Tome 7

XAML

Olivier Dahan

Collection
ALL DOT BLOG

www.e-naxos.com

Formation – Audit – Conseil – Développement

XAML (Windows Store, WPF, Silverlight, Windows Phone), C#

Cross-plateforme Windows / Android / iOS - Xamarin

UX Design

ALL DOT.BLOG

Tome 7

XAML – Tout Profil

Tout Dot.Blog mis à jour par thème sous la forme de livres PDF gratuits

Olivier Dahan
odahan@gmail.com

Table des matières

Table des Figures

Table des exemples XAML

Table des exemples de Code

Présentation du Tome 7 - XAML

Bien qu'issu principalement des billets et articles écrits sur Dot.Blog au fil du temps, le contenu de ce PDF a entièrement été réactualisé et réorganisé lors de la création du présent livre à l'automne 2013. Il s'agit d'une version inédite corrigée et à jour qui tient compte des évolutions les plus récentes autour de XAML, apparition et fin de Silverlight, naissance de WinRT, continuité de WPF, variations entre Windows Phone 7 et Windows Phone 8. Bien que largement inspiré des billets de Dot.Blog le présent Tome est celui qui a demandé le plus de travail à la fois par sa taille et par la réorganisation des thèmes, la réécriture et la mise en conformité des informations pour en faire non pas un recueil de textes morts mais bien un *nouveau livre totalement d'actualité* ! Un énorme bonus par rapport au site Dot.Blog ! Plus d'un mois et demi de travail a été consacré à la réactualisation du contenu. Corrections du texte mais aussi des graphiques, des pourcentages des parts de marché évoquées le cas échéant, contrôle des liens, et même ajouts plus ou moins longs, c'est une véritable *édition spéciale différente* des textes originaux toujours présents dans le Blog !

C'est donc bien plus qu'un travail de collection - déjà long - des billets qui vous est proposé ici, c'est *un vrai nouveau livre sur XAML qui offre une vision à jour à sa date de parution*. Mieux, aucun livre français ni même américain ne traite de XAML sous un tel angle. Ce livre contient parfois d'autres mini livres, comme celui sur le Binding XAML. Aucun texte existant n'est allé si loin sur un tel sujet à ma connaissance et ce n'est qu'un exemple parmi… près de 440 pages A4 et 113 000 mots ! Le standard est d'environ 250 mots par pages pour un livre. C'est bien un livre gratuit d'environ 452 pages sans les illustrations (et elles sont nombreuses) qui vous est offert (donc plus proche de 500 pages) ! On considère qu'un roman comporte 40.000 mots, le présent livre équivaut ainsi à prresque 3 romans… Et ce n'est qu'un Tome parmi la dizaine publiée dans la série ALL DOT BLOG !

> *Astuce : cliquez les titres pour aller lire sur Dot.Blog l'article original et ses commentaires, vous pourrez d'ailleurs vous amuser à comparer les textes et prendre la mesure des modifications ! Tous les autres liens Web de ce PDF sont fonctionnels, n'hésitez pas à les utiliser (certains sont cachés, noms de classe du Framework par exemple, mais ils sont fonctionnels, promenez votre souris…).*

XAML

« eXtensible Application Markup Language »

Prononcer « *Zammel* »

XAML est un langage de description de mise en page utilisant un formalisme XML. Les balises indiquent les instances de classes à créer, les attributs fixant les paramètres de ces instances. XAML se lit de haut en bas, quand une balise en contient une autre la première est le parent visuel de l'instance créée par la seconde.

XAML a été créé pour le framework de création d'applications dites riches « WinFX » qui deviendra officiellement .NET 3.0 associé à WPF lors de la sortie de Windows Vista.

Cet OS mal aimé en raison de son expérience utilisateur assez lourde a fait quelque peu capoter l'entrée en scène triomphale de WPF et XAML, avancée pourtant extraordinaire puisque XAML est un système de description d'UI **entièrement vectoriel** en 2D et 3D gérant le data Binding et particulièrement bien taillé pour faire face au défi à la fois des **nouvelles interfaces**, de **l'importance de l'UX** et du **support de tous les form-factors** (le vectoriel étant sur ce point l'arme absolue).

Le désamour de Vista a terni ce qui aurait dû être un raz-de-marée tellement XAML et WPF associés à .NET bâtissent une plateforme puissante pour développer des logiciels modernes. Cela était vrai en 2006 à la sortie de WPF 3.0 et cela reste vrai avec WPF 4.5 sorti en 2012 à tel point que XAML lui-même est aujourd'hui intégré à l'OS au sein de WinRT sous Windows 8.x et suivants. Il se glisse même au cœur de Windows Phone. Une nouvelle version de WPF serait en préparation d'ailleurs.

WPF/E, comprendre WPF Everywhere - c'est-à-dire WPF Partout – sera plus connu sous le nom de Silverlight. Version allégée de WPF, Silverlight utilise le principe d'un plugin pour browser Internet qui offre une portabilité transparente du même code

sous PC et Mac. Les énormes avantages de Silverlight sur HTML font que sa popularité deviendra vite supérieure à WPF regardé avec un peu de méfiance à l'époque par beaucoup de DSI qui ne comprenaient pas alors le changement de paradigme qu'annonçait XAML pour les applications, même de bureau. En revanche la mode de Flash sur le Web rendra plus évidente l'adoption de Silverlight au sein des mêmes entreprises pour leurs Intra ou Extra Net.

Hélas Silverlight ne résistera pas à la fois à la haine de Jobs côté Apple pour Flash et les plugins de ce type, et à celle de C#/XAML par Sinofsky côté Microsoft. HTML 5 n'y est pour rien dans tout ça, car au moment où lui aussi croyait devenir universel ce sont les unités mobiles qui ont bouleversé le marché rendant les applications natives les vraies reines de ce nouveau monde à conquérir. Silverlight aurait pu être la technologie de ce dernier, Microsoft a décidé de la garder pour Windows Phone uniquement.

En disparaissant Silverlight laisse un grand vide car aucune technologie vectorielle de ce niveau n'est disponible pour le Web. Toutefois il aura participé à rendre plus populaire XAML.

La fin de Silverlight en tant que plugin universel a néanmoins laissé XAML se répandre sur Windows Phone et sur WinRT où il est le meilleur choix pour créer des applications sophistiquées.

Mieux, la fin de Silverlight pourrait bien marquer le retour de WPF !

Car si Windows Phone et WinRT utilisent XAML, il s'agit de plateformes dont le parc est encore très restreint à ce jour. Pour balayer large il n'existe que WPF.

Car WPF est la seule plateforme moderne permettant de développer des logiciels sans contrainte, sans les taxes du Windows Store et surtout pouvant fonctionner sur quasiment 100% du parc Windows.

Seul WPF peut couvrir les 31 % de Windows XP + les 4 % de Vista + les 47 % de Windows 7 + les 9 % de Windows 8.x, soit 91 % du parc Windows fin 2013.

Cela créée une situation bien particulière, Microsoft d'un côté axant toute sa communication sur un OS qui pèse moins de 10 % du parc plus d'un an après sa sortie, de l'autre des entreprises et des utilisateurs ayant des besoins réels en total déconnexion technique avec WinRT.

On comprend alors la frilosité des décideurs face à un OS qui a du mal à convaincre, même sur tablette ou smartphone, mis en place par un directeur qui a été mis à la porte (Sinofsky) sous les offices d'un PDG (Ballmer) qui lui-même a été poussé vers la sortie mais qui est toujours là (jusqu'en août 2014 au maximum)…

Quels seront les décisions du prochain PDG de Microsoft ? Quelles ruptures comme celle de Silverlight ou de Windows Phone 7 devront-elles être subies ? Personne n'en sait rien du tout. Et le doute engendre la peur légitime de se tromper. Et cette dernière engendre l'immobilisme.

Pendant ce temps (perdu en choix discutables et en luttes intestines chez Microsoft) Android a pris 80% du marché des unités mobiles, mais c'est une autre histoire (même si elle nous oblige à penser le développement en cross-plateforme, voir le Tome 5 de ALL DOT BLOG) !

La seule certitude qui émerge de tout cela est que sous environnement Microsoft XAML et C# restent des valeurs sûres pour couvrir 100% du parc installé. Que cela soit avec WPF pour atteindre 91% de ce parc, que cela soit WinRT pour toucher le marché spécifique ouvert par cette plateforme, ou bien même Windows Phone.

Au final, car il faut bien conclure cette présentation et laisser la place aux près de 500 pages A4 de ce livre gratuit, XAML est une technologie incontournable. C'est un survivant qui aura connu toutes les tempêtes, tous les vents contraires mais qui de simple librairie s'impose au fil du temps avec son tooling dont aucun autre OS ni aucune autre plateforme au monde n'est doté aujourd'hui.

XAML est une merveille. Grâce à lui vous pouvez développer des applications efficaces et vendeuses, vous démarquer de la concurrence, utiliser les méthodes de développement les plus modernes que cela soit sous WPF, WinRT ou Windows Phone.

Dans ce livre j'aborderai XAML sous l'angle pratique de WPF pour les raisons de couverture du parc évoquées plus haut et surtout parce qu'étant la technologie « mère » l'implémentation XAML de WPF est la plus complète et donc celle qui permet de montrer toute la puissance de XAML.

Le Tome 6 de ALL DOT BLOG « Windows 8.x et WinRT » aborde les spécificités de cette plateforme dont je ne parlerais pas ici. Le Tome 8 à venir abordera les spécificités de Silverlight. Le présent Tome 7 est celui qui traite de tout XAML et de

ses méandres. Ces trois livres sont conçus pour être lus indépendamment l'un de l'autre même si le Tome 7 pose les grandes bases de XAML.

On retrouvera d'ailleurs parfois ici des exemples utilisant aussi Silverlight en plus de WPF. La mort annoncée du plugin Web n'en fait pas moins une plateforme unique et donc irremplaçable au moins pour quelques années dans les entreprises pour les intra et extranets. Et l'enterrer si vite serait ignorer qu'il est à la base de Windows Phone et que même WinRT ressemble plus à de la programmation Silverlight qu'à du WPF.

Et puis XAML reste XAML. Sous toutes ses formes ! C'est XAML qui est présenté, et je dirai même fêté dans les pages qui suivent. Toutes ses implémentations présentent des intérêts particuliers, aucune n'est ni morte ni oubliée, toutes sont les différents facettes d'une seule et même technologie unique en son genre.

Si ce livre n'a pas de parti pris et qu'il considère tous les profils de XAML comme étant égaux en intérêt, de WPF à WinRT, côté langage j'assume une préférence exclusive pour C# comme langage de programmation.

Que les lecteurs qui préfèrent C++, JavaScript ou VB.NET n'y voient pas une attaque de leurs convictions ou de leur savoir-faire, tout choix est respectable lorsqu'il permet d'atteindre l'objectif de façon professionnelle. Mais mon choix c'est C#, et justement pour ne pas entrer en polémique stérile avec ceux qui en font un autre, je n'expliquerai pas cette préférence, débat qui serait hors sujet.

XAML est un fédérateur qu'on peut manipuler depuis différents langages, cette pluralité n'affecte en rien ce qui est possible de faire avec XAML. Et ce Tome lui est avant tout consacré, c'est lui la vedette des pages qui suivent. Le langage utilisé dans certains articles n'est qu'une illustration choisie parmi d'autres.

Bon développement en XAML et bonne lecture !

Olivier.

XAML – Les bases

Introduction à la première partie du livre

La première partie de ce livre est consacrée aux bases de XAML. Pour les mettre en évidence j'ai choisi d'utiliser le plus souvent WPF car c'est une plateforme merveilleuse, puissante et produisant des logiciels « stand alone » qui tournent aussi bien sous Windows 7 que Windows 8, ce que ne permet pas WinRT par exemple. Certains exemples sont en Silverlight car ce plugin permet de proposer des exemples vivants sur Dot.Blog pour illustrer les articles. Ces exemples peuvent être visualisés en accédant au billet original dans lequel il est intégré (par un clic sur le titre du chapitre concerné dans ce livre).

Toutefois, tout ce qui est mis en avant concerne principalement XAML lui-même. WPF, tout comme Silverlight, n'est que l'alliance de XAML pour l'affichage et de C# pour la partie code (ou de VB.NET). C'est-à-dire la même alliance qui permet de coder aussi sous Windows Phone et Windows 8.x ou sous WinRT pour Surface ou PC.

Les avantages et avancées démontrés sont ceux de XAML peu importe le nom qu'on donne au « package » qui l'enrobe. Qu'on parle de WPF ou de WinRT XAML est le même (à quelques nuances près). Il existe toutefois ce que Microsoft appelle des « profils », c'est-à-dire des sous-ensembles plus ou moins complets de XAML et du framework .NET.

Ainsi, WPF et .NET 4.5 aujourd'hui représentent le « maximum » de puissance que ce couple peut apporter. Alors que le « profil » Silverlight 5 possède un framework .NET un peu rétréci et un XAML avec quelques nuances. Il en va de même pour le profil « Windows Phone » ou le profil XAML de WinRT.

Tout ce qui est dit dans ce livre porte sur XAML, cette première partie tire profit de son profil « maximum » en utilisant souvent WPF et Silverlight de temps en temps. On peut transposer la plupart des exemples dans les autres profils XAML sauf lorsque ces derniers n'implémentent pas la fonction XAML présentée, bien entendu.

Enfin, les lecteurs intéressés pourront télécharger le Tome 6 de ALL DOT BLOG qui est entièrement consacré à WinRT et Windows 8.x, les exemples XAML étant alors directement adaptés à ce profil.

10 bonnes raisons de choisir XAML

J'ai écrit cet article en décembre 2008, il y a 5 ans. WPF était sorti depuis 2 ans. Mais finalement, et après quelques corrections qui s'imposaient – notamment des anachronismes et quelques ajouts – cet article reste totalement d'actualité… J'ai principalement déplacé le point de vue centré WPF vers un angle de vue centré XAML car ce dernier est finalement le sujet central, c'est XAML qu'il faut choisir, quelle que soit son profil.

XAML cet inconnu… Alors que cette technologie est disponible depuis 7 ans elle semble peiner à s'imposer parmi les développeurs. Je me suis demandé à l'époque pourquoi et j'ai cru deviner que WPF, première émanation de XAML, payait un peu son image du "*tout graphique hyper looké*" de ses premières présentations, des démos où l'on voyait des vidéos danser en l'air sous forme de carrousel, de pages qui se pliaient comme un livre pour passer d'une fiche à l'autre et autres débauches d'effets spéciaux. Une époque où tout cela se voyait dans Minority Report mais qu'un DSI avait du mal à projeter dans une application de gestion classique. Tout le monde se fichait du Design à cette époque et si les choses évoluent je ne vois guère de changement en pratique dans la façon dont les logiciels sont conçus 5 ans après en entreprise notamment.

En réalité le développeur "moyen" ne s'y retrouvait pas plus que son DSI et ne s'y retrouve toujours pas d'ailleurs dans une grande majorité de cas. Ce dernier, amateur des grilles fourre-tout pour qui Windows Forms est déjà presque de la science-fiction, n'a pas encore franchit l'étape nécessaire d'élévation de son âme de codeur pour comprendre la nécessité de tout cela et toucher au nirvana de la programmation moderne. Pourtant depuis 7 ans le monde de l'informatique à bien évolué, même votre concierge possède un smartphone où tout est question de look… Mais l'image renvoyée par les présentations de WPF a laissé une impression plus proche de celle du jeu vidéo que de l'informatique de gestion qui fait le gros des applications "de tous les jours". Et quand une première impression n'est pas la bonne, le codeur de base n'y revient que rarement pour se forger un nouvel avis plus contrasté. Je peux le dire avec d'autant plus de franchise que si vous lisez ces lignes c'est que vous faites partie de ceux qui veulent s'informer et que ces remarques ne vous concernent pas !

Je ne blâme pas ceux qui, par trop enthousiastes, ont pêché par excès en créant et en montrant de telles démonstrations. Après tout lorsqu'une nouvelle technologie vient de sortir on a par force envie de faire voir ce qu'elle sait faire de mieux, c'est naturel.

Non, je pense plutôt que c'est sur le plan psychologique que l'erreur a été commise, la même que pour Vista d'ailleurs. Un décalage trop fort, trop rapide, avec une devanture trop graphique qui a fait oublier que **derrière tout cela il y a une vraie révolution technique, de vraies avancées et surtout de vraies nécessités**. Le même décalage semble d'ailleurs troubler clients et programmeurs face aux changements encore plus radicaux que Windows 8.x imposent. Mêmes raisons, même résultats ?

J'ai donc eu envie d'écrire un article qui montre les grands points forts de WPF et surtout pourquoi cette technologie est de loin supérieure à toutes les autres, dont les Windows Forms encore trop utilisées alors qu'elles se fondent sur des mécanismes interactifs dépassés déjà en place du temps de Delphi 1 Win32 ! Il y a presque 20 ans...

Choisir *10 bonnes raisons d'utiliser XAML* a quelque chose d'arbitraire et je l'assume, surtout 5 ans après. Mais ce n'est pas grave car cet article se destine à tous ceux qui ne savent pas encore que XAML est **parfaitement taillé pour faire des applications "normales"**, à tous ceux qui pensent que ce n'est "pas fait pour eux" ou pour le style de programmes qu'ils écrivent. Il s'adresse aussi à **tous ceux qui ont envie de savoir** quels sont les points forts de WPF ou des autres profils XAML, les **nouveaux mécanismes** et la **nouvelle façon de penser les interfaces visuelles**.

En un peu plus de 40 pages inutile d'attendre un tour d'horizon complet et ultra technique de WPF que j'ai choisi pour illustrer mes propos, le moindre livre sur la question en compte 20 fois plus et le livre PDF que vous lisez actuellement compterait environ 800 pages en format livre papier sans pour autant faire entièrement le tour de la question... Mais si vous voulez rapidement faire un point sérieux sur XAML qui ne soit pas qu'un simple survol, si vous voulez voir du code mais pas trop, alors cet article est fait pour vous. Encore une fois quel que soit le profil XAML que vous utiliserez.

Vous pouvez télécharger l'article original pour récupérer le code source des exemples : 10 bonnes raisons de choisir WPF.

Introduction : Choisir XAML ?

> *« 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 gris*

hérités de Win32 ont fait plus que leur temps tout comme les fausses bordures de fenêtres 3D en relief bon marché. »

Cette réflexion date d'il y a cinq ans, cinq années de bouleversements où les smartphones et autres tablettes ont envahi le monde et ont prouvé la justesse de cette vision. Aujourd'hui si la puissance d'une application est toujours liée à son code, son pouvoir de séduction est totalement sous l'emprise du look & feel, des animations subtiles, des aplats de couleurs gaies, des effleurements de doigts. Ce qui fondait l'aspect des UI des ordinateurs de la science-fiction des années 2000/2010 est aujourd'hui une réalité de terrain qui fait vendre des centaines de millions d'ordinateurs miniaturisés qui tiennent dans le creux de la main… Bienheureux ceux sont qui m'ont écouté il y a longtemps, mais que les autres ne se sentent pas exclus, se ranger aux évidences, même un peu tard, est toujours préférable au déni de ceux qui deviendront au fil du temps les cobolistes de notre génération.

La seule plateforme graphique cohérente de l'instant, et cet instant dure depuis 7 ans, est WPF et ses dérivés comme le fut Silverlight ou l'est WinRT (dans son mode XAML). 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 totale et que nombreux sont les projets .NET encore écrits en Windows Forms au moment même où je révise 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. Depuis cinq ans WPF est entré un peu plus dans les mœurs des services de développement, mais pas assez encore.

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 comme Blend (aujourd'hui livré avec Visual Studio), 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 souvent 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 voulaient pas hier sautent sur

Windows 7 (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 ou aux autres plateformes.

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 miens 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. Je changerais aussi un peu l'ordre de sa présentation et proposerais certains de mes exemples plutôt que les siens.

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-cinq 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.

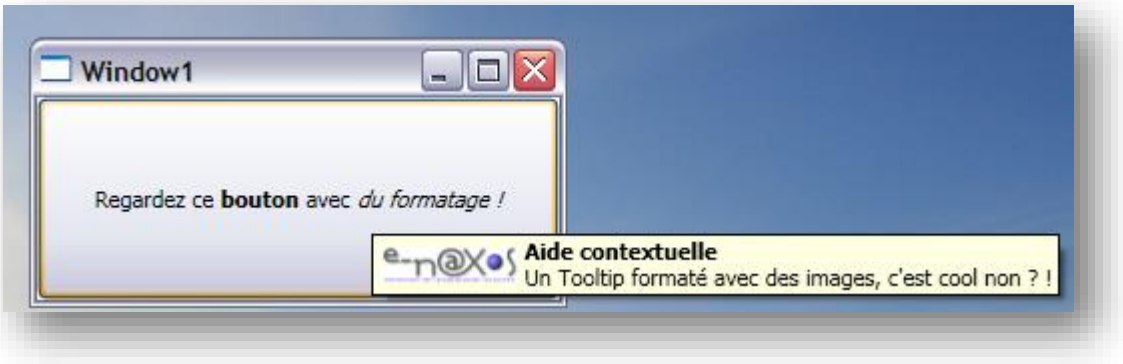


Figure 1 - Projet Wpf10-1

Sur cette illustration vous noterez la présente d'un bouton (occupant toute la fiche, c'est juste un exemple très moche) contenant un texte formaté (c'est déjà plus sophistiqué). Mieux, il dispose d'une aide contextuelle, elle aussi formatée et intégrant en plus une image. Je ne vous propose bien entendu pas ici une leçon de Design, je l'ai fait par ailleurs et vous trouverez de nombreux billets abordant ce sujet sur Dot.Blog. Ici je veux rester au plus près des possibilités offertes et non de leur mise en forme.

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>
```

Code 1 - Tooltip

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… Avec le recul cinq ans après j'assume totalement et j'affirme que cela est vrai et que l'avènement de MVVM est là pour le prouver !

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 :

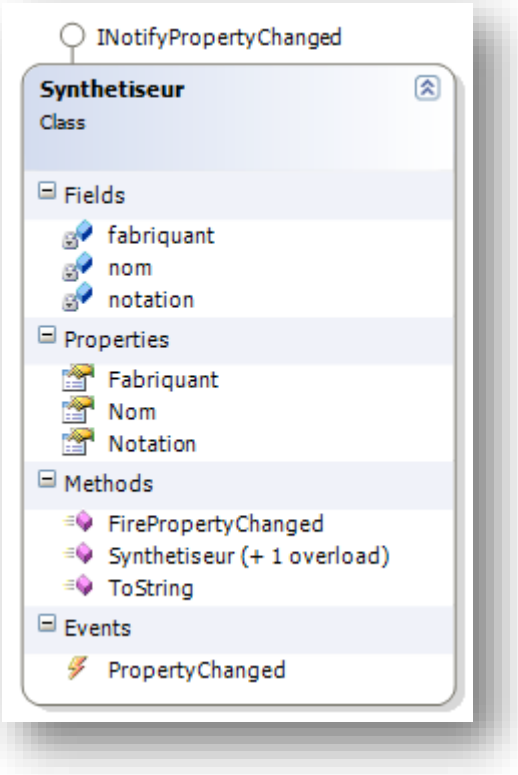


Figure 2 - La classe Synthétiseur

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}" />
```

XAML 1 - Binding simple

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),
                              …
```

Code 2 - Initialisation d'un DataContext

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" />
```

XAML 2 - Binding sur des listes

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 (cela rend le débogue plus facile).

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 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 :

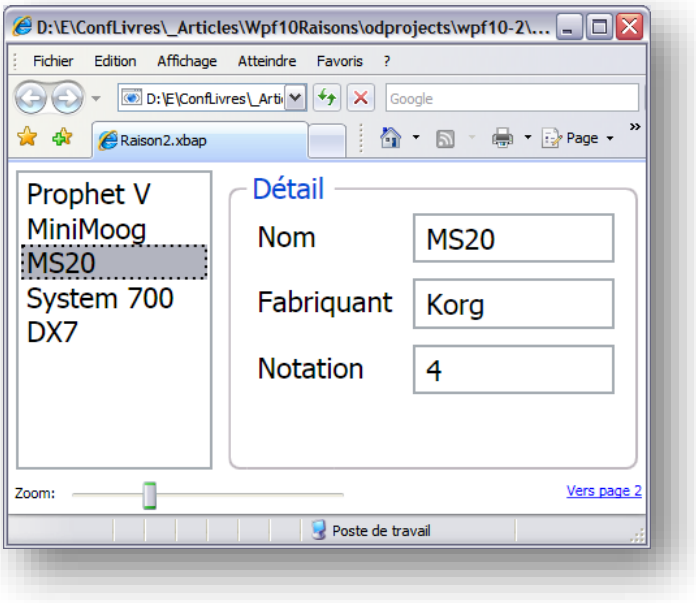


Figure 3 - Projet Wpf10-2

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, pour la démonstration de la versatilité de XAML.

La liste ainsi que les `Textbox` sont synchronisés automatiquement au `DataContext` de la page, cette liaison étant par à double sens (« two-way ») 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}" />
```

XAML 3 - Trigger de Binding

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 ce qui n'est généralement pas une bonne UX).

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>
```

XAML 4 - Element Binding

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 bas 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>
```

XAML 5 - Navigation

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 :

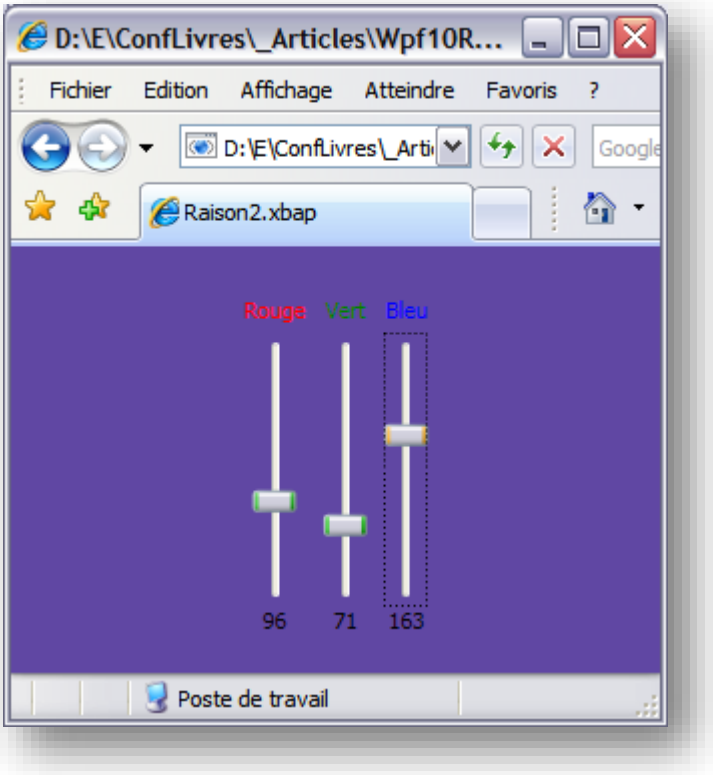


Figure 4 - La page 2 avec ses trois sliders

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>
```

XAML 6 - Multi-Binding

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>
```

XAML 7 - Instanciation de convertisseur

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 XAML (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"/>
```

XAML 8 - Spécification du champ dans un Binding

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 lorsque la ligne est survolée 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 9 - Item Template

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 Elysium, 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 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.

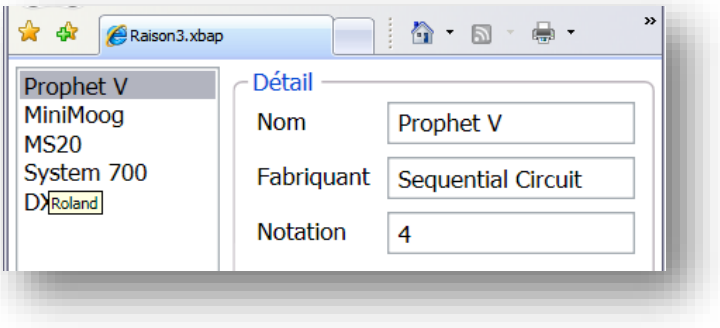


Figure 5 - Le Tooltip en action

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…

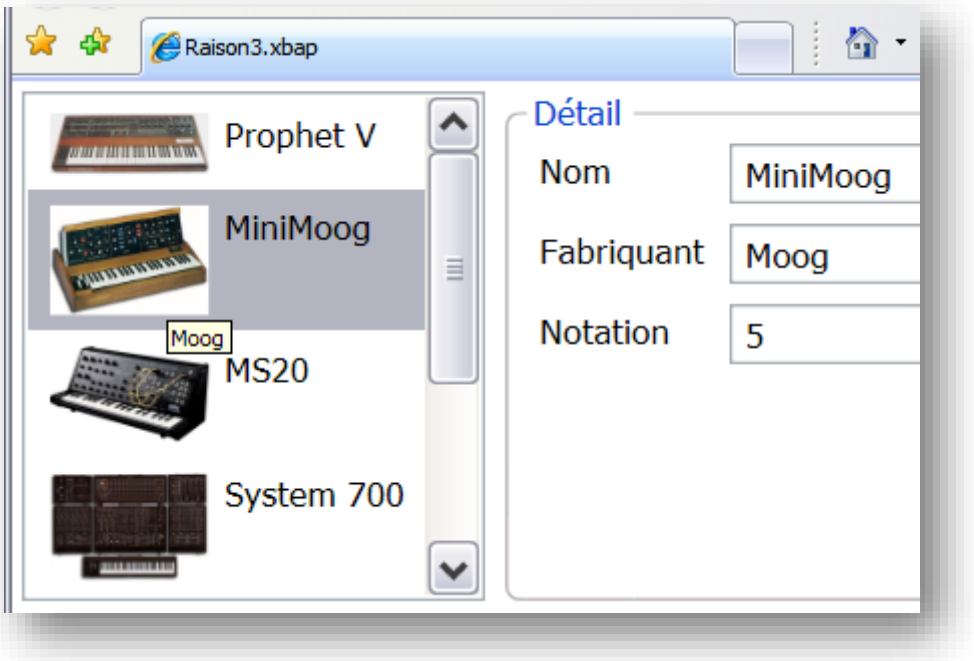


Figure 6 - Projet Wpf10-3

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>
```

XAML 10 - DataTemplate

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 XAML** 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 :

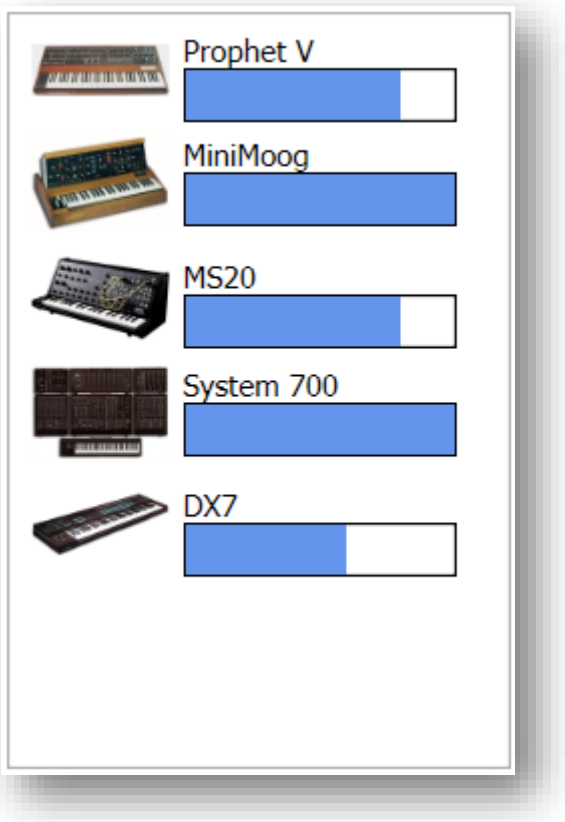


Figure 7 - Projet Wpf10-4

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>
```

XAML 11 - Détournement d'une barre de progression

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 **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 Blend et charger le projet exemple :

