente fonctionnellement, je peux en revanche vous en dire plus sur les autres services que WPF offre aux propriétés de dépendance :

Valeur par défaut

Fixer une valeur par défaut à une propriété est souvent indispensable. WPF permet de le faire pour toute propriété de dépendance avec certains avantages sur la méthode classique. Parmi ces améliorations on notera le fait que la déclaration de la valeur par défaut s'effectue au même endroit et par la même méthode que celle qui permet de déclarer la propriété (c'est donc plus simple et plus centralisé). WPF permet aussi à cette valeur par défaut de « refaire surface » lorsque cela est nécessaire (arrêt d'une animation, suppression d'un style…). Nous reviendrons sur ces aspect plus loin.

Les expressions

En XAML les propriétés dépendantes peuvent se voir attribuer une valeur qui est le résultat d'une expression et non pas seulement une constante. Cela simplifie le code tout en étendant la puissance de XAML, d'autant que le développeur peut même créer ses propres mécanismes de calcul en créant des classes dérivées de `System.Windows.Expression`.

Data Binding

Le Data Binding de WPF ne se présente plus, c'est l'un des points forts de cette technologie (voir à ce propos mon article précédent « *Dix bonnes raisons de choisir WPF* »). Ce service n'est offert qu'aux propriétés de dépendance. S'il fallait une raison pour justifier cet article cela serait certainement celle-là car écrire des classes pour WPF sans utiliser le mécanisme des propriétés de dépendance et se priver du Data Binding ne serait pas sérieux, tout simplement. En utilisant des propriétés CLR on ne peut pas bénéficier du Data Binding.

Exemple de Data Binding (XAML) :

```
<Canvas>
  <TextBlock Text="{Binding Client.Contact}"/>
</Canvas>
```

L'héritage des valeurs

Un peu comme les feuilles de style CSS, WPF permet aux valeurs des propriétés de dépendance de se propager le long de l'arbre visuel, en cascade. Par exemple une fonte définie à un niveau supérieur sera propagée aux enfants visuels (sauf s'ils définissent explicitement une fonte localement) et ce sans que la valeur de la propriété ne soit réellement dupliquée dans chaque instance.

Les styles

J'évoquais CSS au paragraphe précédent, WPF propose un mécanisme de même type (mais plus sophistiqué) grâce aux styles et aux templates.

Validation des valeurs

WPF permet d'enregistrer un code de validation lors de la création d'une propriété de dépendance. Ce code n'est pas placé dans le *setter* comme dans une propriété CLR mais dans une méthode à part. La validation se double d'un autre mécanisme permettant de corriger les valeurs avant qu'elles ne soient prises en compte, nous aborderons ces points plus loin.

Précédence des valeurs

Lorsqu'on obtient une valeur d'une propriété de dépendance on obtient quelque chose qui provient potentiellement de plusieurs sources : valeur locale, template, style, etc. WPF instaure des règles de précédences pour clarifier la logique appliquée.

WPF sait ainsi gérer plusieurs niveaux de précédence au sein des différentes valeurs d'une propriété de dépendance. Il sait même revenir à une valeur par défaut si la valeur courante est supprimée. Par exemple si un style a été appliqué et qu'il est supprimé, les propriétés concernées peuvent reprendre automatiquement leur valeur par défaut sans que cela ne réclame de code spécifique. De même lorsqu'une animation se termine les propriétés concernées reprennent « comme par magie » la dernière valeur connue. Dans le même esprit, si l'animation d'une valeur débute, elle prendra la précédence sur la valeur actuelle de la propriété sans que celle-ci ne soit perdue définitivement. Aucun code n'est à écrire par le développeur pour bénéficier de ces avantages.

Notification de changement

Les propriétés de dépendance exposent un mécanisme de notification de changement de valeur qu'on peut comparer au support de l'interface `INotifyPropertyChanged` pour les propriétés CLR. Dans le cas des propriétés dépendantes cela est automatique.

> *Il s'agit d'une fonction clé. D'ailleurs dans notre tentative de comprendre le sens du terme « dependency property» en introduction nous aurions pu ajouter que ces fameuses dépendances font plutôt référence, selon Microsoft, aux notifications de changement de valeur car c'est ici que la notion de dépendance (d'interdépendance) entre les objets se manifeste de la façon la plus marquante. Très souvent l'état d'un objet dépend de celui*

d'un autre objet, voire de plusieurs, et tout changement d'un de ces états impacte sur l'état des objets liés. Gérer de telles dépendances est toujours une tâche complexe et délicate, tout mécanisme visant à la simplifier est donc bienvenue. Les propriétés de dépendance de WPF apportent une solution à cette problématique.

Point intermédiaire

Arrivé à ce stade de l'article nous commençons à voir comment les propriétés de dépendance et les propriétés jointes fonctionnent et ce qu'elles apportent. Nous pouvons peut-être un peu résumer tout ça…

> *Le système des propriétés dépendantes de WPF est en fait un « moteur de calcul de valeurs ». Le fait que la valeur soit stockée quelque part n'est en fait qu'un des moyens de créer la valeur, il en existe d'autres comme les animations (qui fabriquent les valeurs par calcul au fur et à mesure que le temps passe), les styles, etc…*
>
> *Obtenir la valeur d'une propriété de dépendance ou jointe revient à lancer le moteur de calcul de valeur. L'invalidation d'une ou plusieurs valeurs intervient lorsque le moteur de calcul s'aperçoit que les dépendances ont changé.*
>
> *C'est ce qui fait que l'utilisation du cache de valeurs offert par WPF est très importante car il n'est pas possible de prévoir si le calcul pour créer une valeur donnée est simple ou complexe, donc rapide ou très lent. Il est ainsi essentiel pour les performances globales que les valeurs soient cachées, et disponibles rapidement, jusqu'à ce qu'elles soient invalidées (par exemple une valeur locale peut être supprimée en utilisant la méthode `ClearValue`).*

Il est temps maintenant de voir comment implémenter des propriétés dépendantes !

Définir des propriétés dépendantes

Le composant exemple : une jauge

Si la définition d'une propriété de dépendance est une chose bien simple, encore faut-il la définir quelque part ! Il nous faut donc un composant hôte qui exposera une telle propriété. Il faut, de plus, que les changements de la propriété en question aient un effet visuel pour que la démonstration soit éloquente. Il faut, en outre, que tout cela tienne en peu de code, le but de l'article n'étant pas de créer un gros composants qui noierait l'essentiel du propos (les propriétés de dépendance) dans un flot de code annexe.

C'est pourquoi j'ai choisi de concevoir ici un composant jauge. Ultra simple et pas très beau, cela va sans dire.

Ce composant se présentera sous la forme d'un rectangle dégradé orienté verticalement. Il possèdera un autre rectangle plus petit et d'une autre couleur qui se comportera comme un curseur, montant et descendant à l'intérieur de l'espace du premier rectangle.

La position du curseur sera définie par une propriété de dépendance acceptant des valeurs entières de 0 à 100 (le zéro sera la position la plus basse du curseur verticalement).

Pour comprendre l'intérêt des propriétés de dépendance nous ajouterons une petite animation de la propriété ainsi qu'un moyen plus conventionnel d'en changer la valeur (avec un *slider* posé sur la fiche).

Créer un user control WPF

Je ne passerai pas trop de temps sur cet aspect des choses mais pour le plaisir de faire de la propagande pour Expression Blend je vais ajouter quelques copies d'écran assorties d'explications…

Dans Blend, donc, on ajoute un nouvel item au projet de démonstration utilisé jusqu'ici. On choisit un `UserControl` dans la liste et on lui donne le nom de `jauge.xaml` :

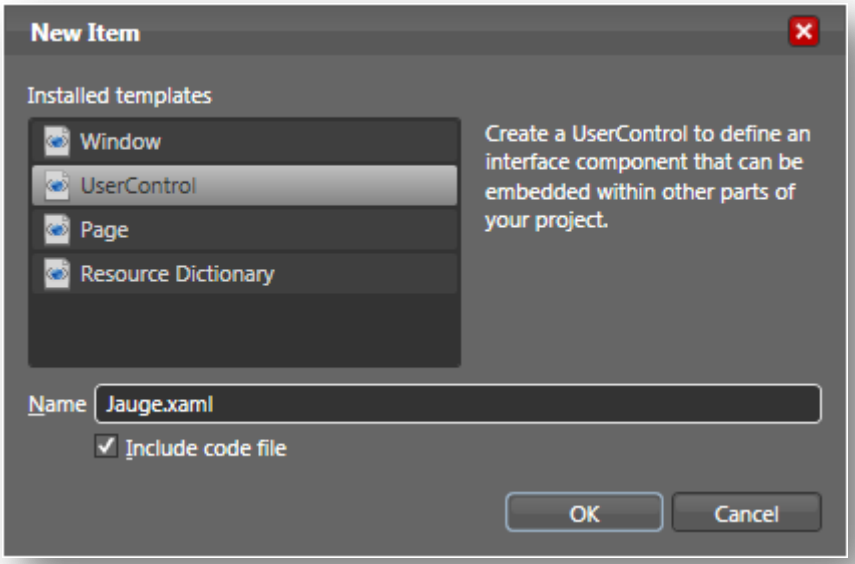

Figure 3 - Création d'un UserControl Jauge

Il ne reste plus qu'à créer le visuel de ce composant :