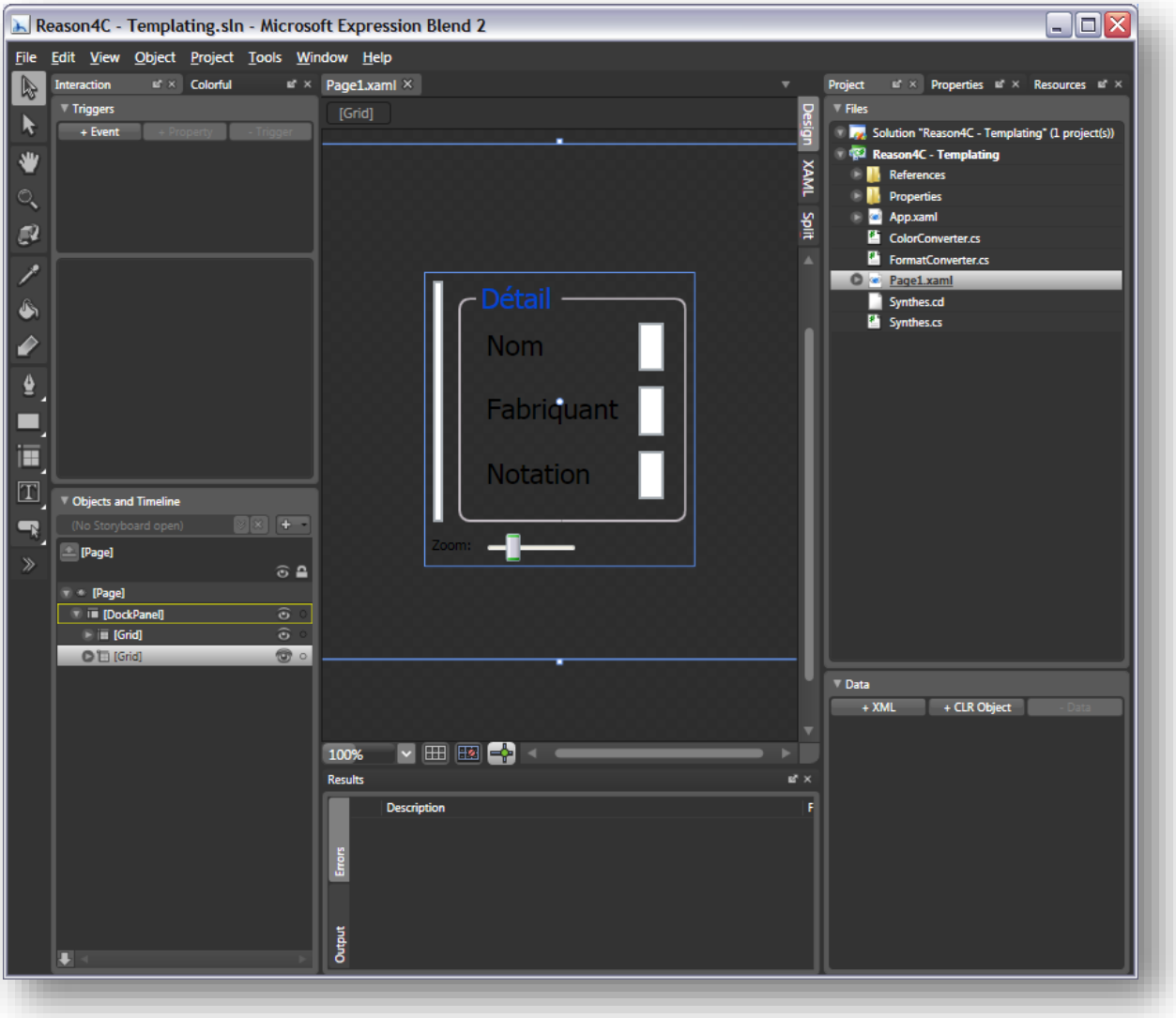


Figure 9 - projet Wpf10-4c

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 :

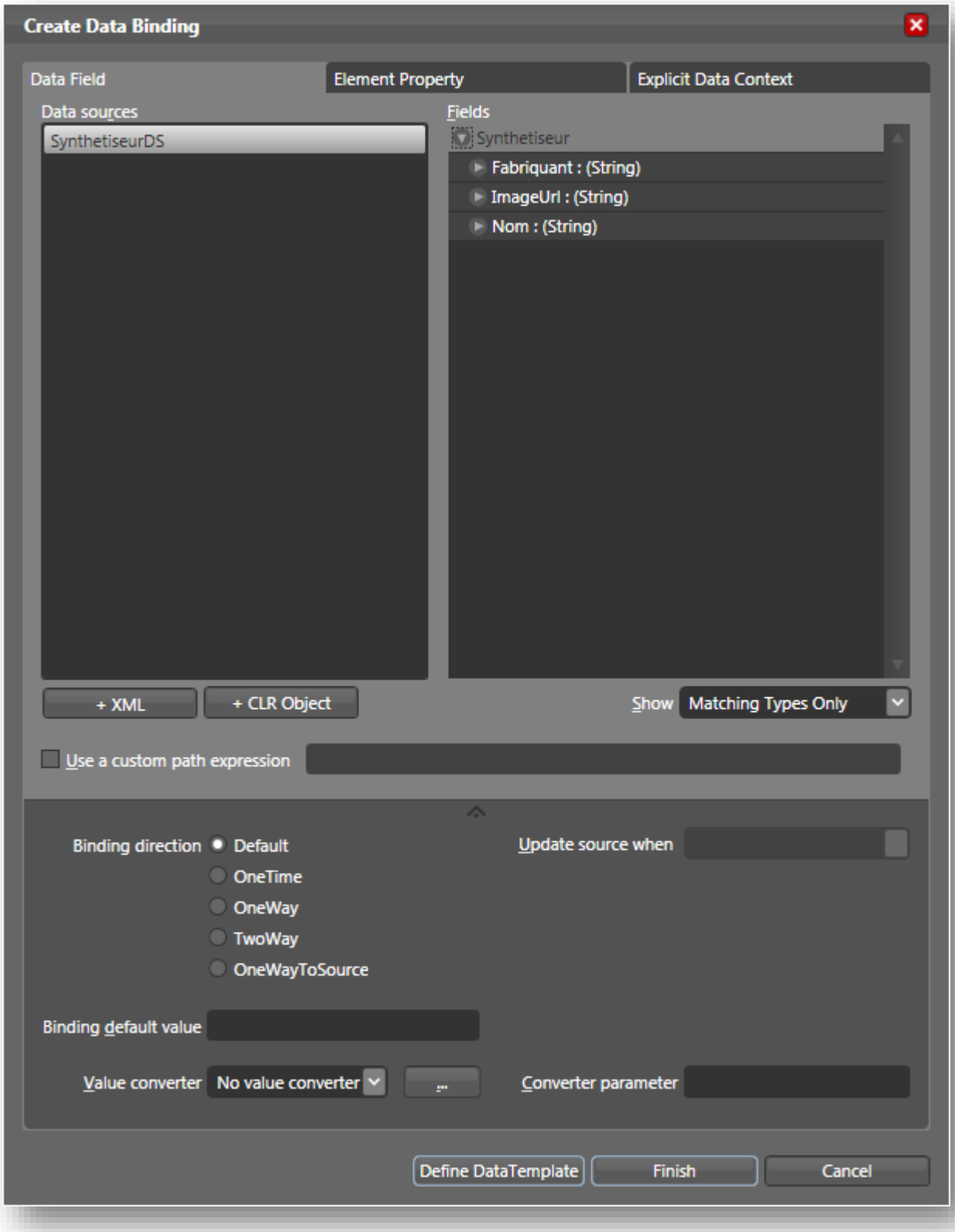


Figure 10 - La création d'un Data Binding sous Blend

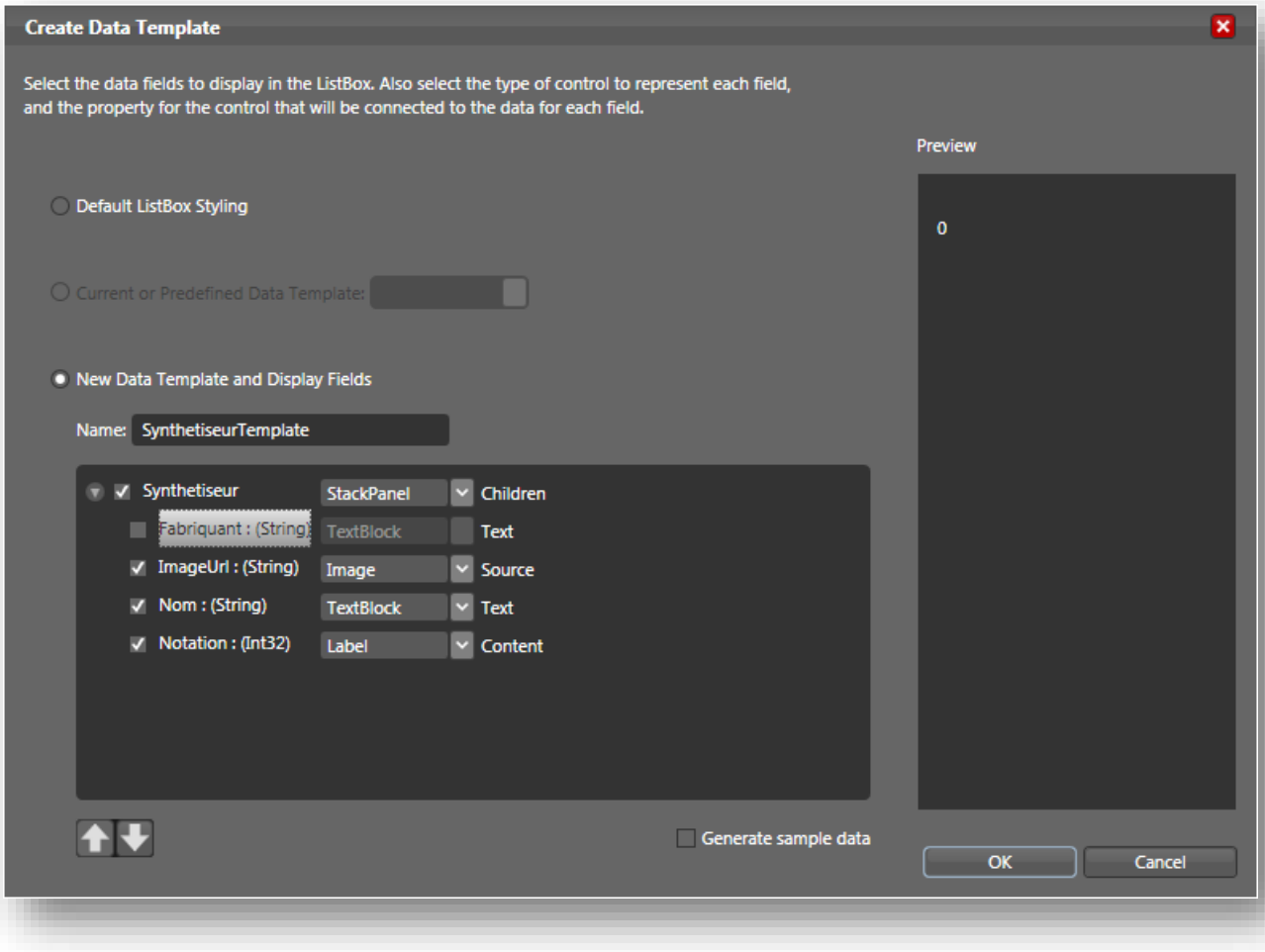


Figure 11 - la création d'un Data Template sous Blend

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).

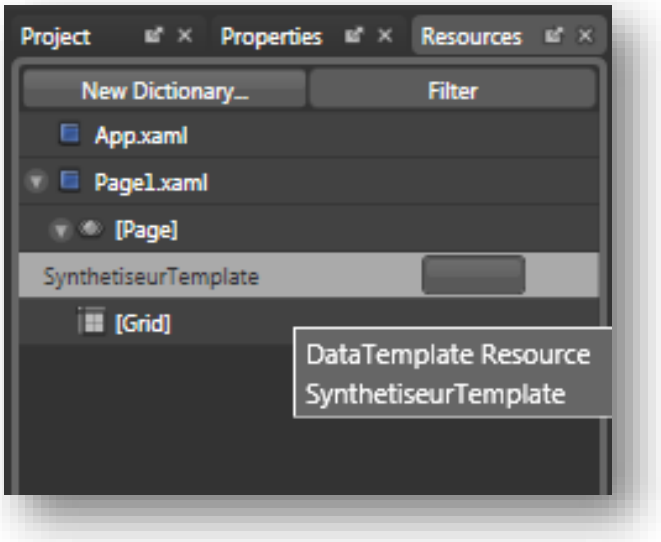


Figure 12 - L'onglet Ressources sous Blend

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 :

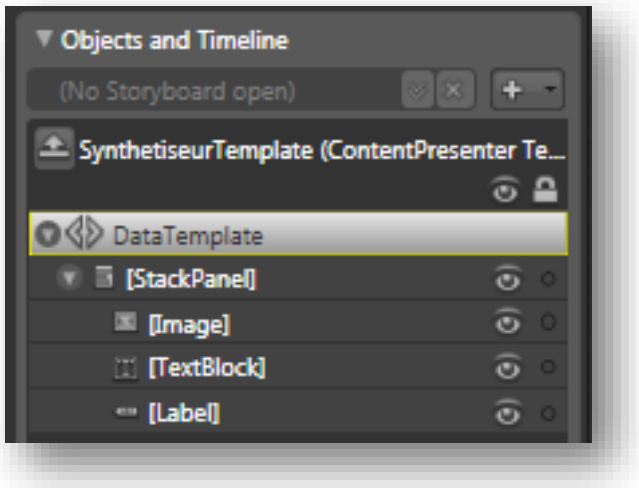


Figure 13 - Le panneau Objets et Animations de Blend

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 maîtriser, il ne faut pas se voiler la face : **pas de développement sérieux XAML sans 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 (même si VS intègre un éditeur visuel tout à fait convenable, c'est l'outil Blend qui a été entièrement conçu pour le travail du visuel, c'est ce qui lui confère sa supériorité sur VS).

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 :

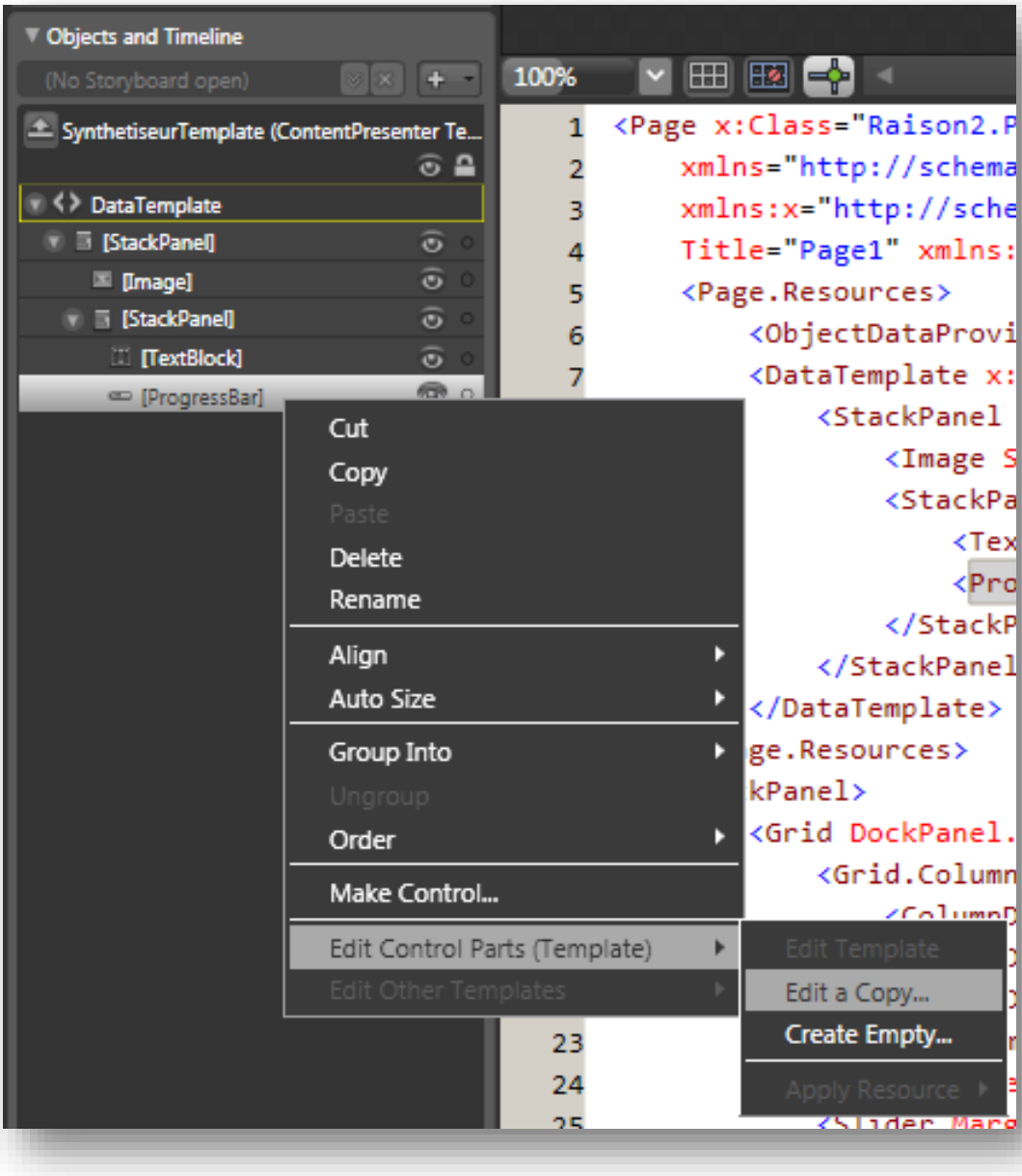


Figure 14 -Création d'un template (mode copie)

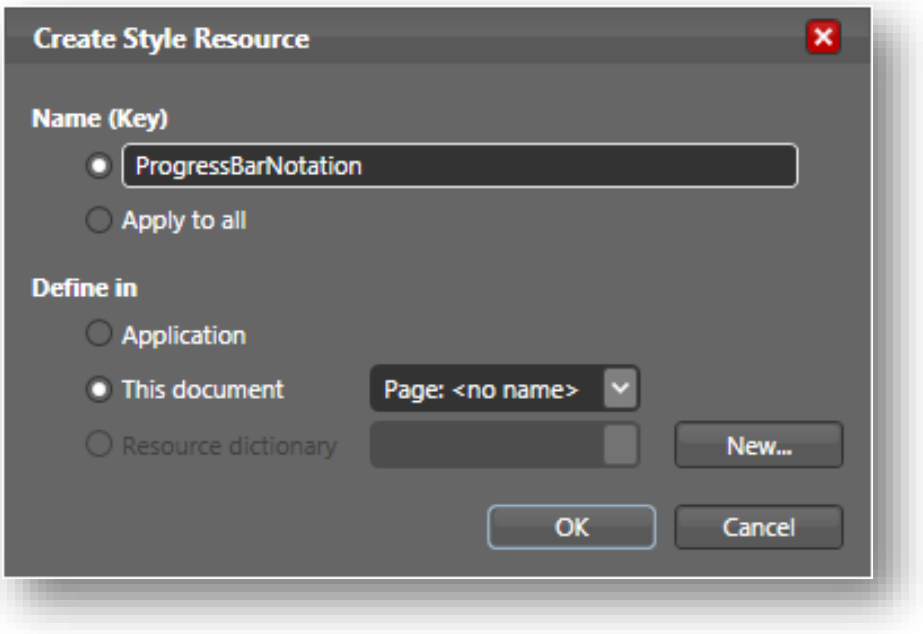
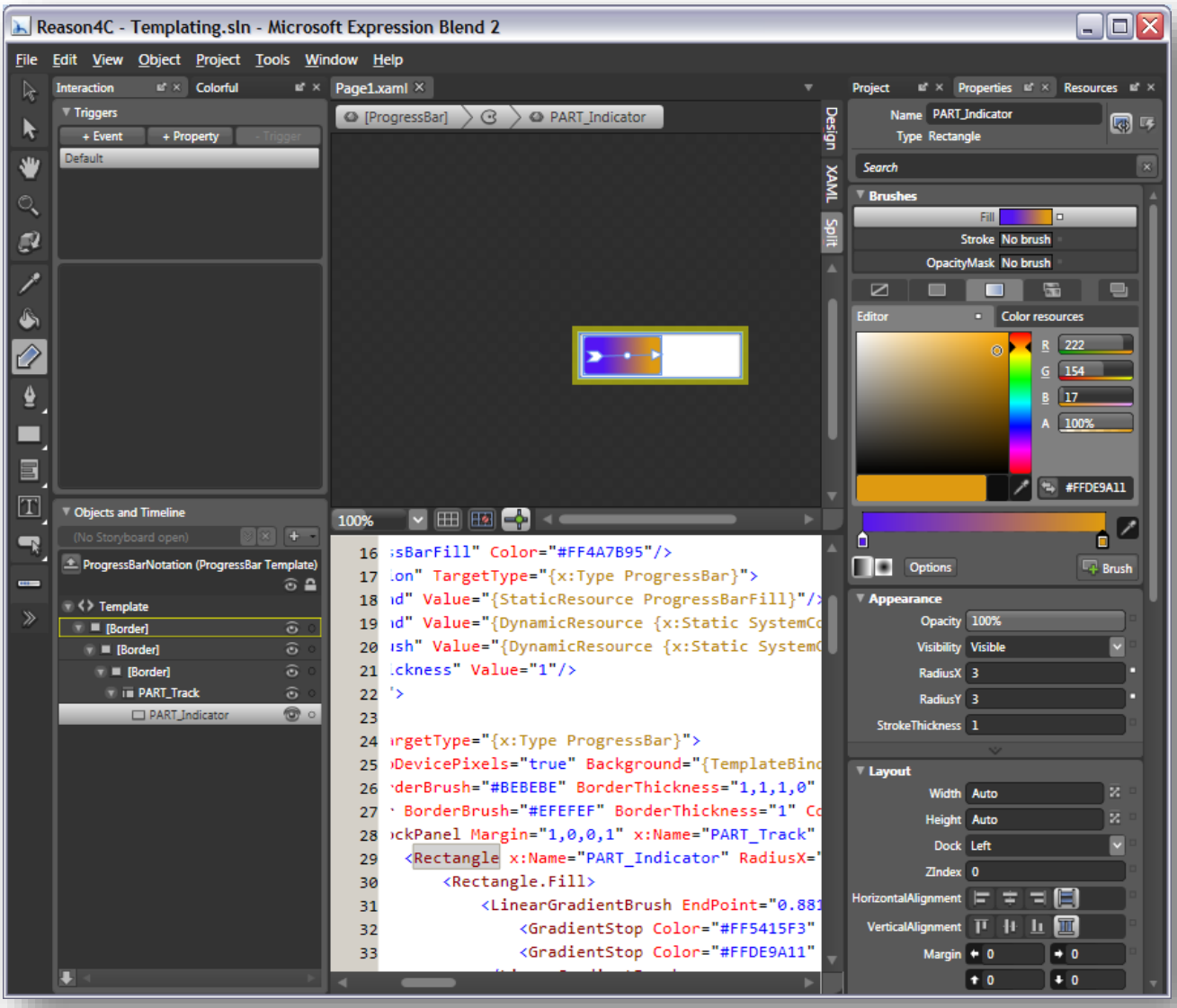


Figure 15 - Création d'une ressource de style

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, Vista, 7 ou Windows 8.x et même Windows Phone ou WinRT). 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 définie 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… La raison est toute bête sur ma machine en dual boot XP/Vista de l'époque j'ai commencé l'article un jour où j'étais sous XP et non Vista, et je l'ai terminé en restant sous XP ! Aujourd'hui on verrait la même chose sous Windows 7 ou même Windows 8.x preuves de la compatibilité de XAML et WPF avec tous ces environnements.

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. Le design pattern *Observer* devient natif 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.).

Ce qui se programme à coup « nine-patches » à fournir dans 50 résolutions et tailles différentes sous Android, ou de façon proche sous iOS, se limite à une gestion des états visuels parfaitement cohérente et intégrée sous XAML, le tout en vectoriel sans avoir à redessiner 10 fois le même « png ». Depuis 7 ans XAML a 20 ans d'avance…

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 XAML 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>
```

XAML 12 - Templating des items d'une liste

Le code ci-dessus reprend le `DataTemplate` créé sous 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 XAML 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>
```

XAML 13 - DataTrigger

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 :

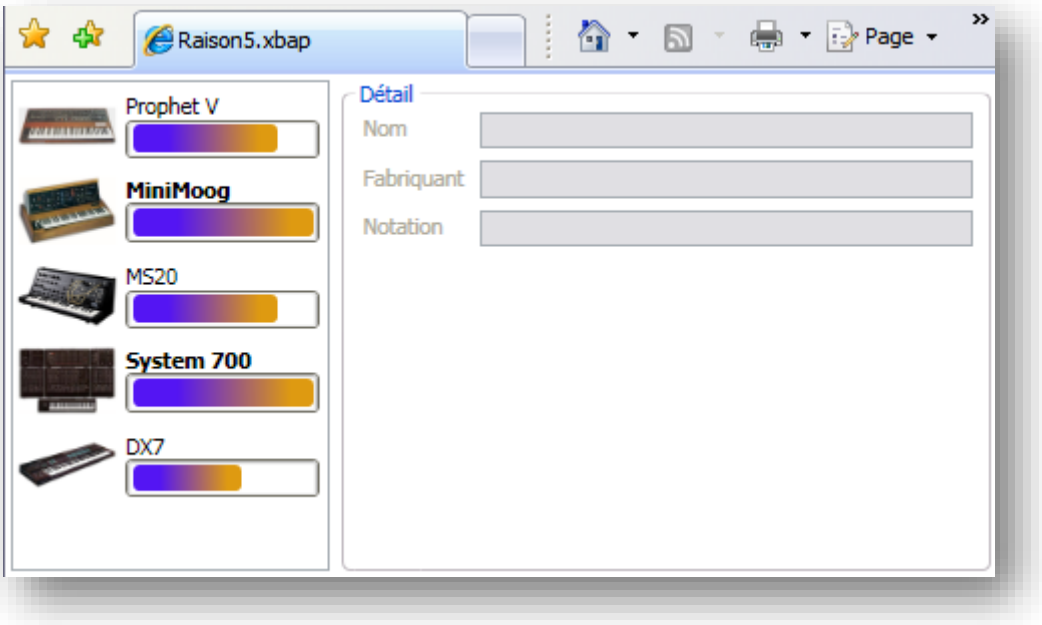


Figure 18 - Le cadre "détail" n'est pas actif

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 XAML 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 (surtout les combinaisons qui se multiplient et les façons d'y entrer ou d'en sortir).

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 là que le `VisualStateManager` de Silverlight, et désormais de tous les profils XAML, 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.

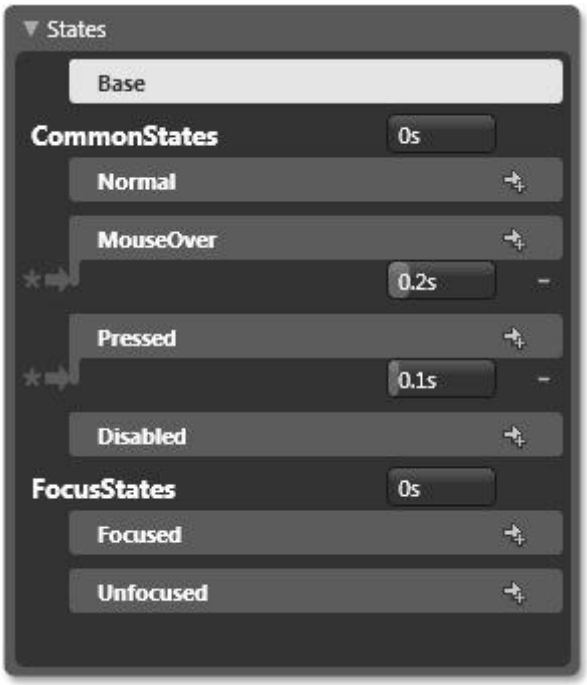


Figure 19 - Le VisualStateManager sous Blend

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 XAML. 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 XAML 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 XAML 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 XAML se placent ainsi un cran plus haut dans la hiérarchie que les templates.

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

Pour cela nous allons créer un nouveau dictionnaire de ressources, c'est la 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.