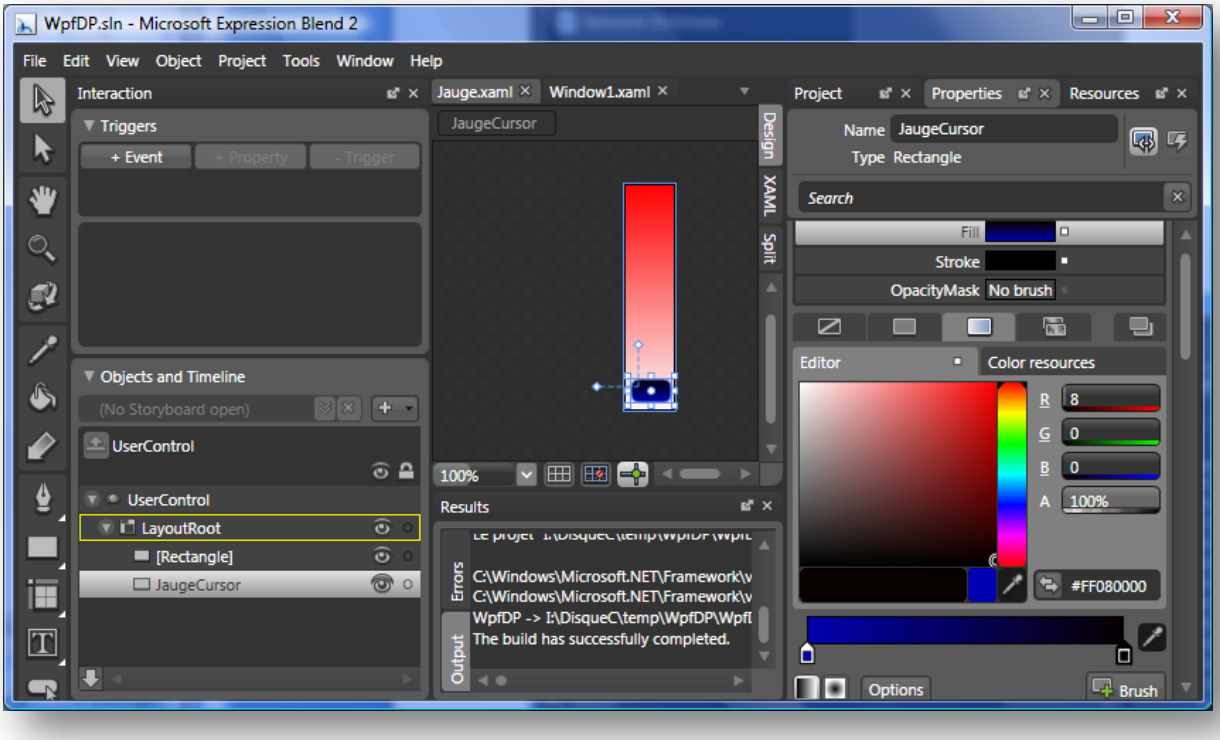

Figure 4 - La conception visuelle du contrôle

Sous Expression Blend tout cela se fait très vite et le code XAML correspondant est écrit automatiquement:

```
<UserControl
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
      mc:Ignorable="d"
      x:Class="WpfDP.Jauge"
      x:Name="UserControl"
      d:DesignWidth="36" Height="160" Width="36">

      <Canvas x:Name="LayouRoot" Width="Auto" Height="Auto">
            <Rectangle Stroke="#FF000000" Width="36" Height="160">
                  <Rectangle.Fill>
                        <LinearGradientBrush EndPoint="0.5,0"
StartPoint="0.5,1">
                              <GradientStop Color="#FFFBF8F8" Offset="0"/>
                              <GradientStop Color="#FFFF0000" Offset="1"/>
                        </LinearGradientBrush>
                  </Rectangle.Fill>
            </Rectangle>
            <Rectangle Stroke="#FF000000" Width="28" Height="16"
Canvas.Left="5" Canvas.Top="138" x:Name="JaugeCursor" StrokeThickness="0"
RadiusX="5" RadiusY="5">
                  <Rectangle.Fill>
                        <LinearGradientBrush EndPoint="0.5,0"
StartPoint="0.5,1">
                              <GradientStop Color="#FF0100AF" Offset="0"/>
```

```
                                <GradientStop Color="#FF080000" Offset="1"/>
                            </LinearGradientBrush>
                        </Rectangle.Fill>
                    </Rectangle>
            </Canvas>
</UserControl>
```

Heureusement qu'avec Blend je n'ai pas eu à taper tout cela à la main ! Mais ce qui nous intéresse le plus dans cet article ce sont les propriétés de dépendance, pas les petits dessins, alors passons vite au code C# du composant !

Le code du contrôle

Lorsque le contrôle XAML a été créé, deux fichiers ont été préparés par Expression Blend (sous Visual Studio cela aurait été identique), le fichier XAML lui-même qui décrit le visuel et un fichier de code C# permettant d'ajouter la logique du contrôle. Nous trouvons ainsi dans notre projet `Jauge.xaml` et `Jauge.xaml.cs`.

DependencyProperty

La déclaration d'une propriété de dépendance passe par la création d'un champ statique et public portant le nom de la propriété suivi du suffixe `Property`. Ce champ est une instance de la classe `DependencyProperty` qui encapsule tout les comportements nécessaires.

La création de l'instance s'effectue par l'appel à la méthode statique `Register` de la classe `DependencyProperty`. Il existe de nombreuses variantes de cette méthode selon qu'on veuille ou non préciser certains comportements.

Dans notre exemple nous utiliserons une version assez complète que voici :

```
public static readonly DependencyProperty LevelProperty =
 DependencyProperty.Register("Level", typeof (int),
                             typeof (Jauge),
                             new PropertyMetadata(0,
                         new PropertyChangedCallback(jaugePropertyChanged)),
                             new ValidateValueCallback(IsJaugeValid));
```

La propriété s'appelle "`Level`", le nom du champ public est ainsi "`LevelProperty`".

Les paramètres de la méthode `Register` que nous utilisons sont, dans l'ordre :

- « `Level` » le nom de la propriété tel qu'il peut être manipulé en XAML
- Le type de cette propriété (un entier)
- Le type du composant concerné (la classe `Jauge` qui est notre contrôle)
- La définition des métadonnées qui permettent dans cette version :
    - De fixer la valeur par défaut (zéro)
    - De fixer le gestionnaire du callback de changement de valeur (ici la méthode `jaugePropertyChanged`)
- La définition du gestionnaire de callback de validation (la méthode `IsJaugeValid`).

Comme on peut le voir en une instruction nous fixons de très nombreuses options. Mais il reste encore un peu de code à ajouter pour terminer la définition de la propriété. Notamment

il nous faut créer la propriété CLR qui jouera le rôle de « proxy » vis-à-vis de la propriété de dépendance créée par le code ci-dessus :

```
public int Level
{
    set {SetValue(LevelProperty,value);}
    get { return (int) GetValue(LevelProperty); }
}
```

La propriété CLR ne doit rien faire d'autre que d'appeler `SetValue` dans son *setter* et de retourner le résultat de `GetValue` dans son *getter*. Aucune autre logique ne doit être ajoutée ici car rien ne dit que la valeur ne sera pas changée en passant directement par la propriété de dépendance qui se cache derrière (une animation, du code XAML…). La propriété CLR n'est donc qu'une façon de faire apparaître la propriété de dépendance comme une propriété standard, c'est une *guideline*, une convention pratique. Intrinsèquement la propriété de dépendance n'a pas besoin de cette définition supplémentaire pour exister et fonctionner depuis le code XAML. Mais pour manipuler la propriété par code C# il est plus pratique d'exposer un proxy, c'est tout.

A partir de là la propriété est totalement déclarée. Des valeurs peuvent être saisies et lues. Mais il n'y a aucun effet derrière ces valeurs et c'est un peu dommage ! Il nous faut donc programmer ce qu'il se passe lorsque la valeur change.

Le callback de notification de changement de valeur

La propriété de dépendance peut être modifiée soit via le proxy CLR que nous avons déclaré soit via code XAML ou des services WPF comme une animation. Il n'est donc pas question de programmer la partie visuelle dans le setter du proxy qui sera ignoré par ces derniers. C'est donc grâce au mécanisme de callback de notification de changement de valeur de la propriété de dépendance qu'il faut agir.