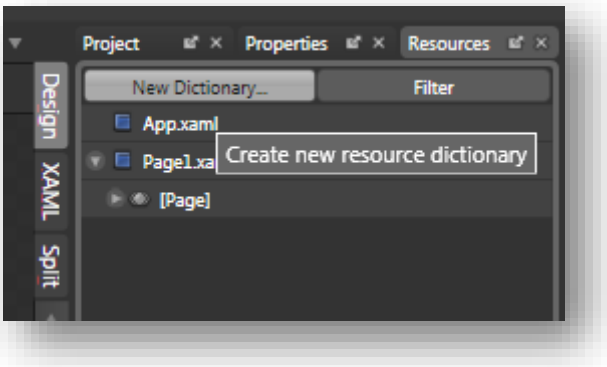


Figure 20 - La création d'un nouveau dictionnaire de ressources

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>
```

XAML 14 - Dictionnaire de ressources et styles

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>
```

XAML 15 - Définition d'une brosse

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 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>
```

XAML 16 - Définition d'un 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 :

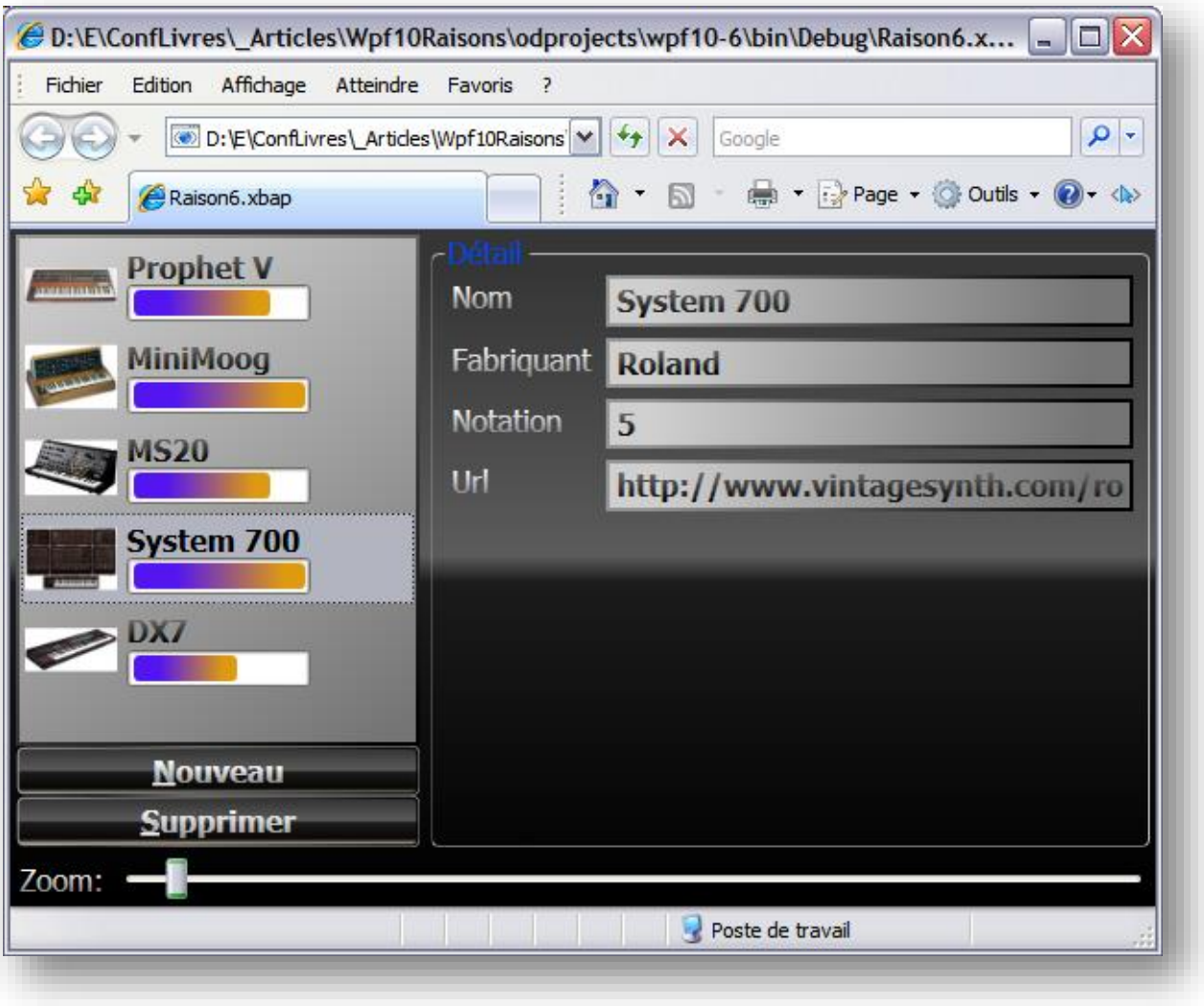


Figure 21 - Projet Wpf10-6

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>
```

XAML 17 - Validation des saisies par défaut

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 XAML 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 XAML 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. Je ne parle pas du cirque indécent des images à créer dans toutes les résolutions pour la programmation Web ou même celle des environnements Android ou iOS, c'est à pleurer pour qui manipule XAML…

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 XAML…

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>
```

XAML 18 - Personnalisation de l'affichage des erreurs de validation

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

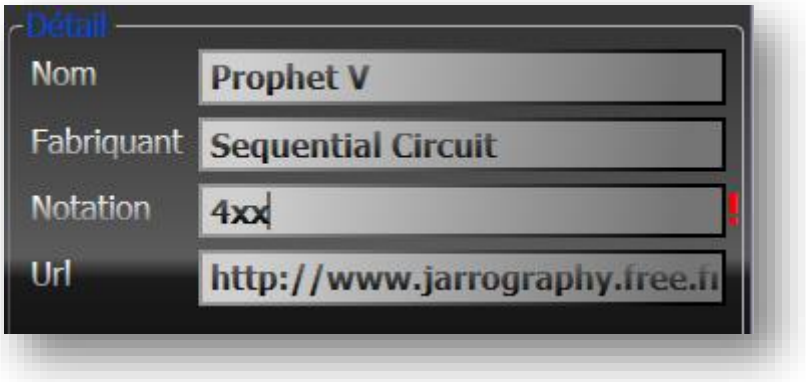


Figure 23 - L'effet du template de validation personnalisé

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 Blend. Le code montré ici ne sert qu'à expliquer les principes et ne constitue pas une méthode de travail conseillée.

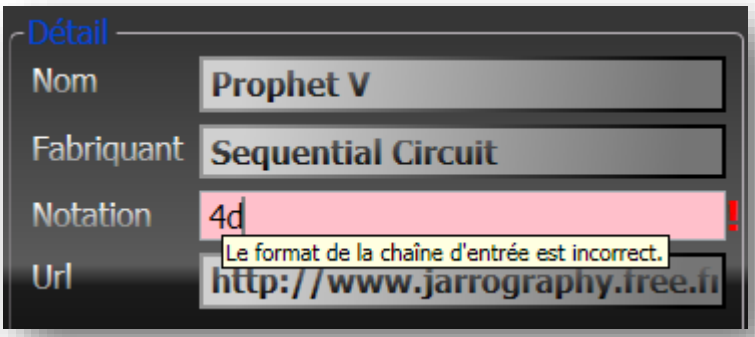


Figure 25 - projet Wpf10-7

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 XAML 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);
            }
        }
    }
}
```

Code 3 - ValidationRule

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" />
```

XAML 19 - Personnalisation des règles de validation des données

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 (des tas d'améliorations ont été ajoutées en 5 ans !), 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 XAML 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 XAML ou 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, sans vraiment de Design particulier ni sophistiqué.

Comme vous l'avez vu, XAML 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 ou Windows 7 utilisent XAML 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 (je parle ici des tables Surface, avant que ce nom ne soit donné à des tablettes).

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.

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 plastique. Le voici en cours de réalisation sous Zam3D :

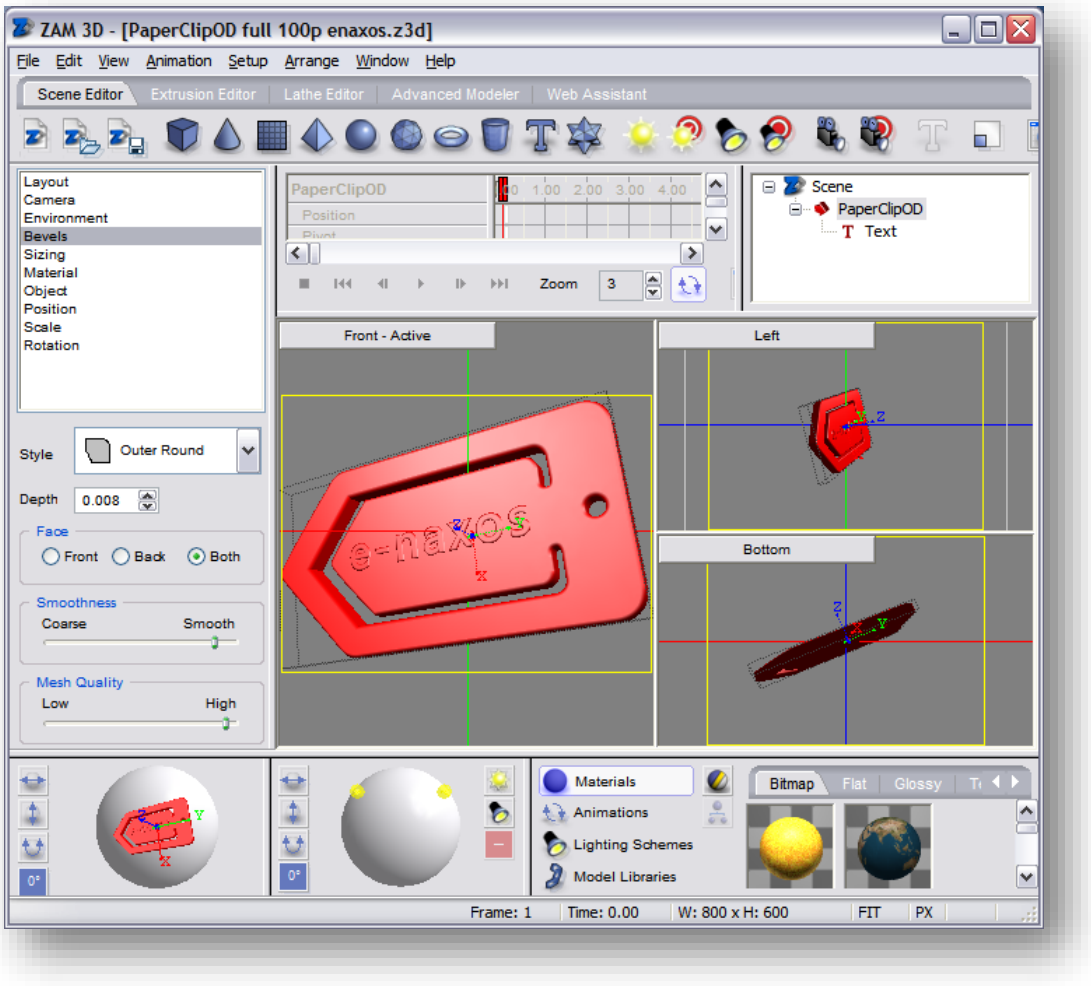


Figure 27 - ZAM3D en action

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 de 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 !

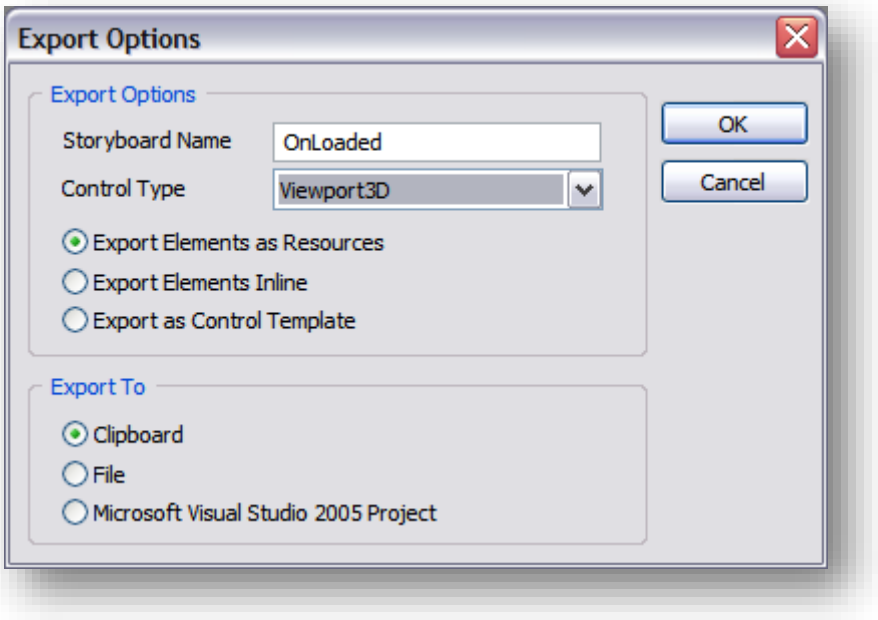


Figure 28 - Exportation XAML d'un projet 3D sous ZAM3D

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

Passons maintenant sous 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 :

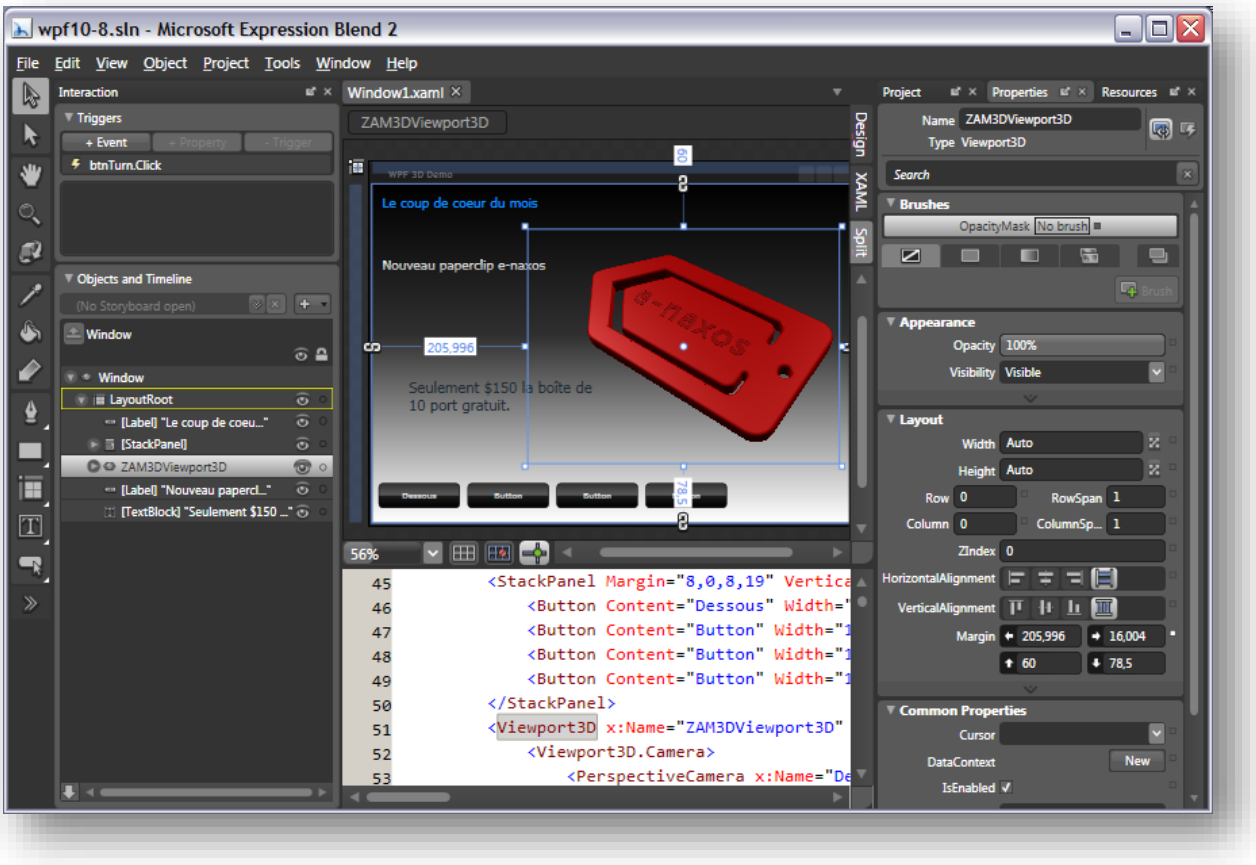


Figure 29 - L'intégration de la source 3D dans une page WPF sous Blend (Projet Wpf10-8)

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.

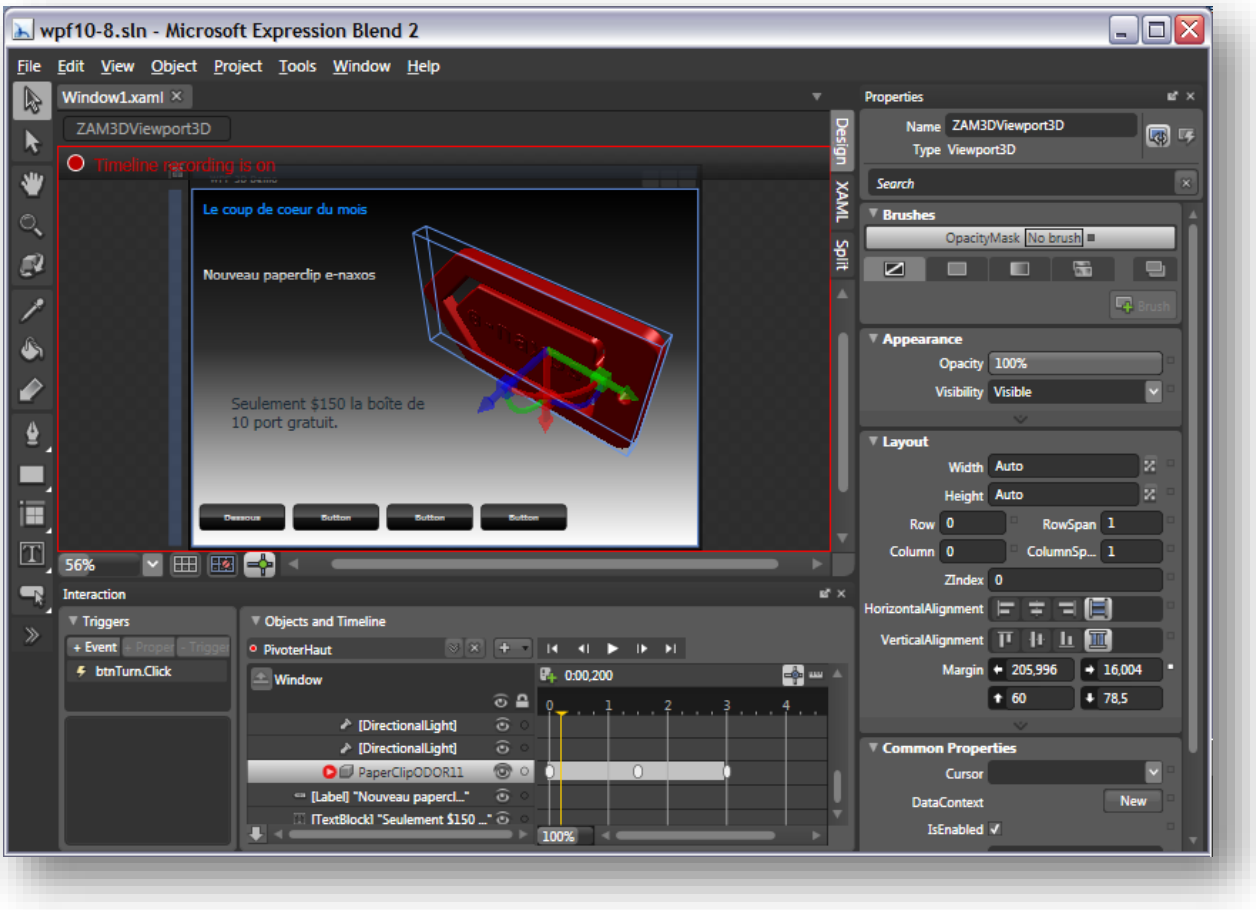


Figure 30 - Animation de l'objet 3D (timeline et story-board sous Blend)

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 :

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 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 !

Il faut noter qu'une 3D aussi simple à manipuler en pur XAML est uniquement disponible sous WPF. Silverlight a reçu le support de la 3D mais par un bricolage passant des librairies de la Xbox non supportées par Blend, ce qui en fit perdre tout

intérêt. Et à ce jour WinRT propose aussi des méthodes DirectX héritées de COM et autres bricolages pour faire des jeux ou de la 3D. C'est certainement plus rapide que d'animer du vectoriel sous WPF, mais cela réclame une spécialisation technique là où WPF fait tout de façon cohérente avec XAML.

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 : 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 avec Windows Phone ou WinRT.

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… Tous les frameworks MVVM créés ensuite comme Prism, Jounce, Mvvm Light, MvvmCross et bien d'autres seront tous basés sur cet esprit unique de séparation de l'UI et du code.

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 et XAML 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 XAML 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 XAML, vous avoir rappelé que même aujourd'hui WPF est un excellent choix 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 !

9 raisons de plus d'utiliser XAML !

Dans mon dernier article, *10 bonnes raisons de choisir XAML*, je vous proposais sous la forme d'une introduction à WPF et XAML ce qui me semblait de bonnes raisons de choisir cet environnement pour vos nouvelles applications. Dario Airoldi de Microsoft Italie nous propose aujourd'hui ses 9 raisons de préférer WPF et ce ne sont pas forcément les mêmes que les miennes, ce qui allonge significativement la liste et vaut un petit détour. Encore une fois il faut comprendre XAML quand je parle de WPF. Ce profil particulier de XAML est historiquement le premier et c'est surtout, même aujourd'hui, celui qui est le plus complet. Toutefois tout ce qui est dit ici peut se transposer à Silverlight, Windows Phone ou WinRT.

Je ne vais ni traduire cet article (suivez le lien juste au-dessus) ni réviser le mien en ajoutant ces nouvelles raisons, mais voici un résumé des raisons de Dario, elles sont intéressantes :

Raison 1

L'analyste fonctionnel et le graphiste peuvent définir l'interface d'un logiciel par le biais d'un langage commun, **XAML**, et non plus par des bitmaps et des documents écrits que le développeur devait traduire en code. Avec XAML, développeur et designer peuvent travailler sur une base commune et des documents directement utilisables par les uns et les autres sans "traduction", ce qui diminue grandement les risques de confusion.

> *XAML n'impose pas systématiquement le travail d'un graphiste, mais pour obtenir une interface de grande qualité graphique une telle présence s'avère indispensable. Il ne s'agit donc pas d'une obligation de XAML mais bien d'une exigence de qualité de l'expérience utilisateur (UX). Vous avez le droit de faire des choses très laides de type Win32 en XAML si cela vous chante ou vous passer d'un infographiste si vous êtes aussi doué au pinceau qu'en C# !*

Raison 2

La quantité de code C# ou VB est grandement réduite grâce à XAML. Toute l'interface est gérée soit par du **XAML** soit par du **Data Binding**.

Raison 3

La séparation entre l'interface utilisateur et la logique métier est nette et franche (voir les stratégies issues du pattern MVVM).

Raison 4

Le système de routage des commandes de WPF permet de découpler l'implémentation d'une action de l'objet émettant la commande.
Cet aspect essentiel dans WPF à sa sortie l'est moins aujourd'hui où les commandes sont gérées par les ViewModels dans le pattern MVVM. Mais cela ne serait pas possible sans le support de ICommand par les objets XAML.

Raison 5

Le chargement dynamique de code XAML rend possible l'adaptation à la volée de l'interface utilisateur, par exemple selon le rôle de l'utilisateur, son groupe de travail, son profil...

Raison 6

Les validations côté client et côté serveur sont largement simplifiées par l'architecture de validation du Data Binding et par le système de routage des événements.

Raison 7

Les applications WPF sont plus faciles à maintenir et plus flexibles.
Cela découle en partie des raisons précédentes et de la nature même de XAML et des outils qui gravitent autour comme Blend par exemple.

Raison 8

Le **templating** des contrôles et les **styles** permettent de créer des interfaces sophistiquées tout en restant humainement maintenables.
XAML a été créé en intégrant le graphisme, le multimédia, les animations, la 3D, etc...

Avec XAML on personnalise l'interface à l'extrême si besoin tout en utilisant des composants standards et sans nécessité d'acquérir des bibliothèques tierces, impossibles à maintenir, chères et incompatibles entre elles.

Raison 9

WPF fonctionne sur la base d'un système de coordonnées déconnecté de la résolution des écrans ce qui rend l'adaptation au matériel bien plus simple.
Il fut un temps où tout le monde fonctionnait avec "la" norme de l'instant, comme VGA par exemple. Aujourd'hui l'utilisateur a un choix énorme d'écrans aux proportions, tailles et résolutions différentes, sans parler des unités mobiles et autres smartphones ! XAML apporte une solution simple à ce problème. Quand on voit comment la gestion des résolutions est un enfer sous Android avec ses PGN, ses nine-patches, etc, on ne peut que constater que le vectoriel est une solution mille fois supérieure à tout ce qui se pratique ailleurs. C'est un avantage qu'on retrouve partout où XAML s'utilise : WPF, WinRT, Windows Phone…

Conclusion

XAML facilite le développement d'applications plus sophistiquées et plus ergonomiques en écrivant moins de code.
Tout cela est vrai. Mais n'oublions pas que XAML impose aussi une solide formation car la façon de travailler sous WPF ou WinRT est très différente de ce qu'on connaissait. Nier cet aspect des choses ne serait pas honnête et c'est ce qui explique que beaucoup de développeurs n'ont pas encore sauté le pas.

Les propriétés de dépendance et les propriétés attachées

En voilà un beau sujet ! Vous allez me dire qui irait investir deux jours à taper 30 pages sur ce sujet, il faut être totalement givré ! Et bien vous en avez un devant vous (par livre interposé) … donc pas de remarques désobligeantes sur ma santé mentale, hein !

Qui plus est, quelle drôle d'idée d'aller placer un tel sujet ingrat et qui porte principalement sur du code C# au début d'un livre consacré à un autre langage, XAML !

N'étant pas totalement fou, enfin je crois, il existe une raison qui m'apparait rationnelle pour justifier un tel choix : Les propriétés de dépendance sont essentielles sous XAML, elles ont été créée pour ce langage pour couvrir des cas d'utilisation que les propriétés habituelles de C# ne couvraient pas. Faire du XAML, parler de XAML sans comprendre ni connaître les propriétés de dépendance serait un peu comme apprendre à piloter un avion en ne voulant pas apprendre à décoller ou atterrir… ça finit mal.

Certes les propriétés de dépendance et les propriétés attachées de XAML ne semblent pas être un sujet aussi excitant que quelques astuces LINQ ou la meilleure façon d'intégrer de la 3D dans une application pour smartphone... Je vous le concède. Mais en revanche c'est un sujet capital car derrière ces propriétés bien particulières se cache l'un des piliers de la puissance de XAML, un mécanisme qui autorise la gestion des styles, des animations, du Data Binding et de bien d'autres choses sans lesquelles XAML ne serait pas ce qu'il est.

Savoir ce qu'est une propriété de dépendance ou une propriété jointe, savoir en déclarer et savoir les utiliser représente **une base impossible à zapper**.

Alors, pour tout savoir sur le sujet lisez ce qui suit… Vous pouvez télécharger l'article original non mis à jour notamment pour récupérer le code source qui lui n'a pas changé : "Propriétés de dépendance et propriétés jointes (WPF/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, j'utiliserai ici l'une ou l'autre de ses traductions. 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.

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 fameuses « dépendances » ?

Sous XAML les objets 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 XAML car le mécanisme plus simple des propriétés CLR n'est pas capable de le faire.

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

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 XAML et par celle du Framework. Ainsi les propriétés de dépendance sont basées à la fois sur une couche de gestion implémentée dans XAML et le Framework .NET 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 XAML unifie la façon dont elles sont déclarées et utilisées.

Et les propriétés attachées ?

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 XAML où le visuel est totalement libre utiliser une telle logique est tout bonnement irréalisable. Que dire de `Top`, `Bottom`, `Right` et `Left` qui, sous XAML, 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 XAML 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. Les limites de l'héritage cher à la programmation Objet sont largement atteintes.

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 attachées 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 XAML.

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 attachée

Prenons une simple `Window` sous WPF 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);
}
```

Code 4 - Propriété attachée

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 `TextBlock` 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 attachées) 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 car lorsqu'on utilise de telles propriétés directement en XAML il n'est pas nécessaire d'ajouter le suffixe.

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) !

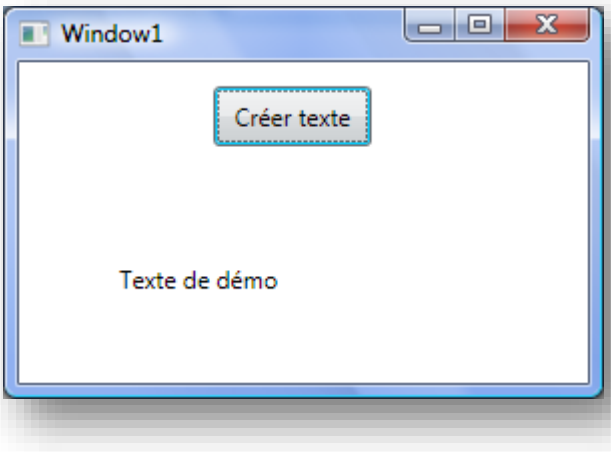


Figure 32 - Création d'un TextBlock

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 XAML est une technologie capable d'instancier des objets sans l'appui d'un autre langage. 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 Blend et Visual Studio 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>
```

XAML 20- Propriété jointes

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 ce n'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 attachées : leur nom n'est jamais qu'un bout de texte ajouté à une liste gérée par XAML, 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 XAML 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 attachée ?

L'exemple que nous venons de développer montre des propriétés jointes ou attachées (*attached properties*). En effet, `Top` et `Left` n'appartiennent pas au composant `TextBlock` mais à la classe `Canvas` qui, via les mécanismes de XAML, 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 compliqué l'existence ?
… Tout simplement pour bénéficier des services que XAML offre aux propriétés de dépendance !

Le service d'animation

Quels sont ces services que XAML 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 XAML 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 d'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 classes 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 XAML ne l'interdit pas (avec de l'imagination on peut peut-être s'en servir !).

```
<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>
```

XAML 21 - Animation de propriété

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 XAML 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 XAML ne se travaille pas en XAML mais à l'aide d'outils visuels, qu'il s'agisse de l'éditeur de Visual Studio ou mieux de Blend qui permet bien plus de choses comme justement la création d'animations.*

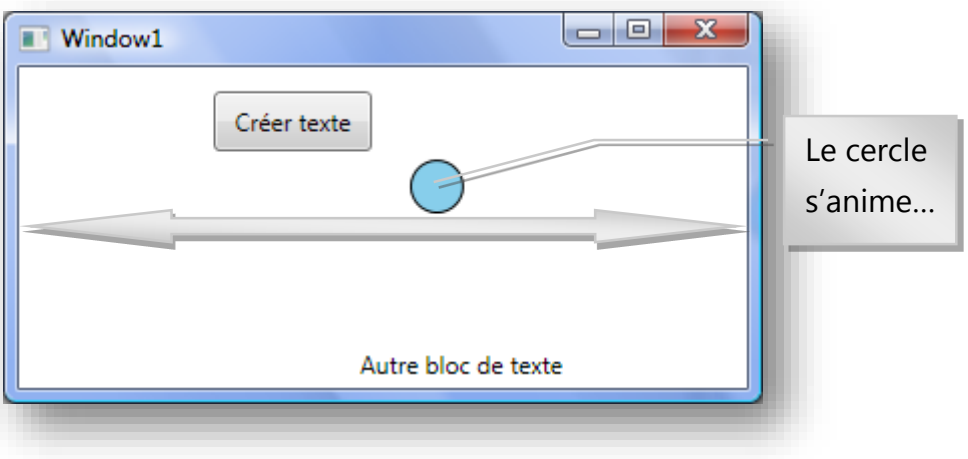


Figure 33 - Animation d'une propriété attachée

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 XAML 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 XAML offre aux propriétés de dépendance :

Valeur par défaut

Fixer une valeur par défaut à une propriété est souvent indispensable. XAML 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). XAML 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 aspects 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 d'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 XAML 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 XAML* »). Ce service n'est offert qu'aux propriétés de dépendance. S'il fallait une raison pour justifier cet article ce serait certainement celle-là car écrire des classes pour XAML 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>
```

XAML 22 - Databinding sur un champ nommé d'un objet

L'héritage des valeurs

Un peu comme les feuilles de style CSS, XAML 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, XAML propose un mécanisme de même type (mais plus sophistiqué) grâce aux styles et aux templates.

Validation des valeurs

XAML 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. XAML instaure des règles de précédences pour clarifier la logique appliquée.

XAML 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 les simplifier est donc bienvenue. Les propriétés de dépendance de XAML 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 XAML 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, voire même moche, 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

Je ne passerai pas trop de temps sur cet aspect des choses mais pour le plaisir de faire de la propagande pour 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` :

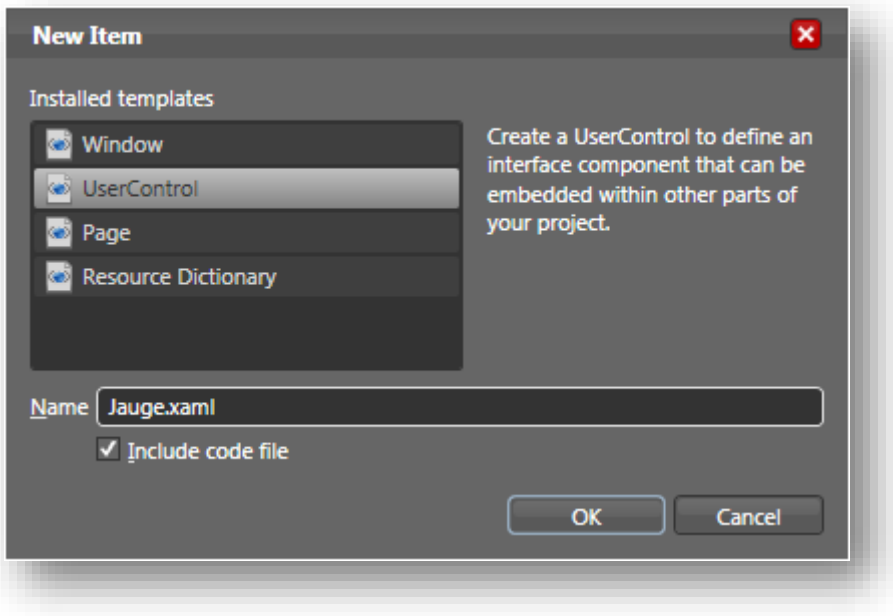


Figure 34 - Création d'un UserControl Jauge

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

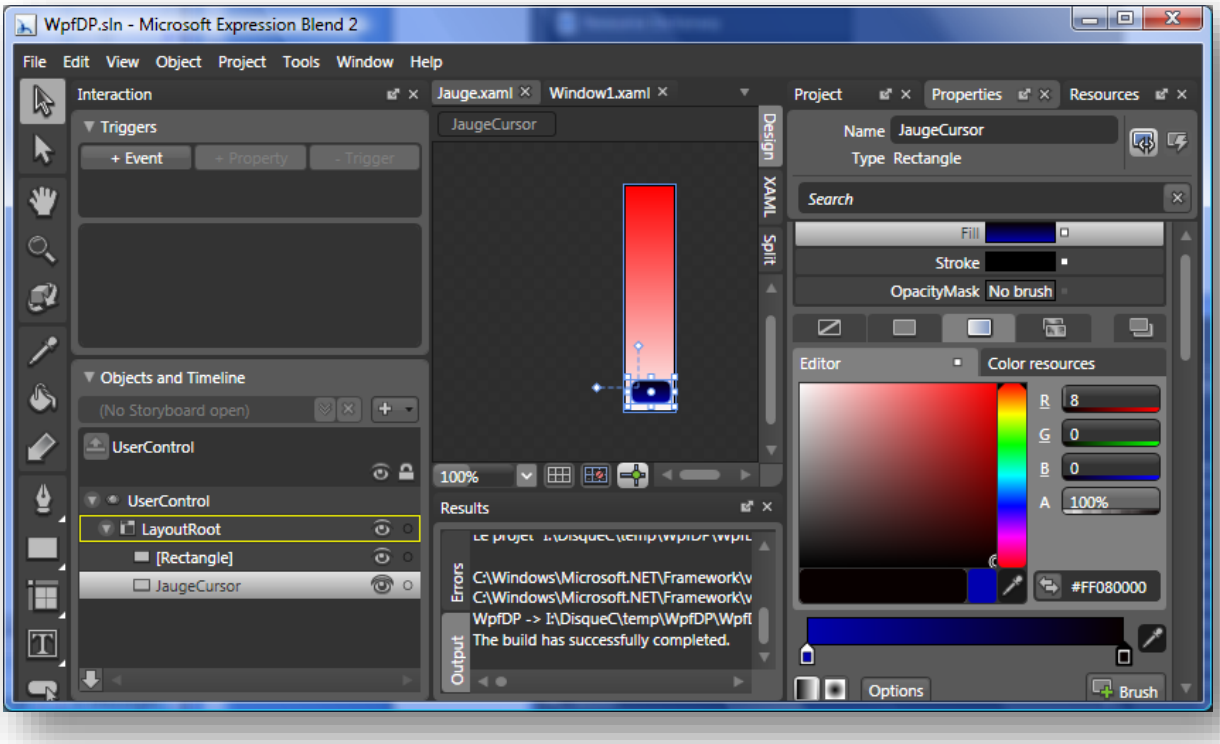


Figure 35 - La conception visuelle du contrôle

Sous 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="LayoutRoot" 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>
```

XAML 23 - Définition d'un 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 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 tous 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));
```

Code 5 - Définition d'une propriété de dépendance

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); }
}
```

Code 6 - Propriété CLR sur propriété de dépendance

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 XAML 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));
}
```

Code 7 - Gestion du changement de valeur d'une propriété de dépendance

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;
  }
 }
```

Code 8 - Callback de validation sur propriété de dépendance

Tout comme le callback précédent, la validation de valeur est une méthode statique. Toutefois si pour le callback de changement de valeur XAML 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 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 directement 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 ! On notera toutefois que ces callbacks particuliers sont présent sous WPF mais ne le sont pas sous Silverlight (version Web ou Windows Phone) car il fallait couper quelque part pour « rétrécir » ces profils XAML particuliers. Simuler la *coersion* de valeur sur les environnements ne le prenant pas en compte s'avère très complexe et les solutions proposées ici ou là ne sont pas satisfaisantes. Dès lors c'est au code C# qui manipule les objets graphiques de gérer la validation des valeurs passées. Cela est moins élégant, mais il n'y a pas d'autres choix.

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 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).

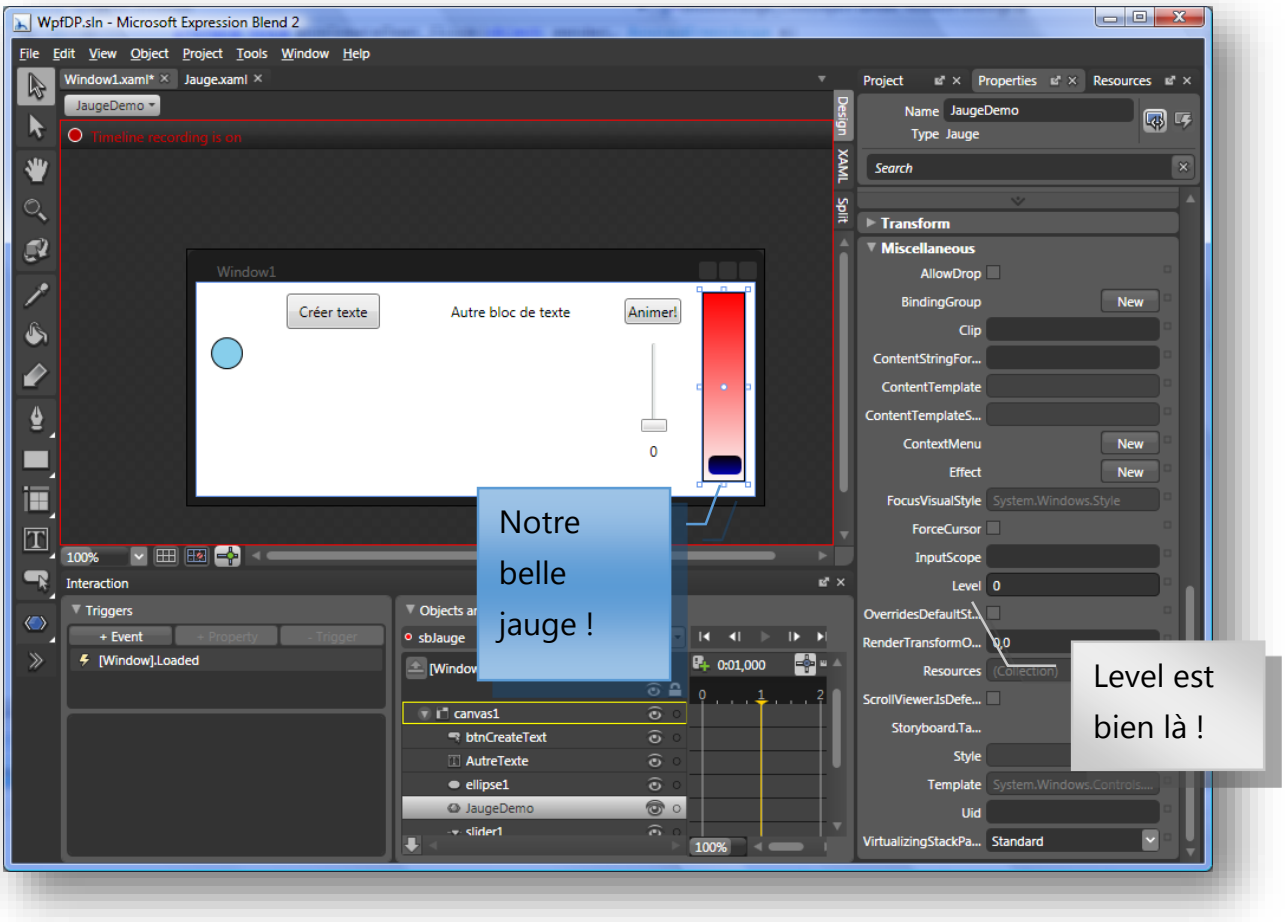


Figure 36 - La propriété de notre contrôle sous Blend

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) :

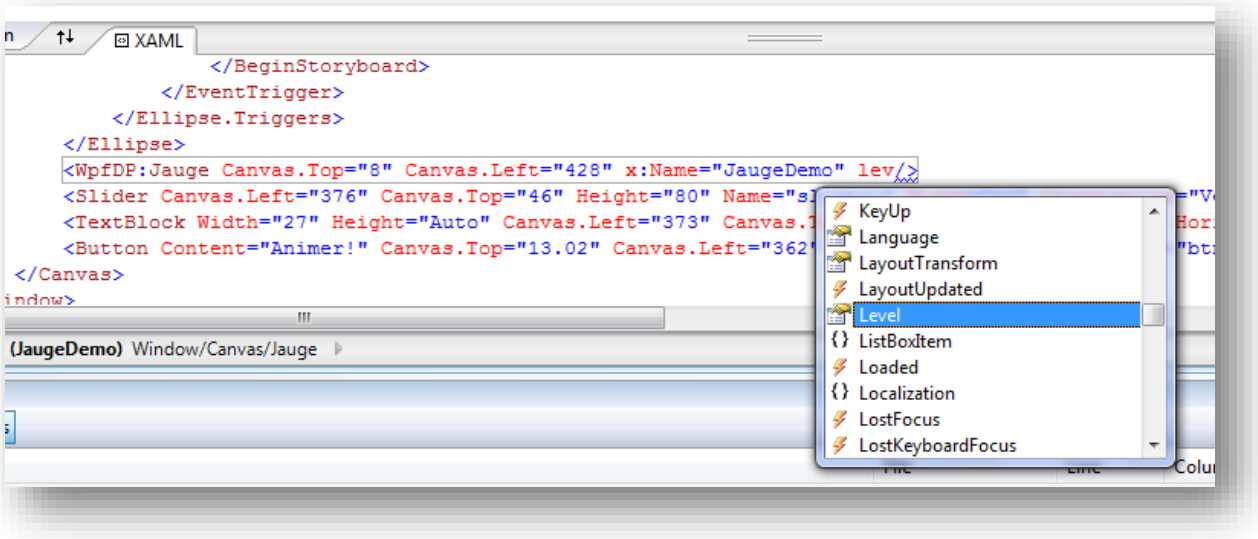


Figure 37 - La propriété manipulée en XAML sous VS

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 vite 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 ? C'est qu'on appelle de l'Element Binding permettant de connecter par binding deux éléments XAML directement sans passer par du code externe.

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}"
/>
```

XAML 24 - Databinding avec StringFormat

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 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 !

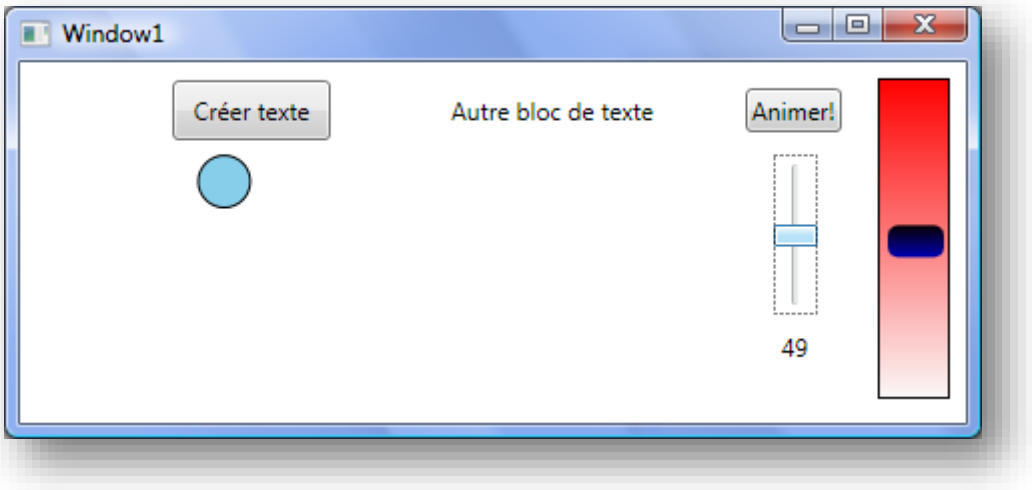


Figure 38 - L'animation de la jauge

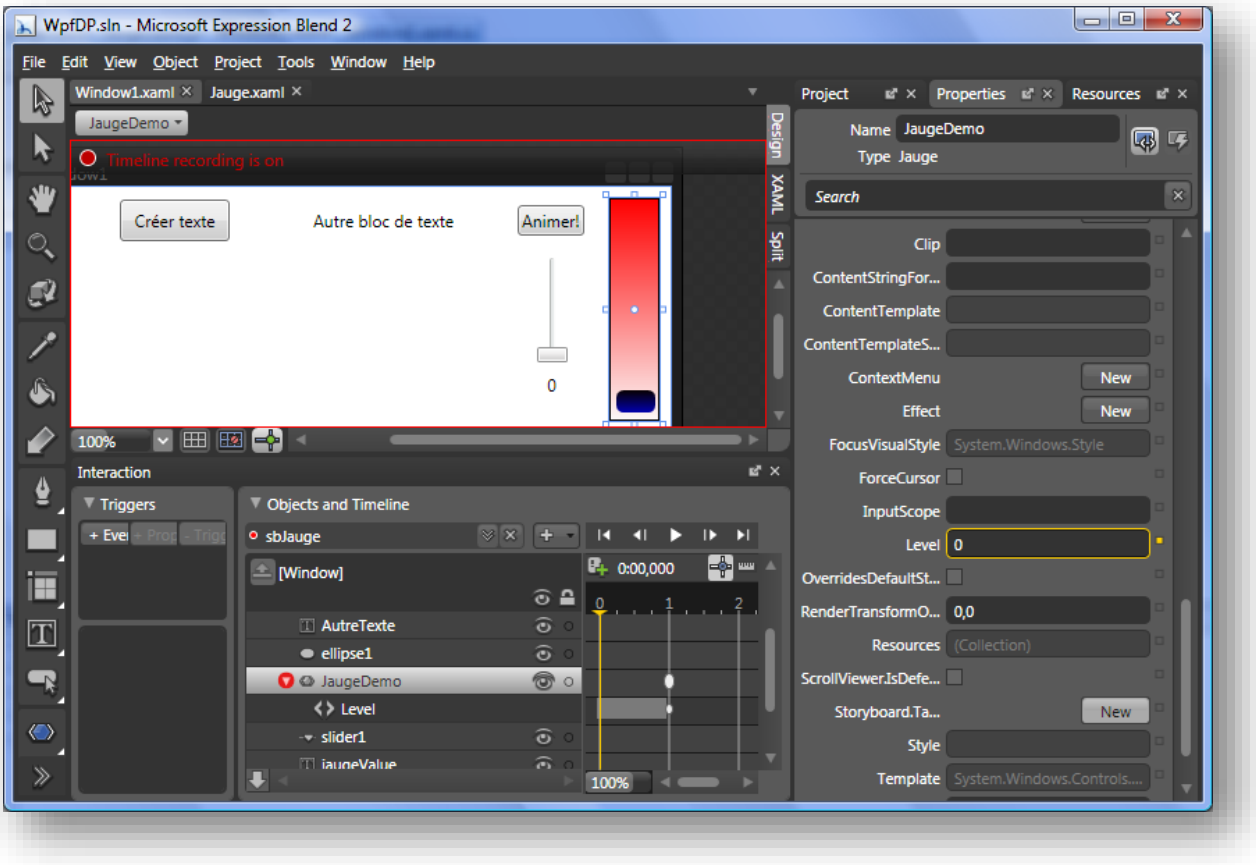


Figure 39 - Création de l'animation sous Blend

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;
}
```

Code 9 - Lancement d'une animation

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));
```

Code 10 - Déclaration d'une propriété jointe

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);
}
```

Code 11 - Déclarations internes du framework pour les propriétés jointes

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));
```

Code 12 - Héritage de valeur d'une propriété de dépendance

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 indiqué 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

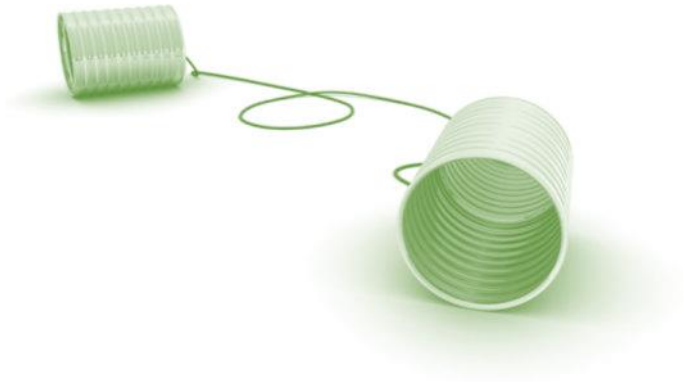
Comme on le voit ici, les propriétés de dépendance ont une mise en œuvre interne assez complexe. Si les propriétés de dépendance n'avaient pas été ajoutées à WPF pour encapsuler tous les comportements que nous avons étudié ici, la programmation WPF avec des propriétés CLR serait un enfer…

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 XAML doivent l'être en utilisant des propriétés de dépendances et non de simples propriétés CLR…

Le BINDING Xaml – Maîtriser sa syntaxe et éviter ses pièges



Références

Vous trouverez ici l'ensemble des références indiquées dans le cours de l'article afin de pouvoir visiter les liens sans nécessairement fouiller dans le texte.

> *Nota : tous les liens ont été vérifiés à la date d'écriture de création du présent livre PDF. Internet est un monde en perpétuel mouvement, si une référence venait à ne plus être valide, n'hésitez pas à me le signaler pour que je mette le lien à jour.*

M-V-VM et Silverlight – De la théorie à la pratique
http://www.e-naxos.com/Blog/post/2010/01/09/Article-M-V-VM-avec-Silverlight.aspx
Le retour du spaghetti vengeur
http://www.e-naxos.com/Blog/post/2010/03/13/Le-retour-du-spaghetti-vengeur.aspx
L'Element Binding
http://www.e-naxos.com/Blog/post/2009/07/23/Silverlight-3-LElement-Binding.aspx
Simple MVVM
http://www.e-naxos.com/Blog/post/2010/02/24/Simple-MVVM.aspx
MVVM, Unity, Prism, Inversion de controle
http://www.e-naxos.com/Blog/post/2009/12/23/MVVM-Unity-Prism-Inversion-of-Controle280a6.aspx
Les propriétés de dépendance et les propriétés jointes
http://www.e-naxos.com/Blog/post.aspx?id=ee3a2372-acd9-4650-a1d9-76ce4f21e483
Model-View-ViewModel
http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx
Reshaper
http://www.jetbrains.com/resharper/
Trace sources in WPF
http://blogs.msdn.com/mikehillberg/archive/2006/09/14/WpfTraceSources.aspx
How Can I debug WPF bindings ?
http://bea.stollnitz.com/blog/?p=52
Spy++
http://msdn.microsoft.com/en-us/library/aa264396(VS.60).aspx
Mole
http://karlshifflett.wordpress.com/mole-for-visual-studio/
Reflector
http://www.red-gate.com/products/reflector/index.htm
WPF Multibinding
http://www.switchonthecode.com/tutorials/wpf-tutorial-using-multibindings
PreviousData Binding
http://jimmangaly.blogspot.com/2008/12/discovering-relativesourcepreviousdata.html
Contrasting Silverlight and WPF
http://msdn.microsoft.com/en-us/library/dd458872.aspx

Code Source

Cet article est accompagné du code source complet de l'application exemple si tel n'est pas le cas vous pouvez télécharger la version complète (PDF + code) sur le site e-naxos (voir la première page de ce document).

Décompressez le fichier Zip dans un répertoire de votre choix. Le code réclame au minimum Visual Studio 2008 et Blend 3. La mise automatique vers des versions plus récentes ne pose pas de problème.

Répertoire	Projet	Page
Sample0	ConverterSample	138
Sample1	WpfMiniBinding	121
Sample2	SimpleBinding	165
Sample3	XmlBinding	183
Sample4	RelativeBinding	188
Sample5	PreviousDataBinding	194
Sample6	TemplateBinding	201
Sample 7	ListBinding	207
Sample8	PriorityBinding	209

Préambule

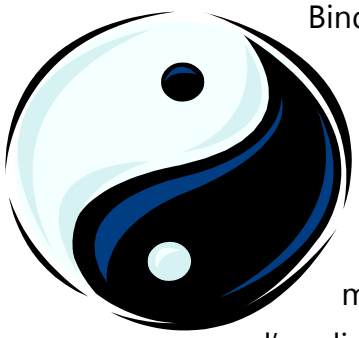
Le Binding (qu'on traduira par « ligature » ou simplement « lien » en français) est un pilier fondateur de **Xaml**, donc de WPF et Silverlight ainsi que Windows Phone ou WinRT. Il s'agit de pouvoir relier deux propriétés de deux objets de telle façon à ce que tout changement dans l'une soit automatiquement reporté dans l'autre (voire dans les deux sens selon le mode choisi).

Apportant une grande richesse à Xaml, le Binding est implémenté dans ce langage sous la forme d'extensions au sein des balises, les commandes étant exprimées en chaines de caractères non contrôlées à la compilation. A la fois cet état de fait et le côté opaque de la syntaxe peuvent être sources de bugs ou de difficultés. Cet article fait le point sur ces dernières, la syntaxe et le débogue du Binding Xaml.

> *Note : Le présent article est fourni avec des projets exemples et les extraits de codes qui sont proposés en sont majoritairement issus. Par souci pratique l'ensemble des exemples a été réalisé sous WPF avec VS 2008 pour que la plus grande partie des lecteurs puisse en profiter. Il existe quelques nuances entre WPF, Silverlight et les autres moutures de XAML, tout ce qui est montré dans cet article ne s'applique pas forcément à ces derniers. Seul WPF est réputé supporter la totalité des fonctions présentées ici. Le lecteur pourra consulter le site MSDN pour connaître les différences existantes à un instant donné. Voir dans les Références de l'article (page 117) pour les liens.*

Le Binding Xaml : Ange ou Démon ?

Dans la pratique le Binding occupe une place essentielle car il évite beaucoup de code behind, simplifie l'écriture des applications orientées données (on parle de Data Binding), clarifie les échanges entre contrôles visuels et code (via un `DataContext` par exemple) ou d'autres contrôles visuels (on parle d'`Element Binding`).



Le Binding est ainsi logiquement au cœur des bonnes pratiques de programmation sous XAML, et se retrouve même propulsé à la place de moteur indispensable à l'application de patterns tel que *Model-View-ViewModel* dont je vous ai longuement entretenu dans un précédent article « M-V-VM et Silverlight – De

la théorie à la pratique » (Le lecteur intéressé lira certainement avec attention le Tome 2 de ALL DOT BLOG consacré aux Méthodes et Frameworks MVVM).

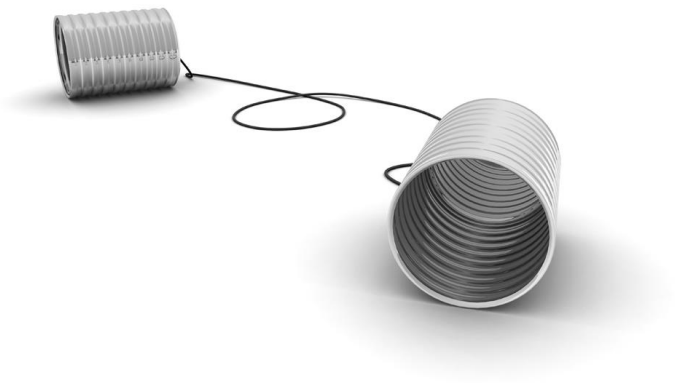
Sur Dot.Blog j'ai lancé il y a quelques temps un pavé dans la mare de cette mécanique superbe mais fragile. Le billet « Le retour du spaghetti vengeur » retrace avec un peu d'humour un audit m'ayant permis de constater à quel point l'implémentation totalement non contrôlée du Binding sous Xaml (puisqu'il s'agit de chaînes de caractères non vérifiées à la compilation) pouvait représenter un réel danger. Dans ce billet je mettais aussi au défi les aficionados de Xaml de me citer de tête toutes les combinaisons possibles de l'extension Binding. Que cela soit parmi mes lecteurs ou les amis et clients à qui j'ai soumis cette question, je n'ai jamais eu de réponse complète. Preuve en est que la chose reste confuse et mérite certainement qu'on s'y attarde quelques instants.

Si l'on rapproche ces deux faits, à savoir que le Binding est au cœur de toute programmation effectuée en suivant les bonnes pratiques, et le constat que peu de développeurs maîtrisent totalement le sujet, on en arrive à la conclusion du billet évoqué plus haut : le code spaghetti fait son grand retour, ce qui peut être fatal à la technologie elle-même… (Je renvoie le lecteur à mon billet pour le détail du raisonnement qui est ici raccourci à l'extrême pour éviter les redites).

Comme je souhaite cette critique positive, car les errements font partie de l'adoption de toute nouvelle technologie, il m'a semblé naturel de faire le point sur ces problèmes et sur les solutions possibles pour les éviter.

Le présent article n'était donc pas réellement un cours sur le Binding au départ, plutôt une collection de conseils et des précisions de syntaxe. Mais au fil de l'écriture il est peut-être devenu le cours le plus complet qu'on puisse trouver à ce jour (et gratuitement) sur le sujet ! J'ai malgré tout choisi d'aborder cet article comme une aide à la compréhension et au débogue du Binding ce qui passe par un point syntaxique assorti d'exemples pour fixer les choses, et, je l'espère, rendre plus clair cette extension essentielle de Xaml.

Car étrangement, si le Binding lui-même est largement discuté et doctement disséqué par de nombreux auteurs, le côté pratique est souvent survolé et ses dangers et ambigüités sont systématiquement passés sous silence. Peut-être que la raison est qu'il est plus facile de parler *théoriquement* d'une nouveauté qu'on a par force peu pratiquée et dont on ignore les chausse-trappes alors qu'il est autrement plus difficile d'en parler *pratiquement* parce que l'on s'en sert au quotidien et qu'on a engrangé une véritable expérience. En tout cas c'est bien de vécu et de pratique quotidienne que je souhaite vous parler ici.



Le Binding

Fixons le décor. Qu'est-ce que le Binding, à quoi sert-il et comment le déclare-t-on ?

Définition

Le Binding (ou ligature) est un moyen de créer une connexion entre deux propriétés d'instances quelconques, une dans chaque instance. Le lien lui-même est matérialisé par une instance d'une classe spécialisée de Binding qui possède son paramétrage propre précisant le comportement du lien établi. La communication ainsi instaurée permet à une valeur de réagir immédiatement aux changements de l'autre (la valeur de la source est copiée dans la valeur cible) sans avoir besoin de programmer du code de gestion d'événement (typiquement `PropertyChanged` ou ses versions spécialisées comme `Checked` ou `UnChecked` d'un `CheckBox` par exemple).

Si la communication établie est totalement réciproque on parle alors de mode `TwoWay` (double sens), sinon de mode `OneWay` (sens unique).

Utilisations

Le Binding est largement utilisé pour coupler des classes métiers (des données au sens large) à l'interface utilisateur. L'avènement de patterns comme M-V-VM renforce encore l'importance du binding dans l'aide qu'ils apportent à la séparation nette entre interface utilisateur et code applicatif. Dans ce cas, même les commandes ou

les événements d'interface vont être « bindés » (liés) à des propriétés d'une classe particulière en charge des actions (le ViewModel dans MVVM). Je revoie le lecteur à la section

Références page 117 du présent document pour obtenir les adresses des billets et articles que j'ai écrits sur ces sujets.

Le principe est en fait très ancien, Delphi, VB, Java, les Windows Forms, toutes ces technologies ont proposé des moyens de créer du « binding ». Les plus anciennes utilisaient des classes spécialisées et le binding ne s'entendait que dans le cadre très restrictif d'une connexion à une base de données. Les technologies plus modernes comme Java, Windows Forms, ASP.NET, ouvraient la notion de binding en objectivant les sources de données, ce qui a marqué une étape décisive dans l'évolution du concept.

Créer un lien (généralement `TwoWay`, à double sens) entre des données et des éléments de l'IU est ainsi, historiquement en tout cas, la principale utilisation du Binding.

Xaml a repris l'objectivation du Binding des Windows Forms et d'ASP.NET en l'amplifiant et en créant une syntaxe particulière permettant de définir ces liens entre objets de façon déclarative au sein d'un format SGML[3]. La connotation base de données SQL est alors définitivement abandonnée. Si des données proviennent d'une telle source, c'est au travers de couches spécialisées (le DAL[4] par exemple) qui les transforment de toute façon en objets et listes d'objets (voire en grappes comportant des sous-graphes).

Si Windows Forms ouvrait la voie à une utilisation très étendue du Binding grâce à l'élargissement du concept de « source de données » à tout objet, Xaml a consacré ce qui n'était au départ qu'un artifice spécialisé à l'époque de Delphi en outil de première classe pour *développer autrement*. C'est cet « *autrement* » qui nous intéresse plus particulièrement aujourd'hui notamment au travers de MVVM et de constructions comme Prism, et qui, par l'utilisation accrue du binding révèlent plus encore ses difficultés de mise en œuvre et de débogue.

Schéma du principe

[3] SGML, voir http://fr.wikipedia.org/wiki/Standard_Generalized_Markup_Language
[4] D.A.L. : Data Access Layer, Couche d'Accès aux Données.

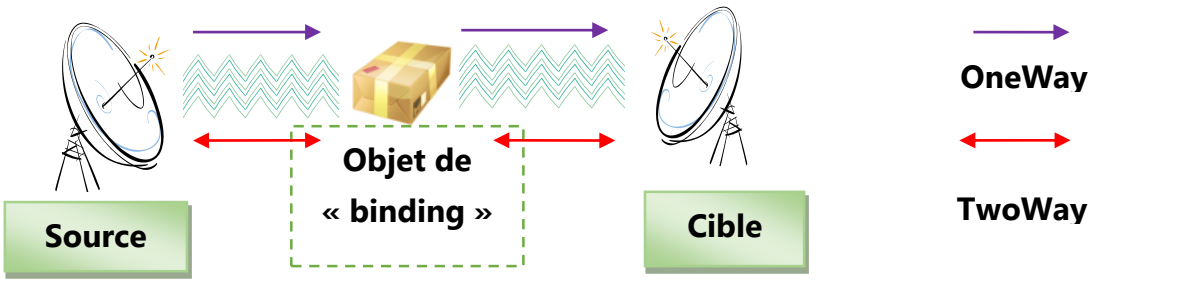


Figure 40 - Schéma de principe du Binding

La propriété d'un objet cible est liée à la propriété de l'objet source via un objet de binding. Cet objet contient les informations du lien et gère les changements de valeurs pour les propager dans un sens ou les deux selon le mode choisi.

Lorsqu'on définit un binding il y a toujours une source et une cible, même implicite. De fait le sens de la définition compte (principalement en mode `OneWay`). En mode `TwoWay` les deux propriétés peuvent être considérées à la fois comme source et cible.

Déclaration

Le Binding se déclare soit directement dans le code Xaml, soit par code classique (C#, VB.NET).

Par code

On peut définir un binding par code. Cela est parfois très utile même si cette façon de faire reste marginale.

Si on suppose un `TextBox` ainsi défini dans le code Xaml :

```
<TextBox x:Name="MaTextBox" />
```

On peut créer un binding entre la propriété `Text` de ce `TextBox` et n'importe quelle propriété d'un autre objet, par exemple la propriété `TimeOfDay` d'une variable `DateTime` :

```
// binding minimaliste
var currentTime = DateTime.Now;
var binding = new Binding("TimeOfDay")
        { Source = currentTime, Mode = BindingMode.OneWay };
MaTextBox.SetBinding(TextBox.TextProperty, binding);
```

On est ici dans le cadre le plus minimaliste qui soit. D'abord la source est un objet très simple (un `DateTime`), dans la réalité on trouvera plus souvent des objets métiers complexes ou un ViewModel[5], et ensuite le type de binding effectué est à sens unique (`OneWay`) ce qui veut dire que seul l'objet cible sera mis à jour.

Ce qui donnera l'affichage suivant :

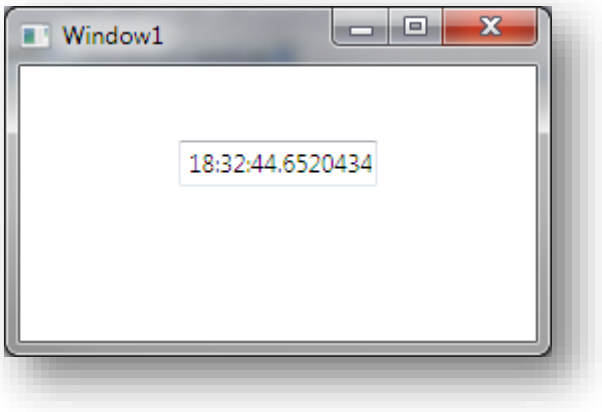


Figure 41 - Exemple de Binding minimaliste

Bien entendu, l'objet source étant une valeur fixe (nous prenons l'heure que nous stockons dans une variable qui est utilisée ensuite pour le Binding, donc une valeur figée), l'heure affichée n'évolue pas. C'est bien du Binding minimaliste, car la source n'implémente aucun mécanisme pour rafraichir la valeur. Un `DateTime` n'est pas un timer, juste un moyen de connaître la date et l'heure à un moment donné. De fait, le lien défini en mode `OneWay` (qui supporte le rafraichissement de la source pour le propager à la cible) est inutile et nous aurions pu préférer un mode `OneTime` (initialisation de la cible une seule fois).

Pour un affichage un peu plus vivant il faudrait définir une classe fournissant l'heure mais qui sache aussi prévenir quand celle-ci change. Nous allons voir cela dans l'exemple suivant (projet source : '`WpfMiniBinding`') qui contient une classe « `Horloge` » capable de remplir cette fonction.

En Xaml

La façon la plus fréquente de définir un binding est de le faire directement dans le code Xaml.