C'est tout l'intérêt d'avoir ajouté ce callback dans les métadonnées de la propriété. Ne reste plus qu'à le programmer :

```
private static void jaugePropertyChanged(DependencyObject obj,
                                         DependencyPropertyChangedEventArgs e)
{
  Jauge j = (Jauge) obj;
  int i = (int) e.NewValue;
  if (i>=0 && i<=100)
  j.JaugeCursor.SetValue( Canvas.TopProperty,
                    5+ (j.Height-10-j.JaugeCursor.Height)/100.0*(100-i));
}
```

Dans le corps du callback nous récupérons l'objet Jauge concerné puis la nouvelle valeur de la propriété. Si cette valeur est convenable nous déplaçons le petit curseur (un rectangle coloré) en modifiant la propriété jointe `Top` du `Canvas` sur lequel il est posé.

Je passe sur le calcul de position qui n'a guère d'intérêt ici.

Le callback de validation de la valeur

Lorsque nous avons enregistré la propriété de dépendance nous avons déclaré un callback de validation. En voici le code :

```
private static bool IsJaugeValid(object value)
{
  try
  {
     int v = (int) value;
     return (v >= 0 && v <= 100);
  }
  catch
  {
    return false;
  }
 }
```

Tout comme le callback précédent, la validation de valeur est une méthode statique. Toutefois si pour le callback de changement de valeur WPF nous passe la référence de l'objet concerné, il n'en est rien pour celui de validation. Moralité il faut se contenter de tester la valeur « brute », hors contexte.

Ici nous pouvons programmer la méthode pour qu'elle rejette toutes les valeurs sous zéro et au-delà de 100, c'est un comportement figé qui ne dépend pas de l'état de la jauge ou d'autre chose. C'est une limitation « par design », la valeur doit être comprise entre 0 et 100.

Il existe d'autres cas, assez fréquents, où tester la validité d'une propriété n'a de sens que pris dans le contexte de l'état global de l'objet, voire d'autres objets avec lesquels il entre en relation. Par exemple, si notre jauge définissait des propriétés de type mini et maxi pour la valeur du curseur, un peu comme un `slider`, tester la valeur si la valeur courante est valide ou non réclamerait de connaître la valeur du mini et du maxi. Au sein du callback de validation cela ne serait pas possible.

Néanmoins il existe une autre façon de valider le contenu d'une propriété de dépendance qui offre, en outre, la possibilité de changer la valeur proposée avant qu'elle ne soit mémorisée. Ce callback s'appelle `CoerceValueCallback` et c'est aussi un champ de l'objet métadonnées de la propriété (`PropertyMetaData` dont nous créons une instance lors de l'appel à `Register`, voir plus haut le code de création de la propriété de dépendance).

En programmant ce callback plutôt que celui de validation il est ainsi possible de tester la valeur proposée dans le contexte de l'instance de l'objet et de rejeter les valeurs inappropriées voire même de modifier d'autres propriétés de l'objet (par exemple si la propriété mini devient supérieure à la valeur de la propriété maxi plutôt que de lever une exception il est préférable de changer la valeur maxi pour qu'elle soit de nouveau valide).

Côté implémentation `CoerceValueCallback` ne présente aucune difficulté, je vous laisse le plaisir de compléter le code de démonstration pour vous faire la main !

Utiliser une propriété de dépendance

Même si nous n'avons pas écrit beaucoup de code, définir une propriété de dépendance réclame malgré tout un petit plus de travail qu'une propriété CLR. Mais cet effort va être récompensé !

Fixer des valeurs

La première chose à faire est de tester le comportement de notre jauge. Pour cela nous allons placer le nouveau composant sur la fiche qui nous a servi de démonstration jusqu'ici. Première constatation, sous Expression Blend (ci-dessous), dans les propriétés du contrôle nous voyons bien apparaître « `Level` ».

A la main

Il est alors possible de taper diverses valeurs pour voir si le visuel suit, donc si le callback de changement de valeur est bien appelé (et accessoirement si notre code fait bien ce qu'on attend).