[5] Code gérant la logique d'une vue dans la pattern M-V-VM (voir les Références page 3 pour l'article complet à ce sujet)

Je vais utiliser ici une classe « Horloge » créée pour l'occasion. Cette classe supporte l'interface INotifyPropertyChanged qui déclenche PropertyChanged toutes les 500 ms par un timer interne. Elle expose une propriété Heure qui retourne l'heure depuis la classe DateTime. Il n'y a pas de stockage de l'heure, cette dernière est lue depuis DateTime à chaque accès à la propriété. L'événement PropertyChanged est déclenché arbitrairement deux fois par seconde par le timer. Il suffit dès lors de créer un binding avec la propriété Heure pour être averti deux fois par seconde que le contenu a changé, ce qui est exploité ici pour mettre à jour automatiquement un TextBox.

```
public class Horloge : INotifyPropertyChanged
    {
        private Timer timer;

        public Horloge()
        {
            timer = new Timer(500);
            timer.Elapsed += timer_Elapsed;
            timer.Start();
        }

        public string Heure
        {
            get { return DateTime.Now.TimeOfDay.ToString(); }
        }

        void timer_Elapsed(object sender, ElapsedEventArgs e)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new
                            PropertyChangedEventArgs("Heure"));
        }

        public event PropertyChangedEventHandler PropertyChanged;
    }
```

Code 13 - Définition d'un objet timer

Puisqu'il s'agit d'une définition en mode Xaml, nous n'allons pas créer l'instance de la classe Horloge par code C# (ce qui serait possible), nous allons directement le faire en Xaml en créant une ressource dans le corps de la fenêtre en cours :

```
<Window.Resources>
        <local:Horloge x:Key="Horloge" />
</Window.Resources>
```

XAML 25 - Instanciation d'un objet timer

L'espace de noms « local » est défini dans l'objet `Window` pour pointer sur l'assemblage du programme (exemple en WPF) :

```
xmlns:local="clr-namespace:WpfMiniBinding"
```

XAML 26 - Espace de nom local

Ensuite c'est dans la définition de `MaTextBox2` que nous allons lier la propriété `Text` de cette dernière à la ressource statique « `Horloge` ». Il faut préciser quelle propriété de cette classe nous souhaitons lier à `Text`. Ce qui s'exprimera de la façon suivante :

```
<TextBox
        Text="{Binding Source={StaticResource Horloge},
        Path=Heure, Mode=OneWay}"
        Name="MaTextBox2"
/>
```

XAML 27 - Binding TextBox et objet timer

Une fois cette définition en place, vous aurez la joie de voir l'heure défiler dans le `TextBox`, même en mode design sous Visual Studio (ou Blend) !

Les modes de binding

J'ai évoqué jusqu'ici quelques-uns des modes de binding, par exemple le mode `OneWay` ou le `TwoWay` qui sont les principaux utilisés. Mais Il existe plusieurs modes de binding adaptés à des situations bien précises. Un mauvais choix du mode de binding peut aussi entraîner des bogues fonctionnels sournois. Il est donc nécessaire d'avoir clairement à l'esprit l'utilité et la logique de chacun des modes disponibles et de toujours déclarer explicitement le mode.

Le mode OneTime

Le mode `OneTime`, « une fois », comme son nom l'indique va initialiser la valeur cible une seule fois, dès que le binding sera interprété s'il est décrit en Xaml, ou dès qu'il sera exécuté s'il est défini par code.

Ce mode est très utile pour les valeurs qui ne changent pas en cours d'exécution d'un programme (ou d'une fenêtre). Dans le projet exemple fourni avec l'article le premier `TextBox` relié à une variable `DateTime` l'est par un binding `OneTime` puisque la valeur d'origine ne changera jamais (le code du projet est ainsi construit, le code exemple plus haut utilise `OneWay`)

La logique du mode `OneTime` peut se voir comme suit :

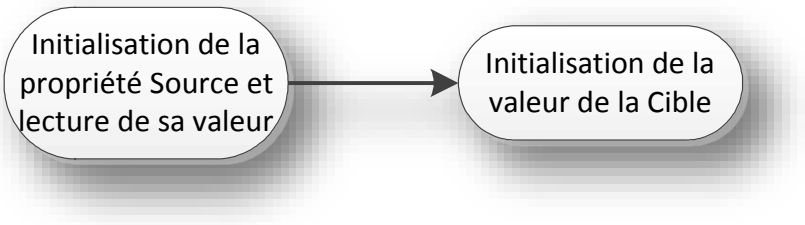


Figure 42 - Le mode OneTime

La séquence démarre par la reconnaissance de la source et elle s'arrête immédiatement après que la valeur de cette dernière a été reportée sur la valeur cible. Les événements de changement de la source qui pourraient intervenir sont ignorés.

Le mode OneWay

Dans ce mode la cible est initialisée puis l'objet de binding surveille les changements de valeur de la source et met à jour la cible si un tel événement se produit. Si la cible est modifiée par d'autres voies, la source reste inchangée.

Schématiquement le mode `OneWay` réagit de la façon suivante :

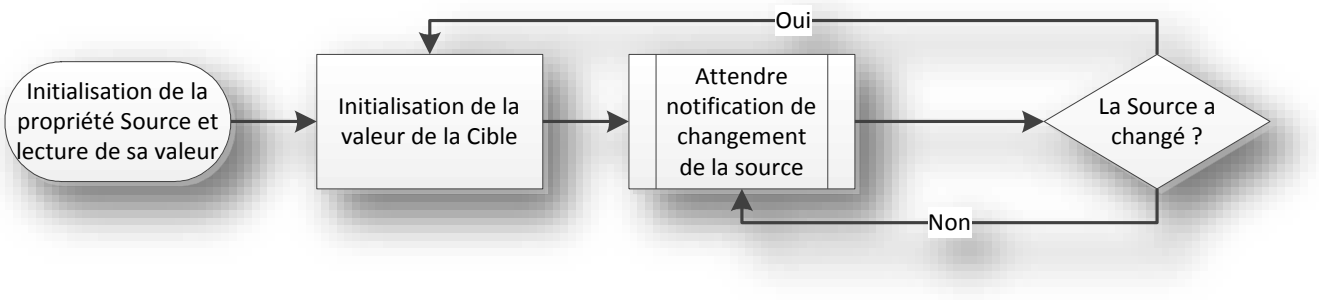


Figure 43 - Le mode OneWay

Le mode `OneWay` est d'utilisation fréquente lorsqu'il s'agit de relier une source de données à un affichage non interactif, par exemple un `TextBlock`, un `Chart`, etc.

Le mode TwoWay

C'est un mode destiné à tout binding de type interactif. Il définit un lien à double sens. Dans un premier temps la cible est mise à jour à partir de la valeur de la source. Comme dans le cas du `OneWay`, si la source change la cible sera mise à jour. Mais le suivi a aussi lieu dans l'autre sens : si la cible est modifiée la modification sera reportée sur la source.

Source et cible jouent ainsi un rôle qui semble équivalent mais il y a une subtile nuance dans la séquence d'initialisation qui fait que ces rôles sont bien distincts.

Schématiquement le mode `TwoWay` agit comme suit :

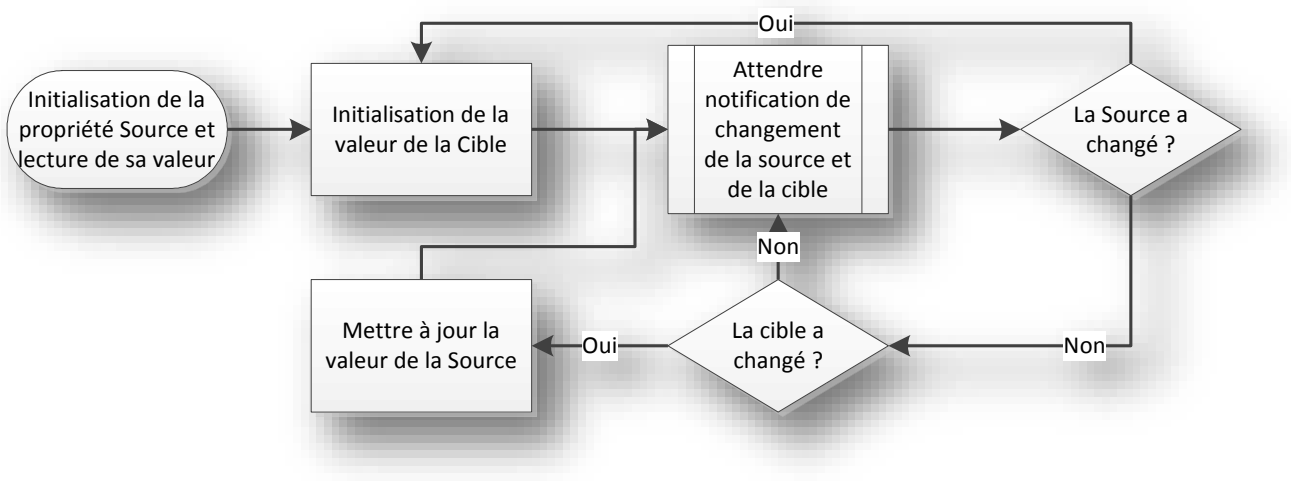


Figure 44 - le mode TwoWay

Gestion du timing

Le mode `TwoWay` propose une option intéressante mais délicate à utiliser, `UpdateSourceTrigger` qui permet de modifier le timing de la mise à jour de la source du binding lorsque la cible change. Je dis délicate à utiliser car utiliser cette option complexifie encore la balise de définition du binding et peut, par le degré de nuance introduit dans le comportement global être source d'erreurs. Sinon l'option en elle-même ne pose guère de difficultés particulières.

Pour illustrer son fonctionnement prenons un exemple.

Imaginons que la propriété `Text` d'un `TextBox` est reliée à la propriété `Nom` d'une instance de type `Personne` (un objet métier) c'est un binding `TwoWay` qui sera généralement utilisé : lorsque la fiche `Personne` est connectée à la cible `TextBox` il faut

que la valeur de la propriété `Nom` soit propagée à la propriété `Text` du `TextBox`. C'est le sens « naturel » : de la source vers la cible.

Mais lorsque l'utilisateur souhaite corriger le nom de la personne, c'est la cible qui est modifiée. Dans un système interactif on souhaite bien entendu que cette modification soit reportée sur la source (la propriété `Nom` de l'instance de `Personne`).

Plusieurs possibilités s'offrent à nous. La première serait de réagir au changement de contenu du `TextBox`, mais alors la source sera mise à jour à chaque frappe de caractère. Si le `Setter` de la propriété `Nom` de la classe `Personne` interdit par exemple les noms vides ou les noms de moins de deux caractères il sera impossible d'effacer le nom en cours pour en taper un autre… Au début le nom sera vide donc rejeté par la propriété `Nom`. De même corriger en gommant les derniers caractères d'un nom court risque d'échouer s'il reste moins de deux caractères (contrôle du `Setter` de `Nom`).

Dans le cas d'un `TextBox` on s'aperçoit que le timing offert par un simple `PropertyChanged` risque de provoquer plus d'ennuis qu'autre chose… Nous évoquons ici une propriété « `Nom` » dont le contenu est par définition assez ouvert. Imaginons une clé d'accès qui doit être formatée correctement avant d'être acceptée, c'est tout au long de la frappe que l'utilisateur recevra des messages d'erreur, jusqu'à ce qu'il termine sa frappe d'une clé bien formée…

Le Framework étant bien fait il nous permet grâce à l'option `UpdateSourceTrigger` de contrôler le type d'événement qui déclenchera la mise à jour de la source quand la cible changera. Pour un `TextBox` c'est l'événement `LostFocus` qui sera utilisé par défaut par exemple. Ces modes par défaut sont définis propriété par propriété par le concepteur du contrôle.

On dispose ainsi des options suivantes pour `UpdateSourceTrigger` :

- `Default`. Pour la majorité des propriétés c'est `PropertyChanged` qui déclenche la mise à jour. Mais on remarquera que la propriété `Text` de certains contrôles possède par défaut un autre comportement : la mise à jour est déclenchée sur `LostFocus`.
- `PropertyChanged`. La source est immédiatement modifiée dès que la cible change de valeur.
- `LostFocus`. La source est mise à jour uniquement quand la cible perd le focus.
- `Explicit`. La source n'est pas mise à jour automatiquement, il faut appeler explicitement `UpdateSource` qui est une méthode de l'objet `BindingExpression`.

La dernière option est assez tortueuse car lorsque le binding est défini en Xaml on ne dispose pas de l'objet de binding et encore moins de l'objet `BindindExpression`. Dans ce cas il faut récupérer ce dernier depuis l'objet qui définit le binding. Pour illustrer ce cas supposons un `TextBox` défini comme suit :

```
<TextBox Name="MaTextBox"
         Text="{Binding Path=LaPropriétéSource,
                UpdateSourceTrigger=Explicit}" />
```

XAML 28 - Binding avec Trigger de type Explicit

Pour faire un `UpdateSource` il faudra écrire le code suivant :

```
BindingExpression be =
                 MaTextBox.GetBindingExpression(TextBox.TextProperty);
                 be.UpdateSource();
```

Code 14 - Mise à jour d'une propriété bindée en Explicit

Tout cela complique les choses malgré tout et un code qui reposerait sur l'utilisation de telles astuces serait certainement très difficilement maintenables. Je vous conseille d'éviter ces options complexes qui répartissent de la logique à la fois dans le Xaml et le code C#. Mieux vaut tout gérer par code C#, en écrivant des gestionnaires d'événement appropriés. On y perd certes un peu dans la « démonstration de force » du programmeur mais on y gagne beaucoup niveau maintenabilité, et c'est un critère primordial pour un code professionnel. Bien entendu dans certains cas justifiés techniquement, bien encadrés et documentés se servir de toute la puissance du binding n'est pas interdit ! Le véritable danger est le risque que chaque développeur d'une équipe, selon ses compétences, utilise ou non certaines subtilités. On obtient au final un code hétérogène difficile à maintenir et où chaque intervention fait peser des risques de régression (imaginez que le développeur ayant la moins grande expérience du binding soit obligé d'intervenir sur une partie comportant des nuances qu'il ne comprend pas).

Le mode Default

Ce mode existe et il faut bien faire attention à lui ! En effet beaucoup de personnes pensent que lorsque le mode de binding n'est pas indiqué c'est le mode `OneWay` qui s'applique (d'autres d'ailleurs pensent que c'est le `TwoWay`), or par défaut, c'est le mode « `Default` » qui est utilisé ! Et son fonctionnement varie en fonction du mode de binding par défaut défini au niveau de la cible par le concepteur du contrôle !

Ainsi, un `CheckBox` a un mode par défaut `TwoWay`, ce qui semble logique mais qui n'est donc pas `OneWay`…

Pour éviter les erreurs de compréhension il est toujours préférable de fixer le mode explicitement. Se reposer sur la connaissance des modes par défaut de chaque contrôle n'est pas raisonnable pour un code professionnel qui se doit d'être maintenable facilement.

> *Une supplique un peu hors sujet : il en va de même pour les niveaux de précédences des opérateurs dans un calcul. Evitez le snobisme qui consiste à ne placer aucune parenthèse en basant toute la logique du calcul sur la précédence des opérateurs. Cela est difficile à maintenir, et se trouve être une source fréquente de régression : on ajoute un terme au calcul pour une évolution du code et c'est tout le calcul qui prend un autre sens. Ne jouez pas les kéké et placez toujours des parenthèses dans vos calculs, vous démontrerez ainsi votre professionnalisme bien plus qu'en rendant votre code impossible à maintenir ! J'ai vu maintes fois en audit des erreurs difficiles à déboguer provenir de ce style de programmation. La défense est toujours la même « si tu connais le langage ça ne pose pas de problème ». Ok, mais une salle de développement n'est pas une salle de musculation, on n'est pas là pour montrer ses muscles mais pour fabriquer quelque chose d'utile au sein d'un groupe hétérogène et faire en sorte que tout le monde puisse intervenir sur le code sans créer de problème. La programmation est un art collectif même quand on est seul devant son PC d'autres viendront peut-être un jour pour maintenir le code. Penser collectif est une grande preuve d'intelligence…*

Le mode OneWayToSource

Ce mode là est un peu une curiosité. Il n'existe pas sous Silverlight et j'avoue ne l'avoir jamais vu utiliser.

Dans le mode `OneWayToSource` on retrouve ainsi la même logique que dans le mode `OneWay` sauf que c'est la source qui est modifiée par la cible et non l'inverse.

Etrange et source de confusion et d'erreur, à éviter sauf si cela s'impose.

Car bien entendu parfois cela s'impose… Les développeurs du Framework ne sont ni fous ni stupides, loin s'en faut. Il y a forcément une raison intelligente à l'existence de ce mode de binding un peu étonnant.

Elle existe et bien peu la connaissent. Posons d'abord le fait que tout binding s'opère sur des propriétés de dépendances, et exclusivement sur des propriétés de ce type, en tout cas en ce qui concerne la source (voir mon article sur les propriétés de dépendances). Dans certains cas il peut s'avérer nécessaire de faire un binding avec une propriété CLR comme source. Ce n'est pas l'esprit de Xaml mais parfois on peut être « coincé » et ne pas avoir d'autres possibilités. Le Mode `OneWayToSource` autorise un tel binding et il est inversé car la propriété CLR ne peut pas être la source d'un binding elle ne peut donc être que la cible. Mais si on désire qu'elle joue le rôle de source il faut bien inverser le fonctionnement du binding lui-même. C'est là la raison d'être de ce mode étrange. S'en servir autrement n'est vraiment pas recommandé.

A noter que la définition d'un binding `OneWayToSource` propose la même option que le mode `TwoWay` concernant le timing du binding pour la mise à jour de la source. (Voir le mode `TwoWay`).

Mon conseil : si vous en arrivez à utiliser `OneWayToSource` c'est que votre code est mal pensé, effacez et recommencez !

Hiérarchie de valeur

Le binding, parce qu'il joue sur des propriétés de dépendance et parce qu'il n'est qu'une possibilité parmi d'autres de donner une valeur à une propriété de ce type, nécessite de bien comprendre la hiérarchie que le moteur des propriétés de dépendance utilise pour fixer la valeur « courante » d'une telle propriété.

Pour comprendre l'importance de cette hiérarchie reprenez l'exemple de code fourni et discuté plus haut dans cet article. Il se compose de deux `TextBox`, la seconde étant mise à jour par un objet « `Horloge` ». Sa propriété `Text` (qui est la cible) change ainsi à chaque fois que la propriété `Heure` de l'horloge change (c'est la source).

Telle qu'est conçue l'application, deux fois par seconde le texte du second `TextBox` change. C'est la conséquence directe du binding mis en place. Cela fonctionne même en conception comme nous l'avons vu.

Exécutez l'application. Le texte du second `TextBox` change régulièrement. Placer le focus sur ce contrôle et tapez quelques lettres (une suffit en réalité).

Que se passe-t-il ?

L'horloge est cassée…

En effet, le texte n'est plus mis à jour. Vous tentez d'effacer les caractères que vous avez tapés, mais rien n'y fait. Le texte ne change plus.

Pourquoi ?

C'est une conséquence des règles de précédence pour l'attribution de la valeur courante d'une propriété de dépendance.

Ici, en tapant quelque chose dans le `TextBox` vous avez créé une valeur « locale » pour la propriété `Text`. Or, une valeur locale à la précédence sur presque tout, dont un binding. Effacer les caractères saisis n'y change rien, plus vous agissez sur le `TextBox` plus vous confirmez que l'utilisateur souhaite qu'il prenne une valeur locale ! Rien n'y fait, l'horloge est cassée.

On pourrait penser que l'utilisation de `ClearValue` saurait, en supprimant la valeur locale, rétablir le fonctionnement de l'horloge. Il n'en est rien. Le binding n'existe plus. En réalité le binding est considéré comme une valeur locale. En en saisissant une nouvelle, ou en l'effaçant par `ClearValue` on détruit donc la valeur locale, donc le binding. Rien ne peut le faire revenir, sauf relancer l'application (ou reconstruire par code le binding)…

Vous comprenez facilement maintenant pourquoi je souhaite vous parler de cette hiérarchie : elle est en elle-même une possible source de bogues souvent difficiles à déceler.

La première des choses à comprendre est qu'une propriété de dépendance peut avoir plusieurs valeurs à la fois tant que ces valeurs sont placées sur des niveaux différents. Le moteur retourne toujours la valeur de plus haute précédence pour la valeur « courante ». Si la valeur d'un niveau supérieur disparaît, la propriété prend immédiatement la valeur du niveau juste en dessous. Il y a bien mémorisation de plusieurs valeurs distinctes pour une même propriété, mais uniquement pour des

valeurs placées sur des niveaux hiérarchiques différents. Au sein d'un même niveau toute nouvelle valeur remplace la précédente sans aucun effet mémoire.

Un binding étant considéré comme de même niveau hiérarchique qu'une valeur locale modifier cette dernière « tue » le binding sans espoir de retour en arrière. Alors que si, pendant que le binding est en place, nous jouons une animation qui ensuite s'arrête, la valeur affichée reviendra à celle du binding…

Règles de précédences

Les règles qui ont été posées sont assez logiques fonctionnellement parlant et ne sont pas très difficiles à mémoriser mais quand on entre dans les détails il existe beaucoup plus de niveaux que ce que la documentation MSDN laisse croire de prime abord. Car en dehors des niveaux hiérarchiques réels, qui sont peu nombreux, il existe des situations concurrentielles comme celle du binding que nous avons vue entre lesquelles il existe bien, dans la pratique, une hiérarchie fonctionnelle.

La hiérarchie présentée ci-dessous tient ainsi compte à la fois des niveaux « officiels » et des « sous hiérarchies pratiques ». La position 1 correspondant à la précédence la plus haute.

1. La valeur donnée à une propriété par une animation active, ou bien une animation terminée mais qui est en mode « `Hold` » (tenue de valeur).
2. La valeur locale.
3. La valeur provenant d'un binding agissant sur la valeur locale (binding classique).
4. La valeur issue d'un template appliqué à l'objet.
5. La valeur d'un binding d'un `property Setter` défini à l'intérieur d'un style
6. La valeur issue d'un style.
7. La valeur « ambiante » héritée de l'arbre visuel.
8. La valeur par défaut de la propriété elle-même.

Comme on le voit ici, si les choses sont logiques, la multitude des niveaux rend malgré tout la maîtrise des valeurs délicates dans une application réelle dès lors que saisies manuelles, animations, styles et templates viennent s'additionner à la valeur par défaut et à celle héritée de l'arbre visuel ! L'ordre dans lequel la séquence d'attribution des valeurs sera jouée aura aussi son importance.

Pour résumer, une valeur obtenue par binding est en réalité une valeur locale et possède le même niveau de précédence qu'une telle valeur. Le fait d'utiliser `SetValue`

ou de modifier la propriété autrement en lui donnant une nouvelle locale remplace et annule le binding en cours (qui est de même niveau hiérarchique). C'est pour cette raison que j'ai placé la « vraie » valeur locale un cran au-dessus dans la hiérarchie que la valeur obtenue par binding. Dans la pratique la valeur locale « prend le dessus » et annule le binding. En revanche c'est parce que le binding est bien une valeur locale que fixer une nouvelle valeur locale annule totalement le binding. En revanche, les valeurs qui sont situées sur des niveaux hiérarchiques différents peuvent coexister et un « retour en arrière » est toujours possible. Par exemple si un rectangle est rouge par une opération de binding sur une couleur ou une brosse et qu'une animation est lancée pour le rendre vert, en fin de celle-ci (si elle n'est pas en mode `Hold`), le binding ne sera pas cassé et le rectangle redeviendra rouge.

Bien comprendre la précédence des valeurs des propriétés de dépendance est un préambule indispensable à leur bonne utilisation. Et cela joue une grande importance dans certains bogues de binding, d'où ces explications.

La notion de DataContext

La notion de `DataContext` est directement et intimement liée à celle de binding puisqu'elle permet de fixer une source globale (un objet, une liste…) dans laquelle tous les bindings définis ensuite dans sa portée iront piocher les propriétés pour se lier. L'utilisation d'un `DataContext` n'est pas obligatoire pour faire du binding, mais les avantages qu'il procure dans de nombreuses situations font qu'il est très utilisé.

`DataContext` est avant tout une propriété de la classe `FrameworkElement` dont héritent tous les contrôles. Sa valeur peut être définie par code behind ou en Xaml. On peut même définir un binding sur un `DataContext` en tant que propriété cible, les règles de précédences s'appliquent aussi puisque `DataContext` est une propriété de dépendance.

`DataContext` accepte comme valeur un `object`. C'est-à-dire tout ce qu'on veut. Cette souplesse peut être source d'erreur. Accepter une `object` et non une classe précise et ses descendants fait qu'il ne peut y avoir aucun contrôle de type sur l'objet transmis au `DataContext`. Une propriété de type `object` est un peu l'équivalent côté code behind des paramètres en chaînes de caractères de Xaml, une grande souplesse mais aucun contrôle à la compilation. C'est donc une source possible de bogues, d'erreurs diverses et variées qui ne pourront être détectées que par des tests systématiques de

l'application en utilisation réelle. C'est principalement pour attirer votre attention sur ces problèmes potentiels que je vous parle du `DataContext`, le but n'est pas de remplacer la documentation ou les articles sur le sujet que vous trouverez sur le Web.

Le fait de faire pointer un `DataContext` sur un « mauvais » objet ne générera aucune erreur, et selon l'utilisation qui est faite des bindings, il sera parfois très difficile de voir qu'il y a eu confusion. Vigilance, toujours, donc…

Le `DataContext` permet d'offrir une source cohérente contenant toutes les propriétés nécessaires à une interface visuelle ou à une partie de celle-ci. L'utilisation du `DataContext` dans un tel cadre est à la base de patterns comme Model-View-ViewModel. Cela justifiait aussi de l'aborder, même brièvement.

Les convertisseurs de valeur

Ils sont au cœur du binding et il est impossible de parler de ce dernier en passant sous silence la notion de convertisseur.

Tout au long de cet article je parle de convertisseurs, plusieurs fois j'en utilise dans la mise en œuvre des exemples. Il est donc important d'en comprendre l'utilité et l'implémentation.

Définition

Le binding permet de connecter deux propriétés quelconques ensemble. Les types de ces propriétés peuvent eux aussi être quelconques et différents. Si le Framework possède des automatismes de conversion, ces derniers ne peuvent couvrir la totalité des besoins créés par le binding. Il est donc nécessaire d'offrir un mécanisme permettant de convertir une valeur source en une autre valeur compatible avec le type de la cible du binding.

C'est le rôle des convertisseurs de valeur. Ils prennent une valeur d'entrée et la transforment pour qu'elle devienne une valeur acceptable par la cible en sortie. Le mécanisme complet supporte la conversion dans les deux sens même si cette utilisation est assez rare.

Scénario

Supposons un rectangle posé sur une fiche ainsi qu'un `Slider` prenant les valeurs de `1` à `10`. Imaginons que nous voulions que le rectangle soit caché (`Visibility.Collapsed`) lorsque la valeur du `Slider` est inférieure à 6 et qu'il soit affiché dans le cas contraire (`Visibility.Visible`).

Naturellement on ne souhaite pas gérer cette situation « à l'ancienne » c'est-à-dire en programmant un gestionnaire d'événement sur le changement de valeur du `Slider`, code qui modifierait la visibilité du rectangle. Ce n'est pas une question de mode, il s'agit plutôt de bonnes pratiques et de séparation forte entre code et interface visuelle. L'implémentation d'un gestionnaire d'événement créera une dépendance forte entre code et visuel alors que l'implémentation d'un convertisseur autorisera une meilleure séparation ainsi qu'une réutilisation du code bien plus grande.

Dans une vision plus moderne et plus conforme à l'esprit de Xaml donc, nous souhaitons utiliser un binding entre la propriété `Visibility` du rectangle et la propriété `Value` du `Slider`.

Seulement ces deux propriétés ne sont pas compatibles entre elles. Aucun automatisme ne peut savoir à partir de quelle valeur du `Slider` il faut considérer que le rectangle est visible ou non.

C'est là qu'interviennent les convertisseurs de valeur. Dans le cas précis de notre exemple fictif du rectangle il faudra écrire un convertisseur acceptant en entrée un `double` (la valeur du `Slider`) et produire en sortie une valeur de type `Visibility` en respectant les contraintes indiquées.

Ce genre de situation est très fréquent sous XAML et les projets possèdent généralement un répertoire dédié contenant de nombreux convertisseurs.

On notera que l'avènement du pattern MVVM et le fait que le ViewModel peut jouer le rôle d'adaptateur de valeur pour sa Vue rendent l'utilisation des convertisseurs plus marginale qu'aux débuts de WPF par exemple. Ils continuent de jouer un rôle important mais leur nombre est aujourd'hui sensiblement réduit le plus souvent.

Implémentation

Les convertisseurs de valeur sont des classes tout ce qu'il y a de plus basiques. Leur seule contrainte : implémenter l'interface `IValueConverter` pour des bindings simples, ou `IMultiValueConverter` pour les bindings multiples.

Ces interfaces exposent deux méthodes : `Convert` et `ConvertBack`. La première est systématiquement utilisée, la seconde plus rarement (certaines transformations n'étant pas facilement réversibles et de nombreux bindings ne sont pas à double sens).

Voici un exemple typique de convertisseur :

```
class BoolToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object
                          parameter, CultureInfo culture)
    {
        if (targetType != typeof(Visibility)) return null;
        bool v;
        if (value is bool?) v = ((bool?)value).HasValue ?
                                ((bool?)value).Value : false;
        else
            if (value is bool) v = (bool)value;
            else v = false;
        return v ? Visibility.Visible : Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, CultureInfo culture)
    {
        if (targetType != typeof(bool) &&
            targetType != typeof(bool?)) return null;
        if (!(value is Visibility)) return false;
        return (Visibility)value == Visibility.Visible ? true : false;
    }
```

Code 15 - Exemple de convertisseur de valeur

Le code ci-dessus sait convertir un booléen en une valeur `Visibility`. Si le booléen est vrai la visibilité retournée sera `Visible`, sinon `Collapsed`.
On remarque que la conversion inverse a été programmée d'abord à titre d'exemple et car il n'est pas insensé de convertir une visibilité en booléen (même si le caractère binaire du booléen oblige à réduire les trois états de la visibilité – `visible`, `hidden`, `collapsed` – à deux états, `visible` ou `collapsed`).

Le convertisseur reçoit dans ses méthodes une valeur à convertir (le premier paramètre) sous la forme d'un `object`, il reçoit aussi le type de la cible, un éventuel paramètre et une culture optionnelle.

Cette dernière permet de forcer la prise en charge d'une culture donnée indépendamment de celle de l'application. Le cas se rencontre assez rarement.

Le paramètre optionnel est un `object`, on peut ainsi utiliser un paramétrage aussi complexe que nécessaire en passant une instance de classe contenant ces derniers.

Utilisation

Une fois codés en C# (ou autre), les convertisseurs sont utilisés au sein des balises de binding de Xaml.

Instanciation

Pour utiliser un convertisseur il faut qu'une instance existe, ce qui paraît logique. Il existe plusieurs façons de créer cette dernière.

La plus commune consiste à créer une ressource dans la fenêtre en cours. Si le convertisseur est utilisable en de nombreux endroits de l'application certains développeurs préfèrent alors le placer dans `App.xaml`.

Après avoir déclaré le namespace on trouvera une déclaration de ce type dans les ressources :

```
    <Window.Resources>
        <conv:BoolToVisibilityConverter x:Key="BoolConv" />
    </Window.Resources>
```

XAML 29 - Instanciation d'un convertisseur de valeur

Le namespace est déclaré comme suit :

```
xmlns:conv="clr-namespace:ConverterSample.Converters"
```

Bien entendu le nom exact du namespace dépend de votre application. Le raccourci `conv` est aussi purement arbitraire. Toutefois il est judicieux de placer tous les convertisseurs dans un même namespace et de les déclarer avec un raccourci propre plutôt que d'utiliser le namespace de l'application comme un fourre-tout.

Invocation

On invoque un convertisseur au sein d'un binding. Il est difficile de vous présenter un binding sans explication mais l'étude de la nature et de la syntaxe de tous les types de bindings est l'un des sujets principaux de cet article, vous trouverez donc toutes les explications dans le présent document. Pour l'instant concentrons-nous sur le convertisseur.