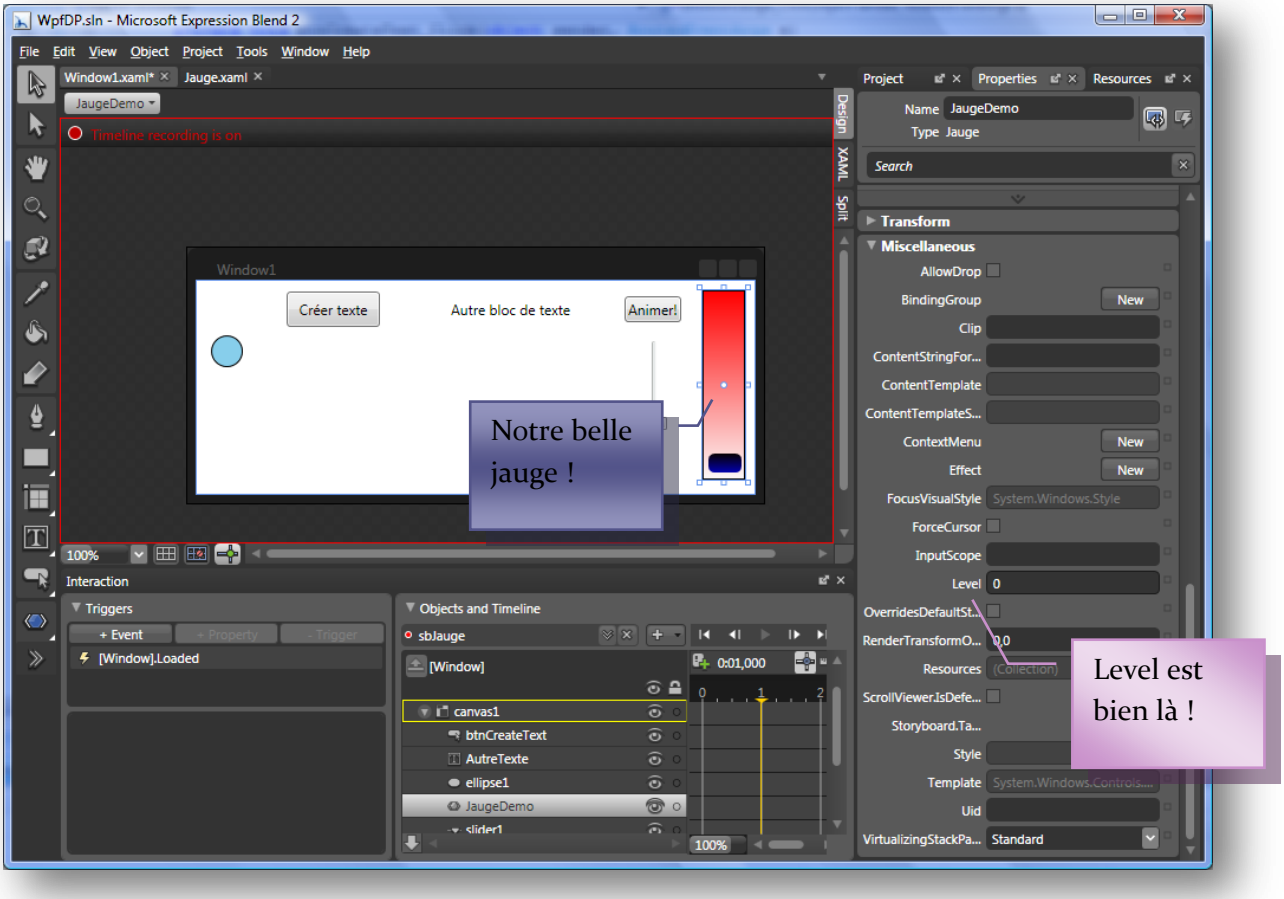

Figure 5 - La propriété de notre contrôle sous Blend

En XAML

Nous pouvons aussi voir sous Visual Studio, dans le code XAML de la fenêtre de la démo, si nous avons bien accès à la propriété `Level` (figure ci-dessous) :

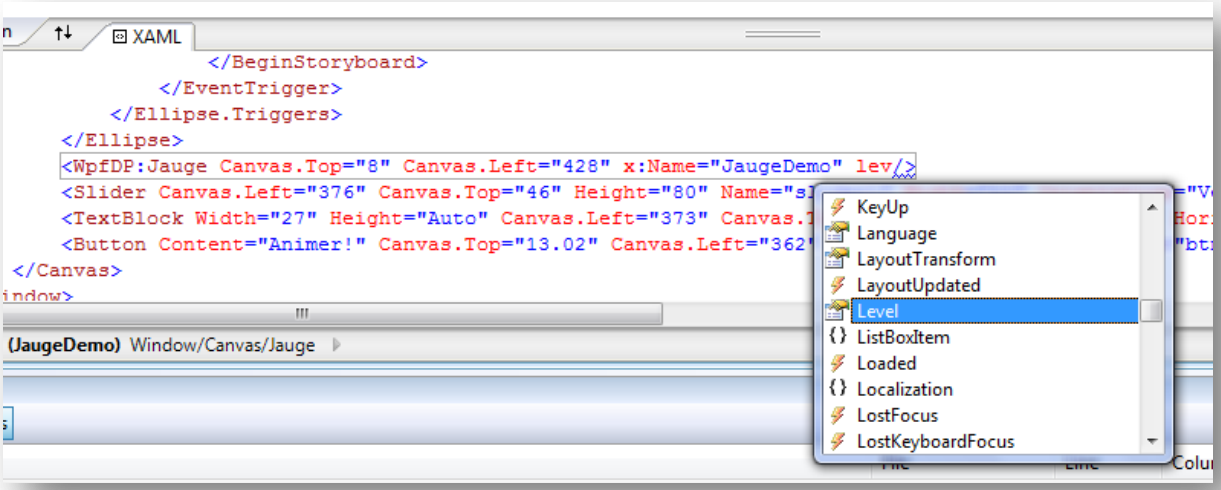

Figure 6 - La propriété manipulée en XAML sous VS

Par Data Binding

Nous pouvons aussi ajouter un `Slider` afin de faire varier la valeur lorsque l'application de démonstration fonctionne.

Le premier réflexe serait d'ajouter un gestionnaire pour l'événement `ValueChanged` du `Slider` et d'écrire un code de ce genre :

```
private void slider1_ValueChanged(object sender,
                                  RoutedPropertyChangedEventArgs<double> e)
{     JaugeDemo.Level = (int)slider1.Value;
      jaugeValue.Text = JaugeDemo.Level.ToString();
}
```

Au passage vous noterez que j'ai ajouté un `TextBlock` sous le `Slider` pour afficher la valeur de la jauge.

Ce code C# est très beau, il fonctionne, mais franchement ça c'est de la programmation « à la Windows Forms », j'allais dire « à la papa » ! Il faut savoir changer sa façon de penser avec les outils. En réalité en tapant un tel code nous sommes aussi ringards et incultes qu'une secrétaire débutante qui essaye de mettre en page un texte sous Word en créant des indentations avec la barre d'espace au lieu d'utiliser des taquets de tabulation… C'est dire si on peut passer pour un *hasbeen* à la machine à café pour le restant de ses jours ! ☺

Non, oublions C# quelques minutes et regardons comment nous pouvons obtenir le même résultat en tirant profit du fait que la propriété `Level` de notre jauge a été déclarée comme une propriété de dépendance qui, notamment, accepte le Data Binding :

```
<WpfDP:Jauge Canvas.Top="8" Canvas.Left="428" x:Name="JaugeDemo"
Level="{Binding ElementName=slider1, Path=Value}"/>
```

Ci-dessus le code de définition de la jauge dans la fenêtre. On peut voir comment, via Data Binding, nous affectons dynamiquement, et sans code C#, la valeur courante de la propriété `Value` du `Slider` à la propriété `Level` de notre jauge. Magique non ?

Ne reste plus qu'à faire la même chose pour le `TextBlock` qui affiche la valeur. Dans un tel cas, comme nous recevons une valeur décimale il faudra ajouter un paramètre `StringFormat` au binding. Mais plutôt que de brancher le `TextBlock` sur le `Slider`, nous allons plutôt le connecter à la propriété `Level` de la jauge. Comme cela nous aurons bien le retour de ce que cette dernière enregistre plutôt qu'une copie de la valeur du `Slider`… Le code de definition du `TextBlock` devient :

```
<TextBlock Width="37" Height="15" Canvas.Left="368" Canvas.Top="134"
TextWrapping="NoWrap" HorizontalAlignment="Stretch" TextAlignment="Center"
x:Name="jaugeValue" Text="{Binding ElementName=JaugeDemo, Path=Level,
StringFormat=##0}"/>
```

J'ai volontairement laissé le paramétrage du `StringFormat` pour que vous puissiez voir comment cela se présente. Mais en se branchant sur le `Level` de la jauge qui un entier nous n'avons pas besoin de cet artifice.

Le format PDF a ses limites et il devient difficile de faire des copies d'écran montrant tout cela à l'œuvre ! C'est pourquoi le projet exemple est livré dans le Zip de l'article, exécutez-le pour voir comment se comporte l'application visuellement.

Animer la propriété

Il est intéressant, pour compléter le tour d'horizon, de voir comment notre jauge peut être animée. A la fois pour le côté ludique et pour le plaisir d'utiliser encore une fois Expression Blend, mais aussi pour voir comment tout le mécanisme des propriétés de dépendance fonctionne lorsque plusieurs sources sont connectées à la même propriété.

En effet, la propriété `Level` de la jauge possède une valeur par défaut (zéro) fixée lors de la création de la propriété, mais elle possède aussi une valeur fixée à la main dans Blend ou Visual Studio, en plus elle est liée par Data Binding au `Slider`. Si nous rajoutons une animation que se passe-t-il ?

Grâce au système de précédence, c'est la valeur du Data Binding qui a le dessus pour le moment. Il suffit de bouger le `Slider` pour s'en apercevoir, le curseur de la jauge bouge… En ajoutant une animation (qui sera lancée par un bouton) ce sont les valeurs fournies par cette dernière qui vont prendre le contrôle de `Level`. Mais quand l'animation va s'arrêter, dans quel état sera `Level` ? Classiquement la valeur resterait à la dernière position envoyée par l'animation, c'est ce qui arriverait dans le modèle de programmation classique sous Windows Forms par exemple. Mais ici, quand l'animation va s'arrêter elle va cesser de contrôler la valeur de `Level`, la précédence reviendra donc au Data Binding et le curseur va se repositionner tout seul à la bonne place… C'est là qu'on comprend toute la richesse des propriétés de dépendance.

Bien entendu, pas plus que tout à l'heure le PDF de l'article ne peut vous faire voir l'effet que je viens de décrire. Une seule solution, exécutez le projet exemple et constatez par vous-mêmes !