Et pour illustrer le propos regardons une capture du projet `ConverterSample` fourni avec l'article :

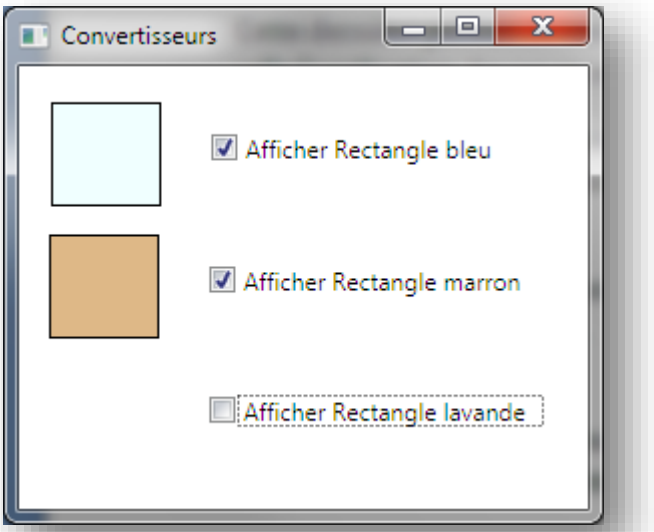


Figure 45 - Convertisseur de valeur en action

Dans cette application nous trouvons des carrés de couleur et des `CheckBox` permettant de les afficher ou de les cacher.

L'exemple qui nous concerne pour l'instant est celui du premier carré (bleu).

Le code Xaml de sa déclaration est le suivant :

```
<Rectangle Height="52" HorizontalAlignment="Left" Margin="16,18,0,0"
   Name="rectangle1" Stroke="Black" VerticalAlignment="Top"
   Width="55" Fill="Azure"
   Visibility="{Binding ElementName=checkBox1, Path=IsChecked,
                        Converter={StaticResource BoolConv}}"/>
```

XAML 30 - Utilisation d'un convertisseur de valeur

La partie importante est celle qui déclare le binding sur la propriété `Visibility` du Rectangle. Elle est liée à la `checkBox1`, sur sa propriété `IsChecked`. Comme les booléens ne sont pas directement compatibles avec le type `Visibility` (une énumération de même nom que la propriété) il est nécessaire de fournir un convertisseur. C'est ce que nous faisons en indiquant l'instance du convertisseur qui a été déclaré dans les ressources de la fenêtre sous le nom de `BoolConv`. Le code du convertisseur est celui donné en exemple un peu plus haut.

Bonnes pratiques

Il n'y a que rarement des règles absolues en informatique, surtout dans des environnements aussi riches que WPF ou WinRT. Il ne peut y avoir que des conseils, des bonnes pratiques éprouvées par l'expérience.

Concernant les convertisseurs les questions qui se posent sont les suivantes :

- Où faut-il les déclarer ?
- Où faut-il les instancier ?

A la première question nous avons déjà répondu en proposant de tous les placer dans un sous répertoire dédié du projet et dans un espace de nom qui leur est propre.

Dans certains projets il peut même s'avérer intéressant de les regrouper dans un même projet géré à part de type Dll, assemblage que les autres applications pourront ensuite partager.

La seconde question est moins évidente à trancher. Les convertisseurs ne consomment pas beaucoup de mémoire, mais si on en utilise beaucoup il n'est peut-être pas forcément utile de tous les placer dans `App.xaml` (leur cycle de vie devient alors celui de l'application). Et puis `App.xaml` peut très vite devenir un fatras inextricable si on n'y prend garde.

L'exemple que nous venons d'étudier créée l'instance sous la forme d'une ressource de la fenêtre. Cette méthode a l'avantage de « localiser » les convertisseurs et de n'instancier que ceux qui sont exploité par la fenêtre. Lorsqu'elle disparaîtra le ou les convertisseurs utilisés seront libérés.

Toutefois s'il s'agit de la fenêtre principale de l'application ou d'une fenêtre (ou un `UserControl` sous Silverlight) qui n'est jamais détruite, l'avantage du cycle de vie « localisé » devient caduque.

Ainsi, certains développeurs préfèrent créer une classe statique regroupant tous les convertisseurs. Ces derniers sont instanciés systématiquement ou bien à la première utilisation (ce qui est plus économe surtout s'il y a beaucoup de convertisseurs et qu'ils ne sont pas forcément tous exploités). Cette stratégie s'implémente comme suit :

```
    static class Converters
    {
        private static BoolToVisibilityConverter boolToVisibility;

        public static BoolToVisibilityConverter BoolToVisibility
        {
            get
            {
                if (boolToVisibility == null)
                    boolToVisibility = new BoolToVisibilityConverter();
                return boolToVisibility;
            }
        }

    }
```

Code 16 - Création d'un point d'accès global à tous les convertisseurs

La classe statique `Converters` regroupe tous les convertisseurs sous la forme de propriétés statiques. Les instances sont créées une fois pour toute mais uniquement lors de la première utilisation. Les convertisseurs inutilisés ne sont pas créés inutilement.

L'application exemple ne comportant qu'un seul convertisseur c'est bien entendu le principe qu'il faut retenir et non son exploitation dans ce cadre précis.

Lorsque les convertisseurs sont réunis dans une classe statique, leur utilisation dans les bindings est légèrement différente. La déclaration du namespace ne change pas (c'est un choix dans l'application exemple), et le binding devient ainsi :

```
<Rectangle Fill="BurlyWood" HorizontalAlignment="Left" Margin="15,84,0,0"
Name="rectangle2" Stroke="Black" Width="55" Height="52"
VerticalAlignment="Top"
Visibility="{Binding ElementName=checkBox2, Path=IsChecked,
    Converter={x:Static conv:Converters.BoolToVisibility}}" />
```

Code 17 - Accès à un convertisseur de valeur global

On notera la syntaxe particulière permettant de spécifier l'utilisation d'une classe et d'une propriété statiques.

Dans l'application exemple il s'agit du second carré, le marron.

Enfin, d'autres développeurs préfèrent ne pas utiliser la stratégie de la classe statique et mettent en œuvre le pattern singleton dans leurs convertisseurs.

Pour illustrer ce cas (le troisième carré de l'application exemple, couleur lavande) nous allons construire un convertisseur implémentant le pattern singleton à partir de la classe déjà utilisée :

```
class BoolToVisibilitySingletonConverter : BoolToVisibilityConverter
{
    private BoolToVisibilitySingletonConverter()
    { }

    private static readonly BoolToVisibilitySingletonConverter
                instance = new BoolToVisibilitySingletonConverter();

    public static BoolToVisibilitySingletonConverter Instance
    {
        get { return instance;}
    }
}
```

Code 18 - Définition d'un convertisseur de valeur sous la forme d'un Singleton

La nouvelle classe hérite du convertisseur utilisé précédemment. Cela par pur souci pratique dans la démonstration, dans la réalité c'est bien le premier convertisseur qui aurait été d'emblée implémenté sous la forme d'un singleton si tel était le choix. L'application exemple gère trois stratégies en même temps et cette astuce évite de dupliquer le code ou de créer trois projets distincts.

La façon dont le pattern singleton est implémenté est classique : le constructeur de la classe est rendu privé pour interdire toute instanciation directe, et une propriété statique `Instance` est proposée pour accéder au convertisseur. On notera qu'on pourrait ici aussi implémenter l'instanciation au premier appel plutôt qu'une instanciation systématique.

Lorsque le convertisseur est implémenté sous la forme d'un singleton, son utilisation en Xaml est très proche de la version précédente (celui de la classe statique regroupant tous les convertisseurs). Ainsi, le troisième carré de couleur lavande est codé de cette façon :

```
<Rectangle Fill="Lavender" HorizontalAlignment="Left" Margin="15,149,0,0"
Name="rectangle3" Stroke="Black" Width="55" Height="52"
VerticalAlignment="Top"
Visibility="{Binding ElementName=checkBox3, Path=IsChecked,
Converter={x:Static conv:BoolToVisibilitySingletonConverter.Instance}}" />
```

XAML 31 - Utilisation d'un convertisseur défini comme Singleton

Pour résumer

Les convertisseurs sont des briques indissociables du binding. Bien comprendre leur fonctionnement, leur implémentation et choisir correctement la stratégie d'instanciation peut avoir un effet visible sur certaines applications. Nous aurons l'occasion de revenir sur leur rôle plusieurs fois encore dans cet article.

Les dangers du Binding

Je ne reprendrai pas le détail de mon billet cité en introduction de l'article, mais pour fixer les choses je rappellerai deux points principaux des problèmes soulevés par le Binding tel qu'il a été implémenté dans Xaml et qu'illustre à merveille le petit exemple de code du paragraphe « Le Binding , Déclaration » page 124 :

1- Même dans sa version C#, la définition d'un binding repose sur le passage d'un nom de propriété sous forme de chaîne de caractères non contrôlée à la compilation.
2- Dans sa version totalement Xaml, et même sur un exemple aussi simple que celui de l'horloge, outre le problème précédent, on voit apparaitre une syntaxe alambiquée pleine d'accolades américaines, de virgules et d'arguments bien difficiles à retenir d'autant plus que tout cela change selon le contexte comme nous le verrons.

Des chaînes non contrôlées

Dans le premier exemple (définition du binding par code C#) on voit clairement que lors de la création de l'instance de la classe de binding nous passons la chaîne « `TimeOfDay` ». **Aucun contrôle n'est effectué et aucune erreur d'exécution ne sera levée si cette chaine n'est pas valide !**

Oublions une majuscule ou ajoutons une lettre et plus rien ne marchera. Dans le projet exemple changez dans la définition du binding Xaml une lettre à `TimeOfDay`, ajoutez un « `s` » à la fin par exemple. Relancez l'application… La `TextBox` sera vide, c'est tout. Ici le problème est visible comme le nez au milieu de la figure, il n'y a que deux `TextBox` (la fixe, et la seconde qui évolue dans le temps). Il est facile de voir qu'il y a un problème. Imaginez seulement un instant les ravages de ce type de bug dans une application réelle de 200 écrans, contenant des milliers de contrôles et des codes Xaml de plusieurs centaines de lignes (donc incontrôlables à l'œil)…

Ce problème existe d'ailleurs à d'autres endroits dans le Framework, notamment dans une fonctionnalité de base : `INotifyPropertyChanged` qui réclame le passage du nom de la propriété modifiée sous la forme de chaîne de caractères. J'ai vu de très nombreux codes montrer des dysfonctionnements très difficiles à déceler uniquement à cause de cette particularité (toujours pareil, on refactore en changeant

les noms des propriétés mais on oublie, ce qui est humain, de mettre à jour la chaîne de caractères utilisée par l'événement `PropertyChanged`).

Ces « trous » dans la sécurité du code au sein d'un édifice aussi bien ficelé que le Framework .NET est comme une planche savonneuse sur laquelle le développeur XAML devra marcher toute la journée sans se casser la figure. Choix conceptuel étonnant et pour le moins risqué.

Il existe malgré tout des moyens de contrôler les erreurs de binding, nous verrons cela plus loin au chapitre des solutions.

Un langage dans le langage

Ce n'est plus un nom de propriété qui est passé en chaîne de caractère mais c'est tout un mini langage de programmation avec imbrication de chaînes et d'accolades que nous réserve la syntaxe du Binding ! Ici la planche est toujours savonneuse, mais sur les deux faces, et en plus vous avez les yeux bandés. Difficile de rester debout !

Plus sérieusement, on comprend la tentation d'utiliser des chaînes de caractères interprétées pour étendre la puissance du langage Xaml tout en lui conservant sa structure assez simple de fichier SGML. Malgré la richesse du concept (assez ancien maintenant) ayant donné naissance à des langages comme Html ou XML et donc Xaml, il faut avouer que SGML n'a peut-être pas été pensé pour tant d'extensions et que Xaml flirte certainement avec les limites conceptuelles de ce format.

Toutefois il existe une réponse à ce problème ponctuel : comprendre et connaître la syntaxe du Binding dans ces différentes tournures. C'est ce que nous verrons au chapitre des solutions.

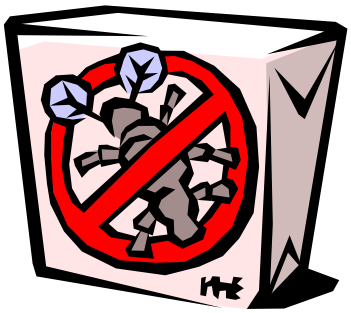
On fait quoi ?

La plus belle des filles (ou le plus beau des hommes, ne soyons pas sexistes) finira toujours par révéler quelques petits défauts cachés… Si on est amoureux on fermera les yeux et on s'adaptera.

Xaml n'est pas parfait, il faut l'avouer, mais une fois ce constat posé, que pouvons-nous faire concrètement ?

Comme nous sommes animés par des sentiments purs et passionnés, nous allons tout simplement tenter de composer avec ses faiblesses. Et des moyens existent. C'est ce que nous allons voir maintenant.

Déboguer le Binding



Vigilance

Règle d'or, la vigilance.

Comme il ne s'agit pas d'abandonner le binding (pas plus que `INotifyPropertyChanged`), il va falloir fournir un effort constant de vigilance. Ne pas hésiter à vérifier le code Xaml même quand il est verbeux. Les utilisateurs de Blend sont particulièrement concernés car l'environnement a été entièrement conçu dans un but : écrire du Xaml à votre place. De fait on peut travailler des heures sur un projet complexe sans même jamais voir une ligne de Xaml. C'est le gros avantage de Blend, notamment parce qu'il permet à des infographistes ou des intégrateurs d'éviter le contact un peu rugueux avec Xaml. Mais sous Blend aussi on peut voir le code Xaml et agir sur lui, et bien souvent il ne faut pas s'en priver !

Une code Xaml court

Le meilleur code Xaml est le plus court possible… Dans ce dédale de balises, le seul moyen de conserver une possibilité humaine de contrôle est encore de se débrouiller pour créer des fichiers Xaml de petite taille. Pour cela plusieurs moyens :

- Créer des `UserControl` ayant des rôles clairs et précis, quitte à imbriquer plusieurs `UserControl` pour en créer un plus gros assurant la totalité de la fonctionnalité.
- Créer des dictionnaires de ressources pour chaque ressource. Un dictionnaire pour la palette de couleurs et les brosses, un autre pour le template du `CheckBox`, un autre pour le template du `RadioButton`, etc. Eviter les gros fichiers « skins » incontrôlables humainement.
- Si vous travaillez sous Blend, n'hésitez pas à revisiter la balise du contrôle que vous venez de toucher : supprimer les numériques ayant des tonnes de chiffres après la virgule. Une position par exemple (ou une `Margin`) n'a pas besoin de 15 décimales, sauf cas exceptionnel qui reste à trouver. Pourtant ce sont des Double qui sont utilisés et souvent à force de manipulation ces nombres possèdent une partie fractionnaire inutile qui brouille la lisibilité.

Refactoring sous contrôle

Tout refactoring, notamment un changement de nom de classe ou de propriété doit être suivi d'une vérification manuelle. Pour cela je vous incite à toujours effectuer une simple recherche du nom original dans toute la solution. On est parfois surpris de

découvrir qu'il se cache des utilisations qu'on ne soupçonnait pas, surtout dans de gros logiciels écrits à plusieurs !

Utiliser des outils intelligents

Visual Studio est certainement l'EDI le mieux conçu, mais il y a de la place pour des améliorations. Vous pouvez ainsi utiliser des outils comme Resharper (voir adresse dans les références) qui proposent un degré d'intelligence supérieur au refactoring de base de Visual Studio, notamment en proposant de chercher dans les chaînes et même dans les commentaires.

Utiliser Blend

C'est tout bête, mais je vois encore beaucoup de développeurs qui font du XAML sans utiliser Blend ! Cela m'apparaît impossible pour créer des compositions visuelles et des animations dignes de ce nom. Blend simplifie aussi beaucoup le binding en écrivant le code Xaml à votre place… Le meilleur moyen de produire un code de binding sans erreur est encore d'effectuer les liens sous Blend. VS 2012 ou 2013 apportent néanmoins quelques plus non négligeables par rapport aux anciennes versions qui ne géraient pas très bien tout cela.

Les autres conseils s'appliquent malgré tout en cas de refactoring (à faire sous VS avec Resharper ou équivalent) ou en cas de modification d'un binding. Utiliser Blend n'exonère pas totalement le développeur d'une bonne connaissance de Xaml ni même d'utiliser Visual Studio qui propose des aides d'écriture et de contrôle du code (Xaml et C#) bien supérieures (comme IntelliSense).

Vérifier les erreurs de binding dans la fenêtre de sortie

Je vous expliquais qu'une erreur dans les chaînes de caractères définissant un binding ne générait pas d'erreur à l'exécution. C'est vrai du point de vue du logiciel qui tourne, mais il y a bien des erreurs qui sont soulevées… Elles ne sont pas fatales, ne soulèvent pas d'exceptions qu'on peut gérer avec un `Try/Catch`, elles se cachent dans la fenêtre de sortie de Visual Studio !

Hélas cette fenêtre est un peu un fourre-tout qui se remplit de messages tout au long de l'exécution d'une application sous débogueur. Mais rien n'interdit de faire un copier-coller de son contenu dans le bloc-notes pour y chercher certains messages particuliers grâce à un simple Ctrl-F…

Car dans l'exemple de l'horloge, si au lieu d'un binding avec « `TimeOfDay` » j'écris « `TimeOfDays` », si aucune erreur d'exécution n'est levée, dans la trace de sortie on peut trouver le message suivant :

```
System.Windows.Data Error: 39 : BindingExpression path error: 'TimeOfDays'
property not found on 'object' ''DateTime' (HashCode=-2099349868)'.
BindingExpression:Path=TimeOfDays; DataItem='DateTime' (HashCode=-
2099349868); target element is 'TextBox' (Name='MaTextBox'); target
property is 'Text' (type 'String')
```

Toujours aussi déroutant… Là où on croyait être tombé dans un no man's land on se retrouve avec des messages d'erreur d'une précision diabolique !

Nous connaissons ici le type d'erreur, c'est à l'intérieur d'un `BindingExpression`, dans son `Path`, la propriété `TimeOfDays` n'est pas trouvée sur l'objet `DateTime`. De fait la cible qui est un `TextBox` portant le nom `MaTextBox` ne verra pas sa propriété `Text` liée à la source.

Vous comprenez maintenant qu'un simple « Chercher » dans toute la fenêtre de sortie de phrases comme « `System.Windows.Data Error` » ou « `BindingExpression path error` » peut vous permettre de relever sinon toutes au moins de nombreuses erreurs de binding causées par une chaine de caractères défaillante même au sein d'une grosse application.

Créer un fichier des erreurs de Binding

L'approche précédente, basée sur un copier-coller de la fenêtre de sortie dans le bloc-notes et assortie d'une recherche peut être largement améliorée sous WPF depuis les versions 3.0 et 3.5.

Le système de Trace Source repose sur le fichier de configuration de l'application (`App.config`) par l'ajout de certaines balises. Voici un exemple d'un tel fichier modifié pour assurer l'enregistrement de la fenêtre de sortie dans un fichier texte (voir le projet exemple fourni) :

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>

      <source name="System.Windows.Data" switchName="SourceSwitch" >
        <listeners>
          <add name="textListener" />
        </listeners>
      </source>
    </sources>

    <switches>
      <add name="SourceSwitch" value="All" />
    </switches>

    <sharedListeners>
      <add name="textListener"
      type="System.Diagnostics.TextWriterTraceListener"
      initializeData="BindingDebugTrace.txt" />
    </sharedListeners>

    <trace autoflush="true" indentsize="4"></trace>

  </system.diagnostics>

</configuration>
```

Code 19 - Configuration d'une trace de débogue WPF

On peut y fixer des filtres, par exemple ici le filtrage se fait sur les messages de `System.Windows.Data`. Si vous souhaitez déboguer des animations c'est plutôt l'assemblage `System.Windows.Media.Animation` que vous indiquerez par exemple. Cela évite d'être noyé dans des tas de messages qui pourraient être gênant pour voir les erreurs recherchées.

Comme nous voulons voir tous les messages d'erreur nous indiquons « `All` » dans la section des `switches`. Les valeurs possibles sont `Off`, `Error`, `Warning`.

Je souhaite que les erreurs soient écrites dans un fichier texte « `BindingErrorTrace.txt` », c'est ce qui est indiqué dans la section `sharedisteners`. On pourrait choisir d'autres Listeners, par exemple `ConsoleTraceListener` qui affiche

les messages à la console, ou bien choisir un format de sortie de type XML ce qui en faciliter le traitement automatiquement par une « moulinette » que vous écrirez pour l'occasion…

Vous pouvez vous référer à l'excellent billet de Mike Hillbergs « Trace sources in WPF » pour avoir plus de précision sur ce mécanisme de log intégré et les subtilités de son paramétrage (voir l'adresse dans la liste des références en début d'article).

En relançant l'application (toujours avec l'erreur du « `s` » ajouté à `TimeOfDay` dans le binding en C#), nous obtenons dans le folder `bin\debug` le fichier texte attendu dont le contenu est :

```
System.Windows.Data Error: 39 : BindingExpression path error: 'TimeOfDays'
property not found on 'object' ''DateTime' (HashCode=1822811999)'.
BindingExpression:Path=TimeOfDays; DataItem='DateTime'
(HashCode=1822811999); target element is 'TextBox' (Name='MaTextBox');
target property is 'Text' (type 'String')
System.Windows.Data Information: 19 : BindingExpression cannot retrieve
value due to missing information. BindingExpression:Path=TimeOfDays;
DataItem='DateTime' (HashCode=1822811999); target element is 'TextBox'
(Name='MaTextBox'); target property is 'Text' (type 'String')
System.Windows.Data Information: 20 : BindingExpression cannot retrieve
value from null data item. This could happen when binding is detached or
when binding to a Nullable type that has no value.
BindingExpression:Path=TimeOfDays; DataItem='DateTime'
(HashCode=1822811999); target element is 'TextBox' (Name='MaTextBox');
target property is 'Text' (type 'String')
System.Windows.Data Information: 10 : Cannot retrieve value using the
binding and no valid fallback value exists; using default instead.
BindingExpression:Path=TimeOfDays; DataItem='DateTime'
(HashCode=1822811999); target element is 'TextBox' (Name='MaTextBox');
target property is 'Text' (type 'String')
```

Comme nous avons demandé tous les niveaux de message ("All"), la trace est même plus complète que dans la fenêtre de sortie de Visual Studio.

Seul inconvénient de cette solution c'est qu'elle ne fonctionne qu'avec WPF pour des tas de raisons évidentes (par exemple où serait écrit le fichier de trace sur la machine hôte alors que les droits pour une telle action n'existent pas pour une application Web ou WinRT).

Il existe des extensions au système de Trace Source dans WPF 3.5 notamment pour déboguer des situations encore plus délicates (cas d'un binding qui fonctionne, mais qui dépend d'une valeur qui est longue à obtenir – depuis un Web service, une base de données… – l'objet bindé pouvant rester alors vide). Je vous renvoie à un autre billet tout aussi intéressant que le précédent, « How can I debug WPF bindings ? » de Bea Stollnitz (voir les références en début d'article).

La feinte du convertisseur inutile

Je tiens cette idée de la même Bea Stollnitz, idée qui se résume à cela : le meilleur moyen pour inspecter un binding récalcitrant est de le forcer à revenir dans le code C# où il sera possible de placer un point d'arrêt pour inspecter les valeurs…

Bien entendu il n'est pas question ici de supprimer le binding en Xaml pour le remplacer par un équivalent en C#, cela n'aura aucun sens. Le binding Xaml reste à sa place dans ses balises. Et pour le forcer à faire un détour par C# il suffit d'adjoindre à la définition du binding l'utilisation d'un convertisseur. Ce dernier ne servira à rien, donc un code minimum, mais sera bien suffisant pour placer un point d'arrêt.

La feinte est intéressante, n'est-il pas ?

Voici le code du convertisseur de débogue :

```
public class DebuggingConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter,
                              System.Globalization.CultureInfo culture)
        {
            return value; // Ajouter le point d'arrêt ici !
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter,
                                  System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException("méthode non implémentée");
        }
    }
```

Code 20 - Faux convertisseur de valeur pour aider au débogue d'un binding

Pour ajouter la prise en charge du convertisseur dans un binding il faut connaître les méandres de la syntaxe du binding que nous verrons à la section suivante. Voici un exemple avec le `TextBox` du projet de démo fourni avec cet article :

Dans les ressources de la `Window` en cours :

```
<local:DebuggingConverter x:Key="debugConverter" />
```

Puis l'ajout du convertisseur dans le binding du `TextBox` :

```
<TextBox Text="{Binding Source={StaticResource Horloge}, Path=Heure,
Mode=OneWay, Converter={StaticResource debugConverter}}" Name="MaTextBox2"
/>
```

Ne reste plus qu'à placer le point d'arrêt sur la ligne indiquée dans le convertisseur et l'affaire est jouée, vous avez accès à tout ce qui est nécessaire pour déboguer le binding posant problème au moment précis où la valeur est modifiée.

Avantage : la technique est utilisable avec Silverlight sous débogueur Visual Studio.

Les points d'arrêt XAML

La possibilité de placer un point d'arrêt directement dans un fichier XAML a été ajoutée très tardivement ce qui explique le développement de nombreuses ruses qui permettent de s'en passer.

Il est désormais possible de placer un point d'arrêt sur du code XAML comme on le fait pour du code C# par exemple. Toutefois cette possibilité est limitée aux lignes XAML contenant un binding. En l'espèce cela nous arrange puisque c'est le sujet de cet article… mais on aurait aimé pouvoir en mettre à d'autres endroits.

Cette nouvelle possibilité est bien entendu à privilégier en premier recours pour inspecter un binding suspect. C'est le plus simple et le plus direct.

Malgré tout cela ne permet pas de savoir qu'un binding ne fonctionne pas correctement : on place le point d'arrêt sur un binding qu'on suspecte déjà de présenter un problème.

Les autres techniques présentées restent donc parfaitement d'actualité pour vérifier qu'aucun binding ne pose problème et découvrir le ou les bindings à vérifier.

Bugs sournois

Les bugs de Binding sont sournois par nature. Xaml n'étant pas compilé on voit d'ailleurs ici toute l'hérésie de vouloir développer des applications professionnelles énormes en Html/Js/CSS où rien n'est contrôlé ! Xaml n'échappe pas aux défauts des langages non fortement typés non compilés...

Les erreurs n'interviennent qu'au runtime et parfois elles sont mêmes difficiles à découvrir.

Quant à trouver la cause, c'est un jeu de piste hasardeux et long.

Il serait tellement simple de pouvoir mettre un point d'arrêt sur un binding et de vérifier si l'objet attaché, sa propriété, sont bien ceux qu'on attend... C'est désormais possible alors regardons par l'exemple comment faire !

Exemple

Prenons un projet minimaliste où l'erreur est tout à fait possible. Et voyons comment le debug de Binding peut résoudre les choses en quelques secondes.

Il s'agit d'une application exemple en Silverlight 5 "normale" sans fioriture ni framework supplémentaire.

J'ai créé une classe "Book" qui permet de stocker les informations principales d'un livre :

```
public class Book
{
    public string Title {get; set;}
    public int Year { get; set;}
    public string Author { get; set;}

    public override string ToString()
    {
        return Title + ", " + Author + " (" + Year + ")";
    }
}
```

Code 21 - Définition d'une classe de test simple

Rien de bien savant. J'ai surchargé le `ToString()` pour simplifier l'affichage dans une `ListBox` sans avoir besoin de créer un `DataTemplate`. Je cherche la simplicité absolue pour cet exemple.

Dans le constructeur du code-behind de `MainPage.xaml` je lie le code de la classe `MainPage` au `DataContext` global de la page (c'est une forme ultra édulcorée "à l'arrache" de MVVM !) :

```
public MainPage()
{
    InitializeComponent();
    DataContext = this;
}
```

Code 22 - Faux Mvvm avec DataContext sur this

Bien entendu pas de données de Design ici !

Dans ce même code, j'instancie une liste de livres "en dur" avec une propriété publique pour y accéder :

```
public List<Book> Books
{
    get
    {
        return books;
    }
}
private List<Book> books = new List<Book>
                                {
                                    new Book {Author = "Ray
Bradbury",Title = "Chroniques martiennes",Year = 1950},
                                    new Book {Author = "Jack
Vance",Title = "Palace of Love",Year = 1967},
                                    new Book {Author = "John T.
Sladek",Title = "Roderick",Year = 1980},
                                    new Book {Author = "Catherine
Lucille Moore",Title = "Tout smouales étaient les Borogoves",Year = 1943}
                                };
```

Code 23 - Initialisation d'une liste d'objets de test

Un peu de SF à lire si vous ne connaissez pas ces œuvres intéressantes...

Côté Xaml je place une `ListBox` dans le `LayoutRoot`, et je lie son `ItemsSource` à "`Books`" par Binding. Ce n'est pas ici que les choses vont aller mal alors passons.

Je place ensuite une `Grid` sous la `ListBox`. Elle contiendra les informations détaillées du livre sélectionné dans la liste.

Pour faire simple, je fais pointer le `DataContext` de cette grille sur le `SelectedItem` de la `ListBox`. Ce n'est pas là non plus que ça va dérailler.

A l'intérieur de cette grille, je pose des `TextBlock`, certains servent de labels, les autres vont être liés aux propriétés du livre sélectionné dans la liste.

C'est là qu'on peut facilement faire des bêtises. Autant jusqu'ici toute erreur de Binding serait très voyante - pas d'affichage dans la liste ou pas d'affichage du tout dans la grille de détail - autant dans les détails une erreur sauterait moins aux yeux. Et comme la structure de notre exemple n'a aucun support de design pour les données (vous en voyez l'importance...) je vais être obligé de taper les binding à la main (on sent le piège !).

Si je lance l'application et que je sélectionne une ligne dans la liste voici ce qu'il s'affiche :

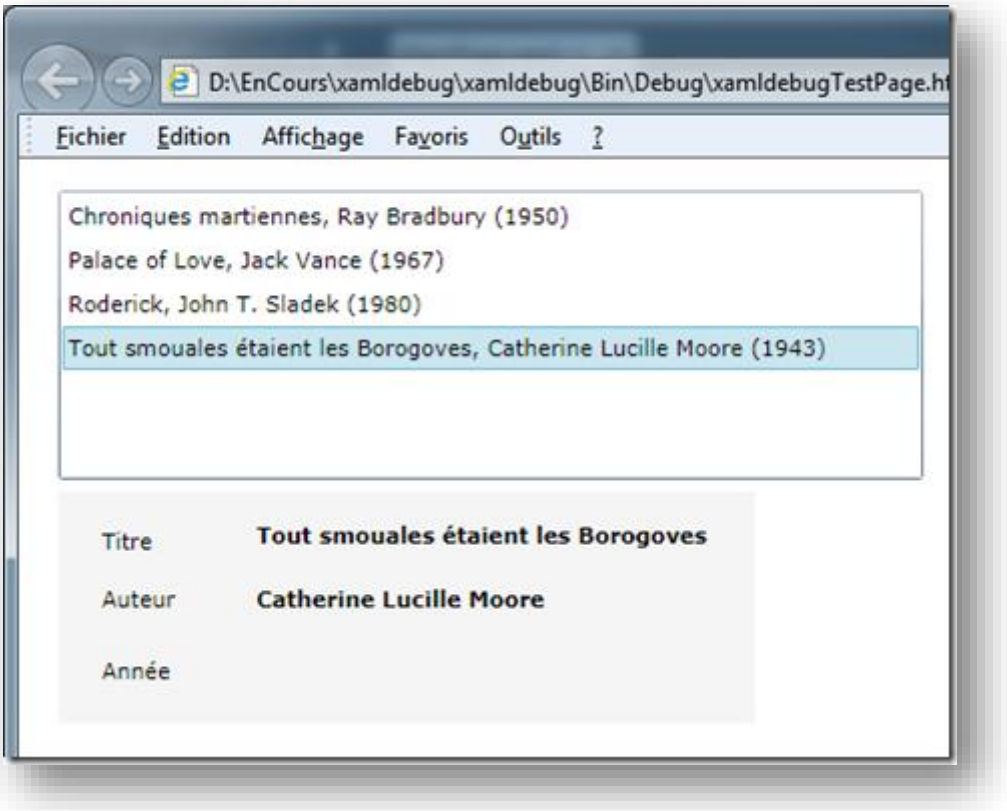


Figure 46 - Exécution de test avec bogue de binding

La grille de sélection en haut, le détail en-dessous. Malheureusement l'année ne s'affiche pas... Dans cet exemple l'erreur se voit très vite, sur un écran chargé cela peut être plus difficile à détecter d'où l'importante d'une phase d'intégration bien menée. Mais là n'est pas la question. Il y a un binding qui ne marche pas...