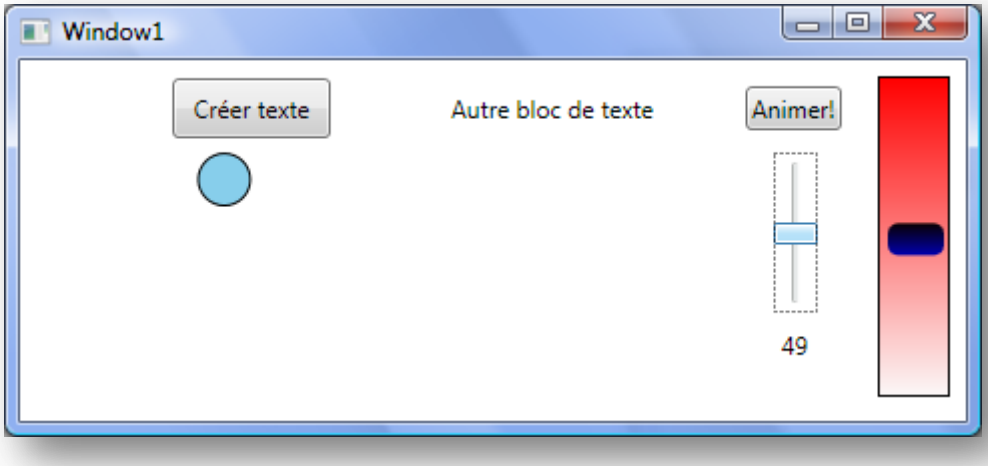

Figure 7 - L'animation de la jauge

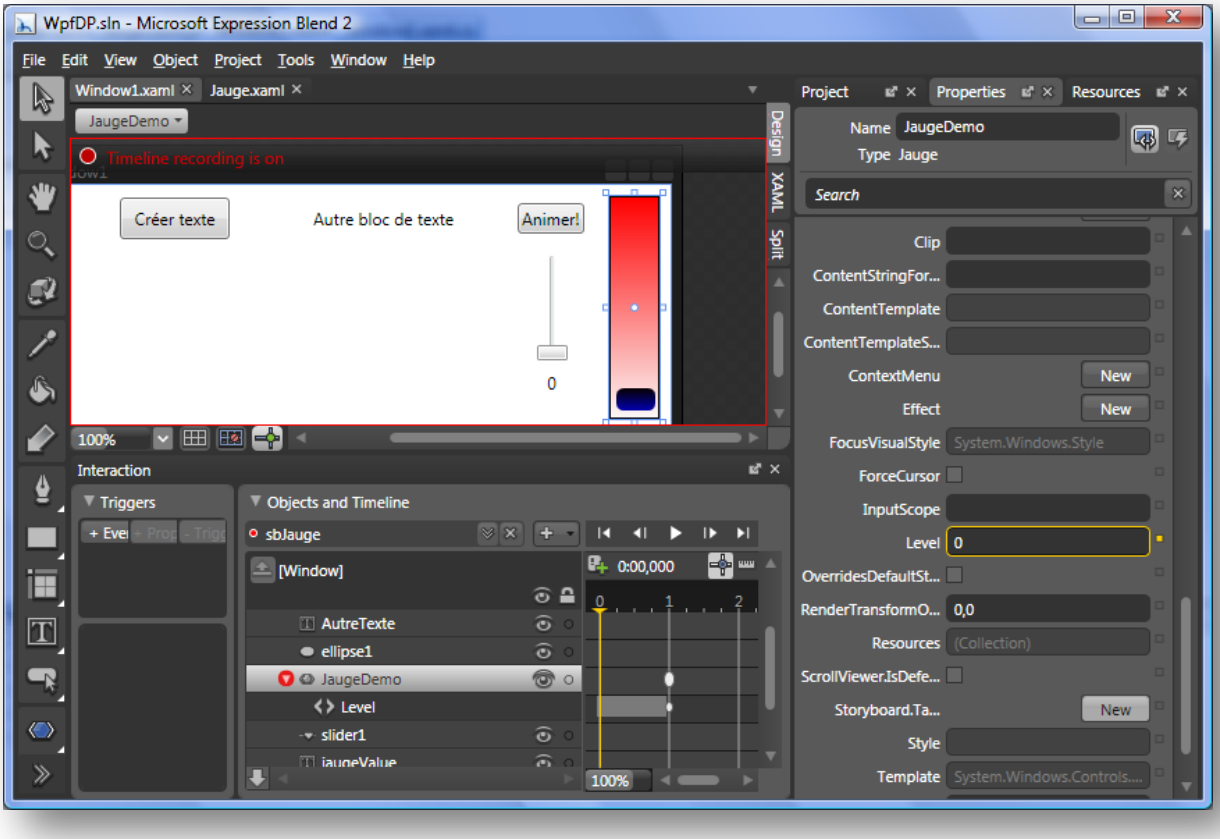

Figure 8 - Création de l'animation sous Blend

L'animation est lancée sur le clic du bouton « Animer ! ». Le code se présente comme suit :

```
private void btnAnimJauge_Click(object sender, RoutedEventArgs e)
{
    var sb = ((Storyboard) FindResource("sbJauge"));
    sb.Completed += new EventHandler(sb_Completed);
```

```
            sb.Begin();
        }

        void sb_Completed(object sender, EventArgs e)
        {
            var sb = ((Storyboard)FindResource("sbJauge"));
            sb.Stop();
            sb.Completed -= sb_Completed;
        }
```

Le code du gestionnaire de clic du bouton commence par localiser le `Storyboard` « `sbJauge` » dans les ressources de la fiche. C'est là que l'animation a été rangée par Blend. Une fois l'objet récupéré nous lui assignons un gestionnaire pour son événement `Completed` qui sera déclenché dès que l'animation sera terminée. Ensuite nous jouons l'animation avec `Begin`.

Le code de `Completed` est tout aussi simple. Il récupère l'objet (j'ai fait un copier/coller du code précédent par paresse, j'aurai pu, dû ?, récupérer l'objet via le paramètre `sender`). Une fois le `Storyboard` localisé, sa méthode `Stop` est appelée. Cela met fin au contrôle qu'exerce l'animation sur la propriété `Level`. C'est à ce moment que le curseur revient à sa dernière position. Vous pouvez mettre ce comportement en évidence en plaçant un point d'arrêt sur la ligne appelant `Stop`.

Tout cela est à tester visuellement pour se rendre compte du mécanisme.

La définition des propriétés jointes

Cet article est déjà bien long et il est temps de conclure, mais avant cela il semble indispensable de dire un mot sur la façon de créer des propriétés jointes dont nous avons parlé plusieurs fois.

Les propriétés jointes sont des propriétés de dépendance un peu particulières mais elles se comportent de la même façon et bénéficie des mêmes services.

Il existe toutefois quelques nuances dans la façon de les déclarer.

Enregistrement

Une propriété jointe se déclare de la même façon qu'une propriété de dépendance, en revanche on utilise `RegisterAttached` au lieu de `Register` (extrait du code du conteneur `Grid` WPF) :

```
FrameworkPropertyMetadata metadata =
   new FrameworkPropertyMetadata(0,
       new PropertyChangedCallback(Grid.OnCellAttachedPropertyChanged));

Grid.RowProperty =
   DependencyProperty.RegisterAttached("Row",
                                       typeof(int),
                                       typeof(Grid),
                                       metadata,
                            new ValidateValueCallback(Grid.IsIntValueNotNegative));
```

Setter et Getter statiques

L'autre différence avec une propriété de dépendance classique est qu'une propriété jointe ne déclare pas de propriété CLR proxy. En effet, une telle propriété n'aurait pas de sens puisque ce n'est pas la classe qui déclare la propriété qui va en être utilisatrice (comme dans le cas de notre jauge) mais d'autres objets dont on ne sait rien à l'avance (c'est par exemple une instance de `TextBlock` qui utilisera `Top` et `Left` d'un `Canvas` si elle est placée sur un conteneur de ce type).

Dans la première partie de cet article nous avons longuement montré l'utilisation des propriétés jointes (voir « Utiliser une propriété jointe ») et, si vous avez bonne mémoire, vous vous souvenez qu'il y a plusieurs façon d'exploiter ces propriétés, soit par code C#, soit directement en XAML. Dans ce dernier cas nous avons vu que nous utilisions directement le nom de la propriété tel qu'il apparaît dans l'appel à `Register`. Mais en C# nous avons vu que nous utilisions des méthodes statiques de la classe déclarant la propriété (rappelez vous de `Canvas.SetTop` par exemple).