```
<UserControl x:Class="xamldebug.MainPage"
    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"
    xmlns:local="clr-namespace:xamldebug"
    mc:Ignorable="d"
    d:DesignHeight="372" d:DesignWidth="557"
             DataContext="{Binding RelativeSource={RelativeSource Self}}">

    <Grid x:Name="LayoutRoot" Background="White" >
        <ListBox Height="145" HorizontalAlignment="Left"
              Margin="20,16,0,0" Name="listBox1"
              VerticalAlignment="Top" Width="437"
              ItemsSource="{Binding Books}"/>
        <Grid Height="116" HorizontalAlignment="Left" Margin="20,167,0,0"
              Name="grid1" VerticalAlignment="Top" Width="352"
              DataContext="{Binding ElementName=listBox1,
                           Path=SelectedItem}" Background="WhiteSmoke">
            <TextBlock Height="23" HorizontalAlignment="Left"
              Margin="22,17,0,0" Name="textBlock1" Text="Titre"
              VerticalAlignment="Top" />
            <TextBlock Height="23" HorizontalAlignment="Left"
              Margin="22,46,0,0" Name="textBlock2" Text="Auteur"
              VerticalAlignment="Top" />
            <TextBlock Height="23" HorizontalAlignment="Left"
              Margin="22,82,0,0" Name="textBlock3" Text="Année"
              VerticalAlignment="Top" />
            <TextBlock Height="23" HorizontalAlignment="Left"
              Margin="100,14,0,0" Name="textBlock4" Text="{Binding Title}"
              VerticalAlignment="Top" FontWeight="Bold" />
            <TextBlock Height="23" HorizontalAlignment="Left"
              Margin="100,46,0,0" Name="textBlock5" Text="{Binding Author}"
              VerticalAlignment="Top"  FontWeight="Bold"/>
            <TextBlock Height="23" HorizontalAlignment="Left"
              Margin="100,82,0,0" Name="textBlock6" Text="{Binding Yer}"
              VerticalAlignment="Top" FontWeight="Bold" />
        </Grid>
    </Grid>
</UserControl>
```

Code 24 - Définition d'un binding erroné

Le code Xaml de la fiche est ultra simple et le `TextBlock` incriminé est le dernier objet déclaré. Un œil habitué à Xaml verra certainement tout de suite d'où vient le problème car ici tout est vraiment super simple...

Imaginons que cela soit plus proche de la réalité, plus complexe, plus difficile à décrypter et que vous soyez charrette, le truc à rendre pour avant-avant-hier, fin de journée dans un open space qui tue la concentration, 2 bonnes heures sup au compteur, tiens la femme de ménage vide les poubelles vous êtes tout seul, trois fois que votre nana essaye de vous joindre sur le portable et que vous refusez l'appel pensant terminer dans 5 minutes (depuis trois heures en fait), ça va encore être chaud ce soir en rentrant... Bref, une journée normale d'informaticien 

Dans ces conditions-là, la petite coquille en Xaml, elle devient beaucoup plus difficile à trouver je vous le garantie !

Mais heureusement le Binding peut maintenant être debuggé avec un point d'arrêt !

Plaçons un point d'arrêt sur la ligne du `TextBlock` récalcitrant, directement dans la page de code Xaml comme on le ferait en C# et lançons le programme...

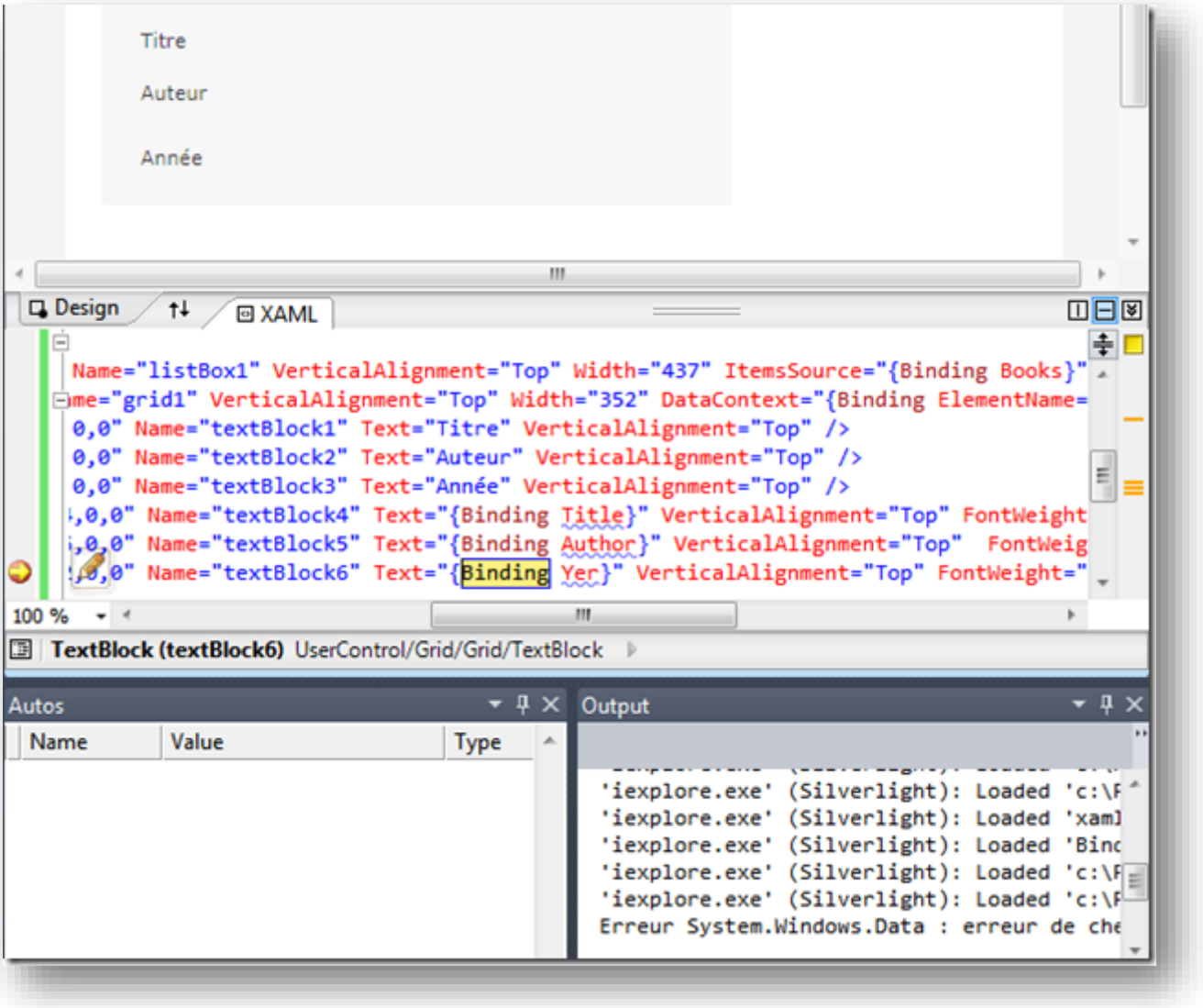


Figure 47 - Point d'arrêt XAML de débogue de Binding

Petit zoom sur la fenêtre “Output” :

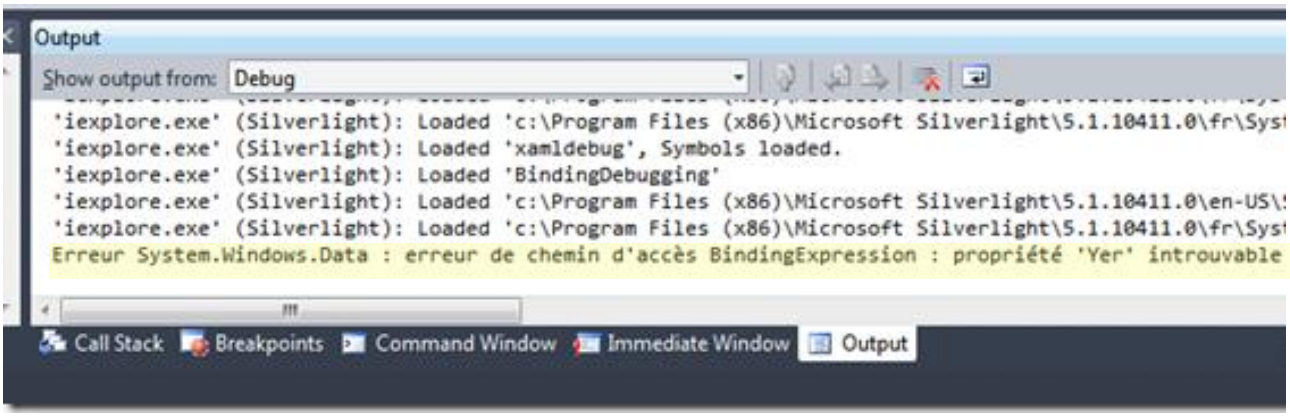


Figure 48 - Fenêtre de sortie Visual Studio avec erreur de binding

Il y a une “erreur de chemin d’accès” dans l’expression de Binding sur la propriété “`Yer`” qui est introuvable sur le livre “`Tout smouales étaient`…”

Là c’est bon… “`Yer`” ça n’existe pas. Même fatigué ça n’existe pas.

Il suffit d’ajouter le “a” qui manque dans le code de Binding pour se lier à la propriété “`Year`” et tout est ok !

Bien entendu les expressions de binding sont parfois bien plus complexes, avec des sources relatives, ou des Path longs, etc. Pouvoir connaitre immédiatement le détail du Binding qu’on soupçonne d’aller de travers est un vrai gain de temps. On peut bien entendu aller dans le code C# pour inspecter les valeurs, parfois l’erreur provient d’un calcul, d’une initialisation mal effectuée au moment du Binding. Sans ce point d’arrêt sur le Binding il est très difficile de savoir dans quel état se trouve le code… Nous pouvons utiliser tous les outils de debug pour inspecter les objets ce qui est précieux car parfois l’erreur ne vient pas de Xaml comme dans l’exemple donné mais d’un objet sous-jacent mal instancié, à nul, etc…

Conclusion

Même s’il n’y a rien de très spectaculaire dans le debug Xaml du Binding il n’en reste pas moins vrai que techniquement c’est une nouvelle possibilité bien pratique qui peut faire gagner beaucoup de temps.

Tout ce qui peut améliorer le debug est toujours le bienvenu !

Quelques outils supplémentaires

Vous retrouverez les liens vers les outils présentés ici dans la partie Références en début d’article.

Bien que tous ces outils de débogue ne soient pas exclusivement destinés au binding, ils vous seront certainement très utile si vous tombez sur un bug récalcitrant…

Spy++

Il s’agit d’un utilitaire de débogue de Microsoft qui permet d’obtenir une vue détaillée des processus, des threads, des fenêtres et des messages Windows. C’est un outil Win32, plutôt utilisé par les développeurs C++ mais qui peut s’avérer utile dans certains cas, notamment lors d’appel de code non managé.

ManagedSpy

Il s'agit d'un équivalent managé de Spy de chez Microsoft aussi. Etudié pour le débogue d'applications managées c'est un outil peu connu qui peut s'avérer très utile. De plus il est fourni avec le code source, autant de sa partie c++ que de sa partie managée en C#.

Pour des raisons étranges les pages consacrées à l'outil ont disparu des sites Microsoft. Il reste néanmoins un article de présentation sur MSDN ainsi qu'un accès direct au téléchargement de l'exécutable.

Snoop

Snoop est un équivalent de Spy++ et de ManagedSpy mais en plus convivial développé par Pete Blois. C'est une application WPF fournie en mode source, donc très instructif en plus d'être utile !

Mole

Après les espions (Spy & Co) voici la taupe ! Conçu par les mêmes personnes que les célèbres Power Toys, Mole est un débogueur qui fonctionne aussi bien avec WPF que les Windows Forms ou même ASP.NET. Mole n'est pas indépendant de Visual Studio, c'est un plugin qui agit comme un viewer personnalisé. L'utilitaire est fourni en binaire et en source.

Reflector

On ne présente plus cet utilitaire qui permet notamment de décompiler un exécutable ou une librairie, ce qui s'avère très utile notamment avec le Framework Silverlight pour vérifier certaines différences avec WPF et regarder l'implémentation de certaines méthodes. Bien que racheté par Red Gates (qui produit aussi le profiler Ants et toute une flopée d'outils orientés base SQL) il existe toujours une version gratuite qui désormais peut être appelée directement depuis l'EDI de Visual Studio.

Vos neurones

On les oublie parfois… en se jetant sur des outils de débogue, même ceux de Visual Studio, je vois souvent des développeurs perdre beaucoup de temps et au final ne pas trouver ce qu'ils cherchent.

N'oubliez jamais que le meilleur outil de débogue du marché vous le possédez déjà : ce sont vos neurones. Jamais aucun désassembleur ni espion de code n'arrivera à la cheville de ce qu'une poignée de neurones bien réveillés peuvent faire pour découvrir et éradiquer un bug !

Les syntaxes du Binding

Je ne ferai pas le tour des moyens de déclarer du Binding par code, les problématiques sont différentes, et seules les chaînes de caractères non contrôlées à la compilation doivent réclamer de votre part une grande vigilance. De plus il est assez rare de déclarer un binding en code behind.

En revanche les syntaxes du Binding lorsqu'il est déclaré dans le code Xaml méritent quelques éclaircissements. Un peu ésotérique et changeante selon les cas, donc difficile à mémoriser, la syntaxe du binding est une source d'erreurs qui pour beaucoup ne seront pas détectées (pas même d'erreur déclenchée à l'exécution, sauf en utilisant les traces expliquées plus haut).

De plus, tirer parti de toutes les subtilités de cette syntaxe réclame de la maîtriser, même un outil comme Blend ne prend pas toutes les facettes en charge dès lors qu'on sort des sentiers battus ou que l'on cherche l'optimisation.

Le binding simple

Le plus simple des binding est celui qui référence directement le `DataContext` courant.

Binding direct

```
Exemple : Content="{Binding}"
```

XAML 32 - Binding simple

Ici la propriété `Content` d'un `label` est directement liée au `DataContext` local.

Pour mettre en œuvre un binding aussi simple, nous plaçons un `Label` dans une `Grid`. La propriété `DataContext` de cette dernière référence une ressource statique appelée `Titre`.

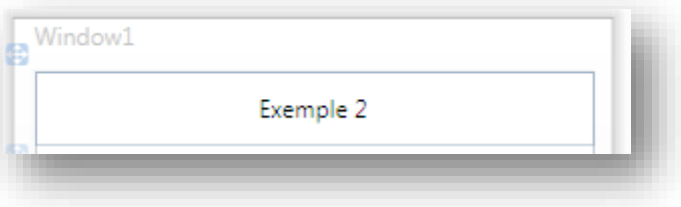


Figure 49 - Binding direct

La ressource est définie comme suit :

```
<Window.Resources>
        <System:String x:Key="Title">Exemple 2</System:String>
</Window.Resources>
```

XAML 33 - Création d'instances de tout type en ressources locales

C'est un moyen très simple de déclarer des chaînes, des constantes numériques qu'on regroupe dans les ressources de la fenêtre en cours ou dans un dictionnaire (`App.xaml` ou un dictionnaire externe).

La grille et son label sont définis comme suit :

```
<Grid Name="GridTitle" DataContext="{StaticResource Title}">
      <Label Margin="0" Name="label1" Content="{Binding}"/>
</Grid>
```

J'ai volontairement supprimé tout le code de présentation (marges, taille, position…) pour ne garder que le code Xaml fonctionnel. Vous trouverez le code complet de cet exemple dans le projet `SimpleBinding` fourni avec l'article.

Binding sur une propriété

```
Exemple : Content="{Binding Now}"
```

Ici la propriété `Content` d'un `Label` est liée au `DataContext` courant, sur sa propriété `Now`. Le `DataContext` est une instance de `DateTime`.

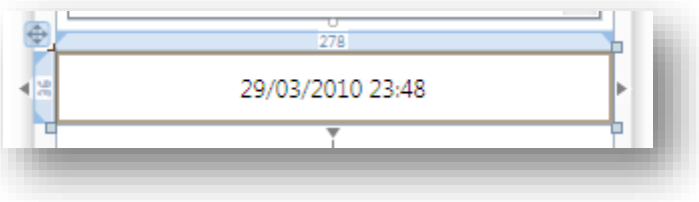


Figure 50 - binding sur une propriété

Pour réaliser cet exemple nous avons besoin d'une autre ressource locale de type `DateTime` :

```
<Window.Resources>
        <System:DateTime x:Key="Time" />
</Window.Resources>
```

Le `Label` est placé dans une grille dont le `DataContext` pointe la ressource ainsi définie. La propriété Content du Label est alors liée par binding à la propriété `Now` du contexte courant :

```
<Grid Name="GridHeure" DataContext="{StaticResource Time}">
   <Label Name="label2" Content="{Binding Now}" />
</Grid>
```

Comme pour l'exemple précédent j'ai supprimé le code de présentation des balises.

Vous noterez que cette démonstration rapide fonctionne très bien en design sous Visual Studio et que la date et l'heure s'affichent correctement. Mais si vous tentez de l'exécuter une erreur se produira car `DateTime` n'a pas de constructeur par défaut[6]. Pour faire fonctionner l'exemple au runtime il est nécessaire d'écrire notre propre classe `DateTime` possédant ainsi un constructeur par défaut. C'est ce que vous trouverez dans le projet exemple `SimpleBinding` qui contient la classe suivante utilisée en place et lieu de `DateTime` dans la balise de création de la ressource :

```
    public class MyDateTime
    {
        public DateTime Now { get { return DateTime.Now; } }
    }
```

Binding sur une sous-propriété du contexte

Dans la même logique il est possible d'atteindre une « propriété d'une propriété » du `DataContext`.

```
Exemple : Content="{Binding Now.Kind}"
```

En reprenant l'exemple ci-dessus il est possible d'accéder à l'une des propriétés de `Now` du `DateTime` (enfin de notre `MyDateTime`) créé en ressource. `Kind` affiche le type de date/heure pris en compte (ici le texte « `Local` » sera affiché).

[6] Je reste étonné que cela passe en design. La raison m'échappe. Si un GL (Gentil Lecteur) connait la raison qu'il me la communique, je la publierai sur Dot.Blog.

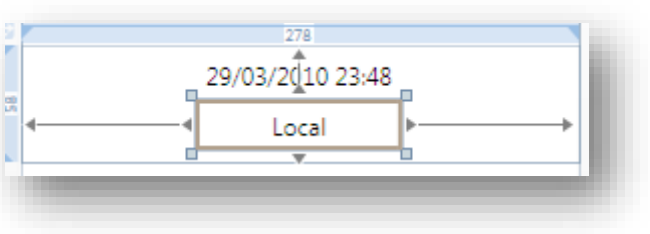


Figure 51 - Accès à une sous propriété du DataContext

L'Element Binding

```
Exemple : Content="{Binding ElementName=slider1, Path=Value}"
```

L'Element Binding consiste à pouvoir lier un élément d'interface à un autre directement. Ajouté dans Silverlight 3, WPF proposait déjà ce procédé très pratique.

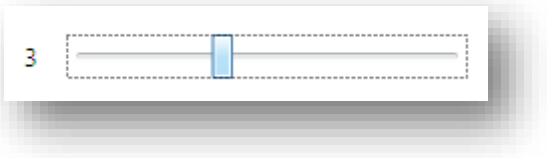


Figure 52 - Element Binding

L'exemple classique consiste à montrer comment un `Slider` est lié à un `Label` pour afficher la valeur courante. C'est ce que présente la figure ci-dessus.

L'affichage direct d'une valeur de type `Double` n'est généralement pas satisfaisant, les nombreuses décimales qui apparaissent ne sont que rarement désirées (en tout cas à l'affichage) et peuvent ruiner la mise en page.

Dans l'exemple fourni (et sur la figure ci-dessus) les valeurs affichées sont des entiers. Cela n'est hélas pas le comportement par défaut… Pour arriver à ce résultat il est nécessaire d'utiliser un convertisseur de valeur.

Convertisseurs de valeur

Le propos de cet article est le binding, si nous entrons dans les détails de tout ce qui peut tourner autour de cette technique ce long article sera très vite un livre à lui tout seul !

Toutefois les convertisseurs de valeur entrant dans la syntaxe du binding il est nécessaire d'en dire quelques mots, les plus curieux pourront chercher sur le Web les documents qui présentent en détail la technique et ses variantes de programmation.

Le Binding permet de relier tout et n'importe quoi ensemble. C'est presque magique jusqu'au moment où l'on comprend que tous ces types incompatibles entre eux sont

rendus « dociles » grâce aux convertisseurs implicites implémentés (de diverses façons) dans chaque classe. Toutefois ces convertisseurs ne sont parfois pas suffisants.

Dans l'exemple précédent il s'agit de transformer un `double` en `entier` pour obtenir un affichage plus sobre. S'agissant d'un binding nous n'avons pas de moyen de forcer le transtypage comme nous le ferions par code.

Dans d'autres circonstances il sera nécessaire d'aller plus loin dans l'adaptation des données utilisées dans un binding. On peut fort bien imaginer une classe possédant une propriété indiquant un niveau d'alerte sous la forme d'une énumération (par exemple '`Ok, Attention, Danger`') et vouloir que cette information soit représentée graphiquement par un contrôle dont la couleur traduit le niveau d'alerte (vert, jaune, rouge par exemple).

Les systèmes de conversion implicites du Framework ne peuvent plus rien ici, les types sont vraiment trop éloignés loin de l'autre (une énumération et une couleur ou une brosse) et la conversion réclame une interprétation (le code couleur ici) qui ne peut être automatisée. Un humain est obligé de choisir quelle couleur représentera quelle valeur de l'énumération.

Bref, dans tous ces cas il est nécessaire de convertir les données d'un binding pour les rendre compatibles entre elles. La conversion peut être à sens unique ou à double sens.

Pour ce faire il faut créer un convertisseur de valeur. Il s'agit d'une classe supportant l'interface `IValueConverter`.

Dans l'exemple du `Slider` le convertisseur est minimal :

```
[ValueConversion(typeof(double), typeof(int))]
public class DoubleToIntConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
                          object parameter, CultureInfo culture)
    {
        var d = (double)value;
        return (int) d;
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Code 25 - Convertisseur de valeur Double vers Int

Ce convertisseur ne marche que dans un seul sens (`ConvertBack` n'est pas implémenté) et il ne fait que transtyper un double en entier.

Pour l'utiliser nous devons le déclarer dans les ressources :

```
<Window.Resources>
    <local:DoubleToIntConverter x:Key="DoubleConverter"/>
</Window.Resources>
```

Ensuite, le binding du `Label` de l'exemple précédent sera complété de la façon suivante :

```
Content="{Binding ElementName=slider1, Path=Value,
                  Converter={StaticResource DoubleConverter}}"
```

Le paramètre `Converter` du binding est lié à la ressource statique que nous venons de créer. Ainsi, les valeurs en provenance du `Slider` seront passées par la moulinette du convertisseur avant d'arriver dans la propriété `Content` du `Label`.

Bien entendu les convertisseurs de valeur sont utilisables dans tous les types de binding, l'Element Binding n'étant qu'un exemple.

Paramètres de conversion

Comme vous l'avez certainement remarqué dans la déclaration du convertisseur ci-dessus, les deux méthodes possèdent un paramètre appelé « `parameter` » de type

`object`. Il s'agit d'un paramètre optionnel qui peut être passé au convertisseur pour en modifier le comportement.

Comme il s'agit d'un `object`, il est donc possible de passer des paramètres très complexes (puisqu'on peut ainsi passer une instance d'une classe comportant autant de propriétés que nécessaire pour le paramétrage visé).

Pour l'exemple nous reprendrons le projet de l'horloge (`WpfMiniBinding`). Comme nous l'avons vu l'heure affichée par l'horloge est le résultat d'un appel à `ToString()`, c'est-à-dire un formatage standard possédant notamment de nombreux chiffres inutiles (en général) pour les millisecondes.

Comment faire si nous souhaitons afficher uniquement l'heure, sans les millisecondes ? Et comment ne pas réécrire des tas de convertisseurs pour les fois où nous souhaiterons afficher uniquement le jour, ou l'année ou tout autre formatage d'une date ?

Le plus simple est de créer un convertisseur de valeur acceptant un paramètre qui sera ici la chaine de format.

Un tel code est très simple :

```
public class DateConverter : IValueConverter
{
   public object Convert(object value, System.Type targetType,
          object parameter, System.Globalization.CultureInfo culture)
        {
            try
            {
                var format = (string) parameter;
                if (string.IsNullOrEmpty(format)) return value.ToString();
                var d = DateTime.Parse(value.ToString());
                return d.ToString(format);
            }
            catch (Exception ex)
            {
                throw new Exception("Paramètre de formatage invalide."
                                    +ex.Message);
            }
        }

        public object ConvertBack(object value, System.Type targetType,
                                  object parameter,
                                 System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
```

Code 26 - Convertisseur de valeur avec paramètre

Il suffit ainsi d'exploiter le paramètre « `parameter` » de la méthode `Convert` (ou `ConvertBack` si elle était implémentée).

Le convertisseur est instancié dans les ressources de la fenêtre en cours (il pourrait être placé ailleurs, dans `App.xaml` par exemple) :

```
    <Window.Resources>
        <local:DateConverter x:Key="DateConverter" />
    </Window.Resources>
```

La définition du binding de la `TextBox` devient ainsi :

```
<TextBox
  Text="{Binding Source={StaticResource Horloge}, Path=Heure, Mode=OneWay,
         Converter={StaticResource DateConverter},
         ConverterParameter='HH:mm:ss'}"
Name="MaTextBox2"
/>
```

XAML 34 - Utilisation d'un convertisseur de valeur avec paramètre

La balise de binding se complexifie singulièrement, mais une fois le convertisseur écrit, il est vrai qu'il n'y a plus besoin de code behind pour afficher des dates formatées selon nos souhaits partout où des dates apparaissent dans un binding.

Au final il s'agit d'une simplification…

De toute façon deux écoles semblent être nées et se renforcer dans leur spécificité de jour en jour : celle des développeurs qui font le maximum en Xaml « sans code », et ceux qui pensent que Xaml doit rester un langage de définition d'interface et que toute logique doit être programmée en code. Il y a toujours certains indécis qui oscillent entre les mondes.

Au premier je dirais que « sans code » cela n'est pas vrai, ce n'est qu'un transfert de code, de C# vers Xaml, d'un langage fortement typé et contrôlé vers un langage moins fiable de ce point de vue. Au second je dirais que Xaml est quand même un peu plus qu'un langage pour dessiner des petits mickeys. Quant aux troisièmes, s'ils s'hésitent c'est qu'ils soupèsent les dangers de chacune de ces approches radicales. La vérité étant souvent ailleurs, et surtout au milieu, trouver la juste balance entre ces deux extrêmes est vraisemblablement le meilleur choix…

StringFormat

Puisque nous en sommes aux détours et au formatage des valeurs d'un binding, profitons-en pour aborder `StringFormat`, un autre paramètre du binding.

Ajouté il y a peu au Framework « standard » et seulement à partir de la version 4 sous Silverlight ce paramètre permet de contrôler le formatage de la zone affichée et évite dans certains cas l'obligation de créer un convertisseur de valeurs.

`StringFormat` reprend les options de la méthode `String.Format` (voir dans les références en début d'article *String Format Options* pour un billet regroupant les différents formats).

Par exemple, reprenons l'affichage de la ressource `DateTime` de l'un des exemples plus haut. Par défaut nous avons un affichage de type date + heure classique. Comment faire si nous ne voulons afficher qu'une date courte, sans heure, composée du nom du jour, du mois et de l'année sur deux chiffres ?

Pour résoudre ce problème il faudrait écrire un convertisseur de valeur prenant un `DateTime` et acceptant un paramètre de format nous permettant d'obtenir le résultat (ce que vous venons de voir juste plus haut). On peut aussi écrire un convertisseur spécifique retournant une chaîne formatée comme nous le désirons. En fait le choix est soit d'écrire un convertisseur assez générique utilisant un paramètre (ce qui compliquera encore plus la syntaxe dans la balise de binding) soit … d'écrire un convertisseur de valeur spécifique (au risque, en bout de projet, d'avoir écrit des dizaines de convertisseurs spécialisés, ce qu'on voit assez souvent en WPF il faut le dire).

Le mieux est de profiter de `StringFormat` :

```
<TextBlock Text="{Binding Now, StringFormat='dddd-MM-yy'}" />
```

On voit l'utilisation d'un `TextBlock` et non d'un `Label` car ce dernier a une propriété `Content` qui accepte un peu tout, et le `StringFormat` n'a aucun effet (si on passait une image comme `Content` on se demande en effet comment le formatage s'y prendrait). `TextBlock` possède une propriété `Text` et le formatage s'applique donc correctement.

Mais… mais… regardons la fiche un instant…

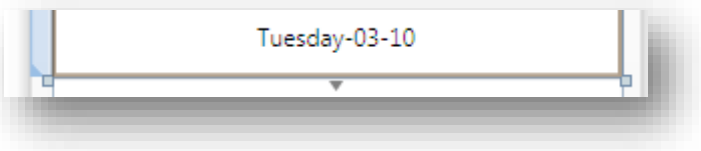


Figure 53 - StringFormat en US

Et oui… il y a un bug ! Le `StringFormat` se moque totalement de la culture et fonctionne en US. Damned !

Il y a une astuce pour corriger les choses, il « suffit » de surcharger la méthode `OnStartup` de `App.xaml` avec un bout de code magique (magique car si vous le devinez tout seul c'est que, sans mentir, si votre ramage se rapporte à votre plumage, vous êtes le phénix des hôtes de ces bois !) :

```
        protected override void OnStartup(StartupEventArgs e)
        {
            FrameworkElement.LanguageProperty.OverrideMetadata(
                        typeof(FrameworkElement),
                        new FrameworkPropertyMetadata(
                                    XmlLanguage.GetLanguage(
                          CultureInfo.CurrentCulture.IetfLanguageTag)));
            base.OnStartup(e);
        }
```

Code 27 - Correction générique de l'erreur de localisation du framework .NET

Au design vous verrez toujours la date en anglais, mais à l'exécution vous aurez le plaisir de lire la date dans la bonne culture. L'astuce corrige tous les aspects du formatage, comme par exemple l'utilisation de la virgule au lieu du point pour les décimaux en français.

Injection de culture

On peut aller encore plus loin avec `StringFormat`, et pour reprendre une façon de parler très à la mode, faire de « l'injection de culture » terme technique totalement inventé pour l'occasion je vous rassure, mais après tout, tout le monde jargonne sans trop savoir qui le premier a inventé le terme, ici au moins vous saurez que c'est moi !

Il peut arriver qu'au sein d'une application dont la culture est déjà prise en compte (notamment par l'astuce précédente) on rencontre le besoin de formater une valeur selon une culture spécifique. Une application de traduction pourra ainsi vouloir présenter un texte ou des données dans plusieurs langues à la fois. L'imagination ne vous manquant pas, je vais me concentrer sur le « comment » :

Supposons des `TextBlocks` définis dans un `StackPanel`. Le contexte de ce dernier sera le suivant :

```
<StackPanel.DataContext>
        <sys:Int32>123</sys:Int32>
</StackPanel.DataContext>
```

On retrouve l'astuce consistant à créer une valeur constante comme `DataContext`, ici un entier de valeur 123.

Le premier `TextBlock` sera défini pour afficher le nombre en culture US :

```
<TextBlock
    Text="{Binding ConverterCulture='en-US', StringFormat='{}{0:N2}' }"
/>
```

Ce qui affichera `123.00`, avec un point pour la séparation décimale puisque nous avons indiqué un formatage avec 2 décimales et un format US.

Le second `TextBlock` sera identique à la définition de la culture près, en FR :

```
<TextBlock
    Text="{Binding ConverterCulture='fr-FR', StringFormat='{}{0:N2}' }"
/>
```

XAML 35 - Fixer la culture dans un binding

Ce qui affichera `123,00` puisque notre belle langue préfère la virgule au point en cet endroit.