Une propriété jointe ne déclare donc pas de propriété CLR proxy mais un couple de méthodes statiques dont le nom, par convention est `Get<nom>` et `Set<nom>`.

Toujours en piochant dans le code de la `Grid` WPF cela donne pour la propriété jointe `Row` (voir la déclaration à la section précédente ci-avant) :

```
public static int GetRow(UIElement element)
{
  if (element == null)
  { throw new ArgumentNullException(...); }
  return (int)element.GetValue(Grid.RowProperty);
}

public static void SetRow(UIElement element, int value)
{
   if (element == null)
   { throw new ArgumentNullException(...); }
   element.SetValue(Grid.RowProperty, value);
}
```

Partager une propriété de dépendance

Un certain nombre de classes partagent les mêmes propriétés de dépendance. Sous Windows Forms le phénomène est très courant pour les composants visuels : docking, fonte, position `Top` et `Left`, etc. Dans ce modèle le partage des propriétés est réalisé tout naturellement par héritage d'une classe de base contenant le socle commun.

Dans le modèle WPF qui est beaucoup libre, et comme nous l'avons déjà dit dans cet article, une telle pratique n'est pas réaliste. D'où la présence des propriétés de dépendance. Toutefois comment réutiliser une même propriété au sein de plusieurs classes qui n'ont pas forcément d'ancêtre commun ?

Prenons l'exemple de l'objet `TextBlock` qui expose une propriété `FontFamily` tout comme la classe `Control`. `TextBlock` ne descend pas de `Control` mais de `FrameworkElement`, comme `Control`. L'ancêtre commun est bien trop haut dans l'arborescence pour lui ajouter de façon classique la propriété `FontFamily` qui serait alors propagée à de nombreuses classes au sein desquelles elle n'aurait aucun intérêt (par exemple `Ellipse`, `Line`…).

La propriété `FontFamily` est en revanche déclarée dans la classe `TextElement` par le procédé que nous avons vu plus haut dans l'article. Voici comment la classe `TextBlock` « vampirise » la propriété (le code suivant est placé cette fois-ci dans le constructeur statique de la classe `TextBlock`) :

```
TextBlock.FontFamilyProperty =
TextElement.FontFmamilyProperty.AddOwner(typeof(TextBlock));
```

Il existe des surcharges de la méthode `AddOwner` permettant de modifier les métadonnées dont principalement les callback (validation, `Coerce`…) ce qui laisse une marge de manœuvre pour adapter la propriété au contexte local.

Attention toutefois aux effets de bords ! En ajoutant un style touchant la `FontFamily` sur les `TextBlock` tous les contrôles dérivant de `Control` se verront modifiés puisqu'en réalité `Control` et `TextBlock` exploitent la même propriété de dépendance provenant de `TextElement`…

Précédence et évaluation

Nous avons évoqué dans cet article que les valeurs des propriétés de dépendance (et propriétés jointes) étaient déterminées par un moteur prenant en compte les diverses sources possibles en appliquant des règles de précédence.

Il est important de connaître ces dernières lorsqu'on conçoit une application pour être en mesure de s'imaginer comment celle-ci va se comporter une fois que des styles, des valeurs directes, du Data Binding ou des animations vont être appliqués aux objets.

Voici comment WPF décide de la valeur de base d'une propriété de dépendance (par ordre décroissant de priorité) :

1) Valeur locale (si vous fixez directement une valeur par code C# ou XAML)
2) Valeur provenant d'un style du projet
3) Valeur provenant d'un style issu d'un thème
4) Valeur héritée (si le drapeau `FrameworkPropertyMetadata.Inherits` a été placé à `True` et qu'une valeur a été fixée plus haut dans l'arbre visuel des objets)
5) Valeur par défaut de la propriété (si elle a été fixée à l'enregistrement de cette dernière)

Cette chaîne ne permet que de calculer la valeur de base, WPF applique ensuite d'autres règles pour décider de la valeur finale… l'ordre dans lequel ce processus s'effectue est le suivant :

1) Déterminer la valeur de base (voir ci-dessus)
2) Si la valeur est initialisée par le biais d'une expression, évaluer celle-ci (principalement les ressources et le Data Binding)
3) Si la valeur est animée, appliquer la valeur calculée par l'animation

4) Exécution de l'éventuel `CoerceValueCallback` pour corriger la valeur si besoin est
5) Exécution de l'éventuel `PropertyChangedCallback` pour interdire les valeurs non valides

Comme on le voit ici, les propriétés de dépendance ont une mise en œuvre interne assez complexe. Si les propriétés de dépendance n'avaient pas été ajoutées à WPF pour encapsuler tous les comportements que nous avons étudié ici, la programmation WPF avec des propriétés CLR serait un enfer…

Microsoft se laisse d'ailleurs la porte ouverte pour complexifier encore plus le pipe-line et ajouter, si besoin, d'autres services autour des propriétés de dépendance.

Conclusion

Près de vingt-cinq pages pour parler des propriétés de dépendance ce n'est pas rien malgré tout ! J'espère seulement qu'arrivés au terme de ce périple vous avez compris tout l'intérêt de ces nouveaux compagnons car nul besoin de préciser que toute classe, tout composant développé pour WPF et Silverlight doivent l'être en utilisant des propriétés de dépendances et non de simples propriétés CLR…

Je remercie les plus courageux d'entre vous, ceux qui auront eu la ténacité nécessaire pour arriver à cette dernière ligne !

Thanks,

Et … Stay Tuned !

Olivier Dahan

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

Cet article est accompagné du code source de tous les projets utilisés. Ce code peut être diffusé librement avec l'article. Toute diffusion ou exploitation du présent article ne peut être faite sans l'accord de l'auteur.

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