Certains curieux auront noté les étranges accolades vides dans la chaîne de format… Oui c'est curieux, étrange et même bizarre. Encore une source de confusion et d'erreurs difficilement cernables dans un gros logiciel. N'insistez pas je ne vous en donnerai pas l'explication ! Niet ! Bon… c'est bien parce que j'ai bon cœur… En fait, les chaînes de format utilisées sont celles de `String.Format`, et elles utilisent déjà pour certaines des accolades américaines (*curly braces*). Si on les utilisait directement il y aurait conflit avec les mêmes symboles qui ont un sens particulier dans le binding Xaml. Le premier jeu d'accolades vide permet en quelque sorte d'annuler la signification des symboles, une sorte de séquence escape permettant d'utiliser derrière la chaîne de format qui comporte des accolades de même nature. Voilà, vous savez. Et mon article explose un peu plus en taille. Vous en assumez la responsabilité, c'est bien et je vous en remercie ☺

Le ContentStringFormat

Oui, il y a encore une option de formatage… Utile comme les autres bien sûr, mais c'est un peu comme trop de crème chantilly sur un gros gâteau, au début c'est appétissant, mais si on en met vraiment de trop ça devient écœurant… Attention à l'overdose, début de la perte de maitrise, fille de tous les vices et surtout du code spaghetti !

Cette option de formatage ne fonctionne que sur les contrôles offrant un `Content`, donc `ContentControl`, `ContentPresenter`, le `Label` et le `TabControl` (j'en oublie peut-être). Elle a l'avantage de remplir un vide et de s'appliquer hors du binding.

Comme nous avons vu que le `StringFormat` fonctionne uniquement sur des propriétés de type texte, il fallait bien une astuce pour les conteneurs.

Prenons un bouton qui va déclarer un contenu numérique que nous souhaitons voir formater comme un monétaire :

```
<Grid Name="GridContentFormat" >
            <Button ContentStringFormat="{}{0:C}" >
                <System:Int32>546</System:Int32>
            </Button>
</Grid>
```

XAML 36 - ContentStringFormat

Ici la valeur à afficher est un entier valant `546`.

Au design vous verrez la chose suivante :

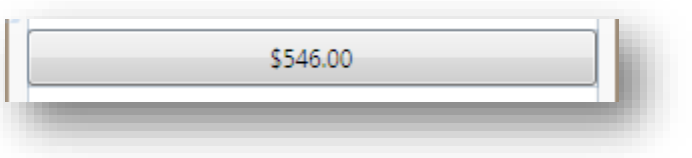


Figure 54 - ContentStringFormat

Les dollars ne sont pas encore utilisés en Europe, mais c'est toujours le même bug qui s'illustre. Grâce à la solution présentée plus haut et intégrée à `App.xaml`, à l'exécution nous retrouverons bien heureusement notre chère (très chère) monnaie :

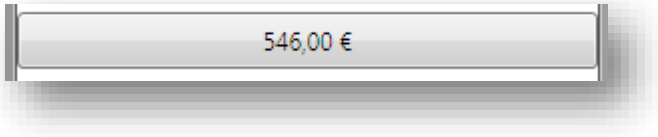


Figure 55 - ContentStringFormat corrigé

Gérer les nuls

Il ne s'agit pas ici de trouver une occupation à ceux qui n'auraient pas réussi à suivre cet article, non, juste de se demander ce qu'il se passe dans le cas d'un binding avec formatage sur une source qui peut retourner une valeur nulle…

Si, je vous l'assure, il se passe quelque chose. Ça plante.

Pour faire court voici la syntaxe de l'option `TargetNullValue` qui s'ajoute bien évidemment à tout ce que la chaîne de caractères définissant le binding contient déjà (!) :

```
<TextBox
    Grid.Column="1"
    Grid.Row="1"
    Margin="0,11,11,11"
    VerticalAlignment="Top"
    Text="{Binding Path=NumberOfComputers,
            TargetNullValue={x:Static sys:String.Empty},
            StringFormat=\{0:D\}}" />
```

XAML 37 - TargetNullValue pour gérer les NULL dans un binding avec format

Dans cet exemple le `TargetNullValue` est « sous bindé » à une ressource statique qui est `String.Empty`. Dans le cas où la source de binding est nulle, c'est donc cette valeur qui sera reprise et non le formatage par `StringFormat`. On pourrait bien entendu définir toute autre valeur, voire une constante chaîne.

Autant dire qu'il s'agit là d'une simplification qui … complexifie aussi les balises de binding et qui renforce bien entendu les craintes que j'exprime en introduction et qui explique aussi pourquoi personne ne connait réellement toutes les options du binding… Mais cela peut rendre service si la stratégie d'utilisation est clairement définie et documentée.

Le Binding Multiple

Encore une option récente qui, si elle renforce le potentiel du binding, rajoute sa petite dose de complexité. Une dose par ci, une dose par là, l'édifice devient difficilement maîtrisable et les dérapages du code spaghetti ne se cachent pas loin, alors restez vigilant !

Le `MultiBinding` est une réponse à une problématique simple : imaginons que vous deviez afficher un texte formaté constitué de plusieurs valeurs différentes et que vous souhaitiez, bien naturellement, que l'affichage se mette à jour si l'un ou l'autre des fragments change.

Une autre problématique à laquelle le `MultiBinding` répond aussi, et c'est peut-être là son intérêt le plus grand, c'est la capacité à agréger plusieurs valeurs pour en créer une seule à la sortie. Imaginez simplement trois sliders permettant de régler le niveau

du rouge, du vert et du bleu d'un rectangle, donc de la couleur de sa propriété `Fill`. Arriver à « mixer » les trois valeurs indépendantes pour créer une couleur ou directement une brosse qui pourra être liée par binding à la propriété `Fill` du rectangle peut, si cela est bien utilisé, aller vers une simplification du code (puisque qu'on supprime tous les gestionnaires d'événements de changement de valeur des sliders).

Toutefois le `MultiBinding` ne vous évitera pas un passage par C# car il faut un convertisseur spécial pour gérer la situation (ce qui se comprend assez facilement).

Vous trouverez dans les références en début d'article l'adresse d'un tutor portant sur le `MultiBinding`, je vous conseille d'y faire un tour pour compléter mes explications.

Ici je vais me contenter de lier la propriété `Text` d'un `TextBlock` aux propriétés `Nom` et `Prenom` d'une instance de la classe `Personne` créée pour l'occasion et de formater le tout comme suit : '`Prenom, Nom`'.

La classe Personne

Je vous la présente brièvement car je m'en resservirai plus loin :

```
public class Personne : INotifyPropertyChanged
    {
        private string nom = string.Empty;
        public string Nom
        {
            get { return nom; }
            set
            {
                nom = value;
                doChanged("Nom");
            }
        }

        private string prenom = string.Empty;
        public string Prenom
        {
            get { return prenom; }
            set
            {
                prenom = value;
                doChanged("Prenom");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void doChanged(string propertyName)
        {
            var p = PropertyChanged;
            if (p == null) return;
            p(this, new PropertyChangedEventArgs(propertyName));
        }
    }
```

Code 28 - Définition d'une classe de test exposant 2 propriétés

Rien de bien particulier, une classe avec deux propriétés et supportant `INotifyPropertyChanged`. Vous noterez la façon dont `doChanged` est programmé en créant une variable locale depuis `PropertyChanged`. Cette stratégie est conseillée pour éviter certains problèmes en multithreading. Une bonne habitude à prendre mais que, hélas, je ne développerai pas dans cet article (déjà bien chargé malgré tout).

Le code du multi convertisseur

Comme cet exemple se veut court et que l'article porte sur Xaml et le binding plus que sur C#, j'ai conçu un convertisseur multiple assez minimaliste, le voici :

```
class PersonneMultiConverter : IMultiValueConverter
{
    public object Convert(object[] values, Type targetType,
                          object parameter, CultureInfo culture)
    {
        var nom = (string) values[0];
        var prenom = (string) values[1];
        return prenom + ", " + nom;
    }

    public object[] ConvertBack(object value, Type[] targetTypes,
                                object parameter, CultureInfo culture)
    {
        return new object[] {};
    }
}
```

Code 29 - Définition d'un convertisseur de valeurs multiples

Les multi convertisseurs se différencient des convertisseurs simples par l'interface qu'ils supportent : `IMultiValueConverter`. Le principe reste le même, une méthode `Convert` et une autre `ConvertBack` pour convertir les données dans l'autre sens.

Le code ci-dessus n'implémente qu'un seul sens et formate les données comme annoncé plus haut.

Le code Xaml

La ressource est créée de la façon suivante :

```
<Window.Resources>
    <local:Personne x:Key="Personne" Nom="Dahan" Prenom="Olivier" />
</Window.Resources>
```

Le `TextBlock` est programmé comme suit (les balises ont été fortement « éclatées » pour rendre la décomposition plus évidente) :

```
<Grid Name="GridMultiBinding" DataContext="{StaticResource Personne}">
   <TextBlock>
     <TextBlock.Text>
       <MultiBinding Converter="{StaticResource PersonneMultiConverter}" >
         <MultiBinding.Bindings>
            <Binding Path="Nom"/>
            <Binding Path="Prenom"/>
         </MultiBinding.Bindings>
       </MultiBinding>
     </TextBlock.Text>
   </TextBlock>
</Grid>
```

XAML 38 - Utilisation d'un convertisseur à multiples valeurs

On peut illustrer le `MultiBiding` de façon malgré tout plus simple en évitant le convertisseur.

Toujours au sein de la même grille définissant le `DataContext` sur l'instance de la classe Personne définie en ressource, nous pouvons écrire :

```
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="Nom: {1} {0}">
                        <Binding Path="Prenom" />
                        <Binding Path="Nom" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
```

XAML 39 - Binding à multiples valeurs sans convertisseur de valeur

Ici nous n'avons pas besoin de convertisseur, c'est bien plus pratique (mais moins riche en possibilités). L'ordre des champs traités est indiqué juste au-dessus de leur apparition là où le convertisseur précédent était obligé d'assumer un ordre fixe (source de bug).

Les possibilités ne sont pas exactement les mêmes mais cette forme de `MultiBinding` est peut-être la plus intéressante pour une utilisation au quotidien dans le sens où elle simplifie le code sans ajouter trop d'éléments à contrôler « manuellement » (comme l'ordre des paramètres dans le convertisseur).

Le Binding XML

Le binding XML est peut-être celui qui est le plus bluffant. Les sources XML ne sont jamais très simples à traiter. Le Binding Xaml montre ici toute sa puissance en offrant une syntaxe claire et concise. Mais c'est une syntaxe de binding de plus à connaître et à maîtriser !

Binding sur une source Web

```
DataContext="{StaticResource DotBlog}" ItemsSource="{Binding
XPath=item/title}"
```

Dans cet exemple syntaxique un `DataContext` est lié à une source XML déclarée par le biais d'une ressource `XmlDataProvider`. Le chemin à afficher est donné par l'expression `XPath`, ici « `item/title` ».

Prenons l'exemple de bout en bout pour mieux comprendre.

Le Framework offre la classe `XmlDataProvider` qui se charge de créer un lien avec une source au format XML. D'autres moyens d'accéder à une source XML existent, Linq To XML par exemple. Cela s'écartant de notre sujet regardons juste comment la source est définie côté Xaml (projet exemple `XmlBinding`) :

```
<Window.Resources>
        <XmlDataProvider x:Key="DotBlog" XPath="rss/channel"
             Source="http://www.e-naxos.com/blog/syndication.axd" />
</Window.Resources>
```

XAML 40 - XmlDataProvider

Le plus simple pour trouver une source XML est d'aller la chercher sur le Web… Ici le `XmlDataProvider` pointe tout simplement le flux RSS de mon blog (et le chemin `rss/channel` à l'intérieur de la source).

On peut dès lors créer un binding avec une `Listbox` pour récupérer tous les titres du blog :

```
<ListBox
        DataContext="{StaticResource DotBlog}"
        ItemsSource="{Binding XPath=item/title}"
/>
```

XAML 41 - Listbox liée à une source XML par XPath

Et c'est tout. Déjà au design la `ListBox` se remplit des données. Exécutons et voyons le résultat :

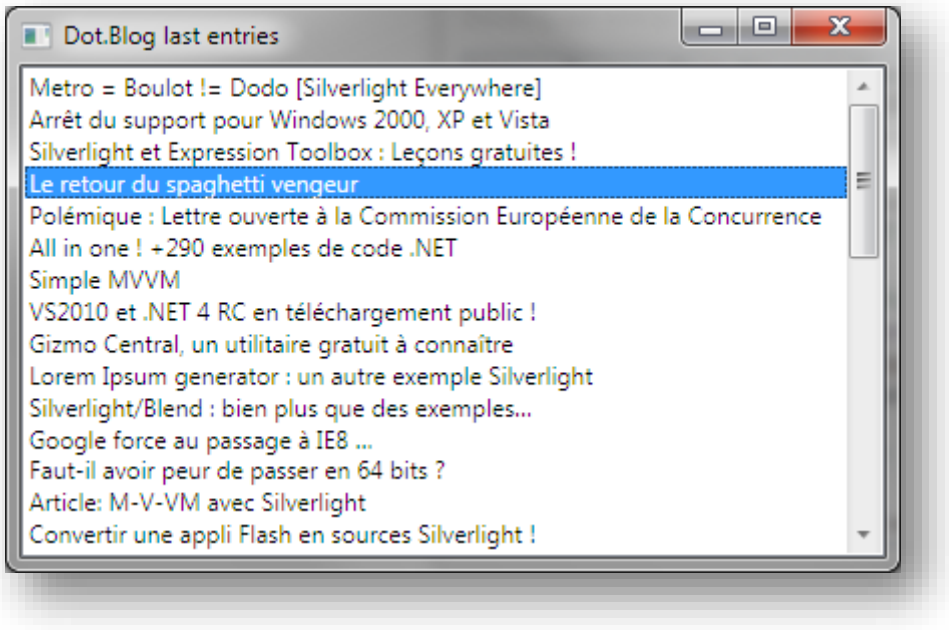


Figure 56 - Binding XML

Il y a quelque chose d'un peu magique dans ces deux ou trois lignes de Xaml qui créent une application capable de lister un flux RSS sans aucun code behind. Imaginons juste quelques instants la quantité de code et de librairies externes que cela aurait nécessité dans les environnements d'avant Xaml, d'avant .NET. C'est le côté clair de la force du binding. Ne jamais sous-estimer pour autant la puissance du côté obscur !

Binding sur une source XML en ressource

Les possibilités du binding sont telles qu'il existe plusieurs façons d'obtenir le même résultat. Le mieux est de voir un autre exemple d'utilisation de source XML.

Ici nous créons un répertoire `data` dans le projet et nous y ajoutons un fichier XML. Cette source contient une liste de livres. Le fichier XML est placé en ressource de l'application.

La définition de la source est très proche de celle de l'exemple précédent :

```
<XmlDataProvider x:Key="Books" Source="data/8534-4594.xml"
                                          XPath="Books/Titles" />
```

La clé définit le nom de la ressource, la `Source` pointe le fichier XML, et le `XPath` indique le chemin qui nous intéresse (la balise `Books` englobe tout le fichier, chaque livre est une entrée `Titles`).

Ce qui nous intéresse c'est le binding, alors regardons maintenant comment une `Listbox` est liée à cette source :

```
<ListBox ItemsSource="{Binding Source={StaticResource Books}}"
                                       DisplayMemberPath="title" />
```

Vous noterez que cette fois-ci je n'ai pas utilisé le `DataContext`. Puisque seule la liste est liée à la source on peut définir le binding directement sans passer par cet intermédiaire. De fait la source du binding est liée à la ressource statique `Books` créée plus haut. En fixant le `DisplayMemberPath` on choisit ici de n'afficher que le contenu de la balise `title` de chaque livre.

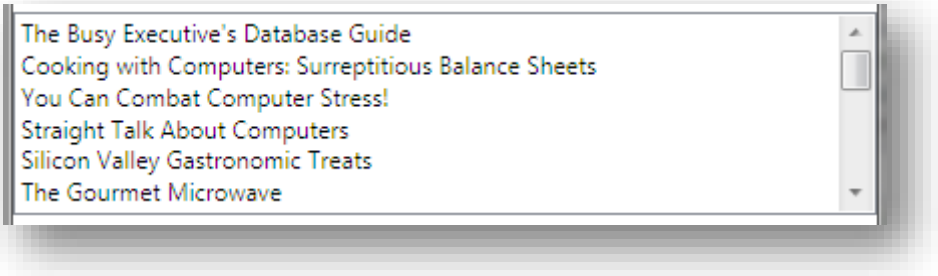


Figure 57 - Binding XML depuis un fichier en ressource

Binding sur une requête Linq To XML

Il n'est évidemment pas question de traiter de toutes les manières d'accéder à une source XML et encore moins d'entrer dans les détails de Linq To XML. Mais voici un cas pratique qu'on pourrait croire simplement dérivé des exemples précédents qui sont fort simples, ce qui n'est pas le cas.

Les difficultés rencontrées ne concernent pas le binding lui-même mais l'accès au fichier XML en ressource et son traitement en Linq To XML. Le binding repose ensuite sur un `ObjectDataProvider` plutôt qu'un `XmlDataProvider` auquel il faut accéder par code pour en initialiser la source. Plus quelques autres détails. D'où l'intérêt de cet exemple : la multiplicité des petites difficultés.

La source XML sera la même que celle des exemples précédents, un fichier placé en ressource de l'application (projet exemple `xmlBinding`).

La partie Xaml est simple, on commence par créer une ressource `ObjectProvider` :

```
    <Window.Resources>
        <ObjectDataProvider x:Key="Linq" />
    </Window.Resources>
```

XAML 42 - ObjectDataProvider

Il n'y a pas de source déclarée ici car nous allons traiter le fichier XML avant de le transformer en source utilisable. Dans la liste des livres nous souhaitons extraire tous les livres ayant un prix inférieur à 20 euros et les classer par ordre décroissant de prix.

Cela n'empêche en rien de définir la `ListBox` et son binding :

```
<ListBox
    ItemsSource="{Binding Source={StaticResource Linq}}"
    DisplayMemberPath="FullDescription"
/>
```

Nous avons prévu de retourner un type anonyme dont l'une des propriétés s'appellera `FullDescription` (une concaténation du prix et du titre).

Côté binding les jeux sont faits. Reste à coder la logique. Sans trop entrer dans les détails voici le code qui réalise tout cela :

```
StreamResourceInfo sr =
    Application.GetResourceStream(
          new Uri("xmlBinding;component/data/8534-4594.XML",
                  UriKind.Relative));
var xr = new StreamReader(sr.Stream);
XDocument xd = XDocument.Load(xr);
var nfi = new NumberFormatInfo();
nfi.NumberDecimalSeparator = ".";
var q = from x in xd.Descendants("Titles")
        where x.Element("price") != null
        let price = double.Parse(x.Element("price").Value, nfi)
        where price < 20d
              orderby price descending
        select new
                  {
                      Title = x.Element("title").Value,
                      Price = price,
                      FullDescription =
                         price.ToString("##0.00") + " " +
                                  x.Element("title").Value
                  };
var p = Resources["Linq"] as ObjectDataProvider;
if (p!=null) p.ObjectInstance = q;
```

Code 30 - Source de données via LINQ to XML

En partant du début (c'est plus simple !), le code ci-dessus réalise les choses suivantes :

- Obtention d'un objet `StreamResourceInfo` sur le fichier XML placé en ressource de l'application.
- Déclaration d'un `StreamReader` sur le flux de l'objet précédent pour le lire.
- Création d'un `XDocument` par chargement du flux.
- Comme les prix sont stockés au format américain (avec un point pour les décimales) le code créé un `NumberFormatInfo` qui est modifié pour indiquer cette particularité.
- Vient ensuite la requête Linq To XML qui sélectionne tous les livres coûtant moins de 20 euros et qui les classe par ordre décroissant de prix. La requête retourne un type anonyme contenant le prix, le titre, et le `FullDescription` qui n'est que la concaténation des deux autres champs.

- Reste à récupérer l'instance de `l'ObjectDataProvider` qui a été défini en Xaml dans les ressources de la fenêtre, puis à assigner le résultat de la requête à sa propriété `Source`.

Exécutons :

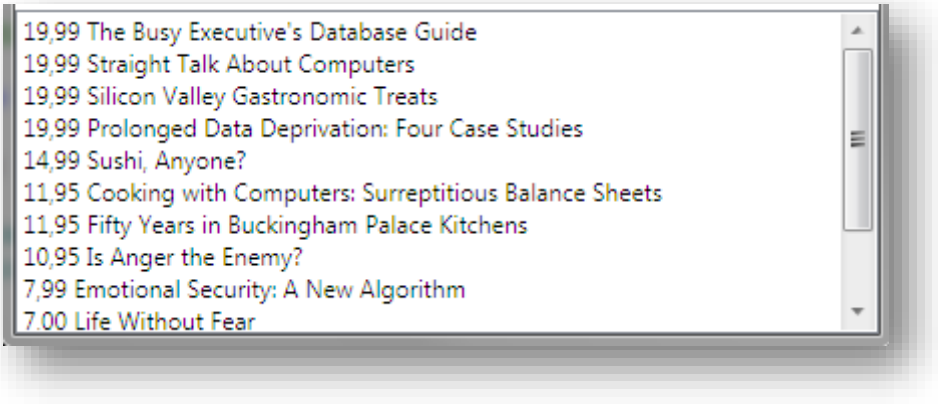


Figure 58 - Binding sur le résultat d'une requête Linq To XML

Le Binding Relatif

Tout est relatif dans la vie… même le Binding !

En effet, nous avons vu jusqu'ici différentes formes de binding mais toutes avaient en commun de référencer explicitement une source « fixe » : `DataContext`, propriété, expression `XPath`.

Or, il se trouve des cas où le binding doit pouvoir être exprimé de façon plus nuancée. C'est le binding dit relatif, avec sa syntaxe bien à lui, comme on peut s'y attendre.

Il existe plusieurs formes de binding relatif :

- `Self`
- `TemplatedParent`
- `FindAncestor`
- `PreviousData`

Binding to Self

Dans un tel titre autant écrire tout en anglais. (Amusant non ?)

Pour rester dans l'amusant (enfin, pas hilarant non plus, n'exagérons rien), voici un exemple dont je vous demande d'imaginer le résultat avant de lire l'explication :

```
<TextBlock
    Text="{Binding RelativeSource={RelativeSource Self}, Path=Background}"
    Background="Chartreuse"
/>
```

Ça donne quoi à votre avis ?

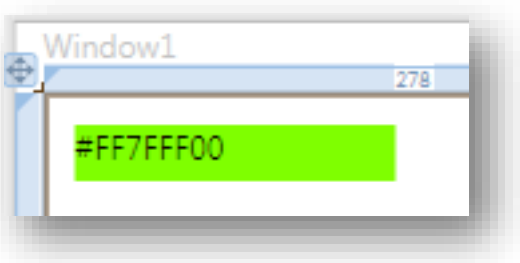


Figure 59 - Binding to Self

Oui, le `TextBlock` se regarde le nombril. En liant sa propriété `Text` à une source relative bien spéciale (`Self`) et à la propriété `Background` de celle-ci, on force le `TextBlock` à écrire le code de sa propre couleur !

Ce qui est plus intéressant est bien entendu la syntaxe de cet amusant regard sur soi-même de l'objet. Car si les objets n'ont pas d'âme, ils pratiquent grâce au binding relatif le « Connais-toi toi-même » de Socrate…

Le TemplatedParent Binding

L'intérêt de ce binding particulier est peut-être plus grand encore que le précédent, il s'agit ici de pouvoir effectuer un binding sur une propriété du template de contrôle dans lequel l'objet créant le binding se trouve.

Prenons rapidement un exemple pour éviter le flou.

Imaginons un bouton qui possède un template. A l'intérieur de celui-ci créons divers éléments imbriqués dont un `TextBlock` qui, de façon similaire à l'exemple précédent, va se lier par binding à la propriété `Name` du `TemplatedParent`, soit du parent du template, donc le bouton.

Le code Xaml est le suivant :

```
<Button Name="LeBoutonParent" Content="LeBouton">
    <Button.Template>
       <ControlTemplate x:Name="TemplateDuBouton">
          <StackPanel Name="StackPanelInterne">
             <TextBlock Background="Aquamarine"
              Text="{Binding RelativeSource={RelativeSource
                                  TemplatedParent}, Path=Name}"/>
          </StackPanel>
       </ControlTemplate>
    </Button.Template>
</Button>
```

XAML 43 - Templating et Binding Relatif

Le résultat visuel est alors le suivant :

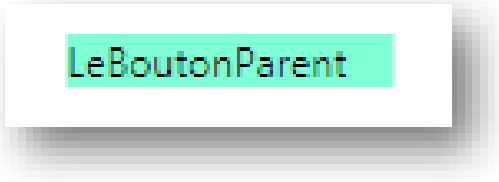


Figure 60 – TemplatedParent Binding

Visuellement notre bouton est méconnaissable et n'est pas vraiment « looké » c'est le moins qu'on puisse dire, mais ce n'est pas ce qui nous intéresse ici. Nous voyons un espace occupé par une couleur de fond et un texte indiquant « `LeBoutonParent` ».

La couleur provient du `Background` du `TextBlock` lui-même (`Aquamarine`), quant au texte qu'il affiche il provient de la propriété `Name` de l'objet en cours de templating, c'est-à-dire le bouton.

Ce type de binding à l'intérieur d'un template permettant de référencer l'objet parent du template est très puissant puisqu'il s'affranchit de références explicites. Si le template était défini dans un dictionnaire de ressource au lieu d'être intégré au bouton, il serait facilement applicable à d'autres boutons, et tous afficheraient ainsi leur nom alors que tous seraient des instances différentes possédant un nom différent. La technique peut être utilisée à des choses plus utiles, cela va sans dire et elle fait partie des astuces utilisées dans l'art du templating (qui s'effectue le plus souvent sous Expression Blend).

Le Binding sur recherche d'ancêtre

Voici une autre forme de binding relatif. Son intérêt : permettre de se lier non plus à soi-même, pas même à l'objet parent du template, mais à n'importe quel objet de l'arbre visuel doté d'un type bien particulier qu'on spécifie.

Il est difficile de donner des exemples concrets de l'utilisation de ces formes si particulières de binding sans avoir à présenter des objets complexes au sein de projets qui le sont tout autant. C'est pourquoi mes exemples de templating sont ici courts, mais moches, avouons-le ! Votre imagination est un champ infini, le binding sur recherche d'ancêtre est une des clés qui permet d'en ouvrir les portes…

Reprenons toutefois l'exemple précédent et ajoutons deux autres `TextBlock` à l'intérieur du `StackPanel` du template du bouton (je ne vous donne que la déclaration de ces deux contrôles, regardez le code du template un peu plus haut. N'oubliez pas non plus le projet « `RelativeBinding` » dans lequel se trouve les sources de cet exemple) :

```
<TextBlock Background="Azure"
   Text="{Binding RelativeSource={RelativeSource FindAncestor,
          AncestorType={x:Type Button}}, Path=ToolTip}"/>
<TextBlock Background="CadetBlue" Text="{Binding
          RelativeSource={RelativeSource FindAncestor,
          AncestorLevel=1, AncestorType={x:Type StackPanel}}, Path=Name}"/>
```

XAML 44 - Binding Relatif

Le résultat visuel est :

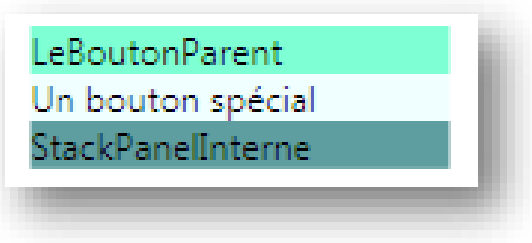


Figure 61 - Binding sur recherche d'ancêtre

Le premier `TextBlock` créé un binding sur une source qui est définie comme étant une recherche sur tous les ancêtres visuels du contrôle courant, recherche devant s'arrêter dès qu'elle trouve un élément de classe `Button`. Quand cela est fait, le binding est créé sur la propriété `ToolTip` de ce dernier. N'oubliez pas que tout cela se

tient à l'intérieur d'un bouton templaté même si visuellement on ne voit plus rien qui ressemble à un bouton.

Et c'est bien le texte du `ToolTip` du bouton que nous voyons en seconde ligne (la première ligne provenant de l'exemple précédent).

Mais on peut aller plus loin avec la recherche d'ancêtre, par exemple en indiquant le niveau de profondeur recherché.

Ainsi, le second `TextBlock` de cet exemple (le troisième au total) indique-t-il, en plus du type d'ancêtre recherché (le type `StackPanel`), un niveau de recherche où s'arrêter : la valeur 1. De ce fait c'est le `StackPanel` interne, celui qui est défini à l'intérieur du template, qui est trouvé. Une fois la localisation de ce contrôle effectuée, le binding est créé sur la propriété `Name` de ce dernier. C'est pourquoi la troisième ligne affichée indique « `StackPanelInterne` », valeur de la propriété `Name` du `StackPanel` contenu dans le bouton (voir le code du template plus haut).

Peut-on aller encore plus loin ?

Oui, car le binding sur recherche d'ancêtre (visuel), tel un passe-muraille est capable de sortir des frontières du template en cours, même de dépasser les limites de l'objet définissant le template !

Imaginons que nous ajoutions un autre `StackPanel` qui ne fait qu'entourer le bouton de l'exemple. En toute logique appelons-le `StackPanelExterne`. Il s'agit donc d'un parent visuel du bouton sans aucun lien avec ce dernier que cette parenté visuelle, cela pourrait être un `Canvas`, une `Grid` ou tout autre conteneur.

Il est alors possible de référencer ce contrôle par un binding sur recherche d'ancêtre en indiquant un niveau de profondeur de « 2 » (le niveau 1 étant rencontré lorsque la recherche trouve le `StackPanel` interne) :

```
<TextBlock
       Background="Coral"
       Text="{Binding RelativeSource={RelativeSource FindAncestor,
              AncestorLevel=2, AncestorType={x:Type StackPanel}},
              Path=Name}"
/>
```

XAML 45 - Binding Relatif avec FindAncestor

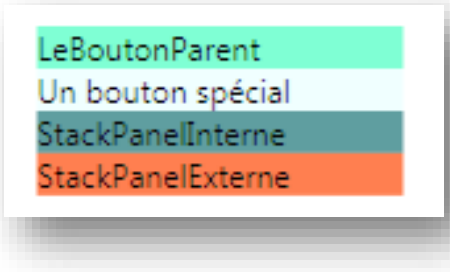


Figure 62 - Binding sur recherche d'ancêtre avec niveau de profondeur

La dernière ligne ajoutée indique bien le nom du `StackPanel` externe, c'est à dire celui qui est au-dessus du bouton dans l'arbre visuel.

Cette technique est assez risquée malgré tout. Tant qu'on reste à l'intérieur du template ou de l'objet qui le supporte on sait à l'avance ce qui peut s'y trouver. Dès lors qu'on tente de référencer des objets extérieurs on prend le risque qu'une modification du visuel fasse s'écrouler le binding et le fonctionnement de l'interface visuelle. Imaginons simplement dans notre exemple que nous décidions de changer le `StackPanel` extérieur en une `Grid`, le dernier binding étudié ne fonctionnera plus. Le risque de régression est donc très grand.

Nous avons vu le binding réflexif et socratique, le binding gendarme, sachant toujours se référer à sa hiérarchie, et enfin le binding spectre traversant les murs… Un drôle de bestiaire mais qui, bien exploité, confère une puissance incroyable à Xaml. Hélas, je n'oublie pas l'un des thèmes de cet article : vous avertir des dangers du binding.

Toutes ces formes alambiquées de ligatures, et encore plus ces dernières codant des niveaux de recherche dans des types se trouvant dans l'arbre visuel, toutes ces techniques puissantes sont malheureusement très délicates à appliquer sans se prendre les pieds dans le tapis, surtout sur de gros projets écrits à plusieurs. Xaml est au final un langage comme les autres, mais il ne dispose pas des analyseurs de code et du typage fort d'un C#, ni même d'un débogueur de qualité approchante. Toute erreur, tout bricolage, et pire encore tout code spaghetti créée par manque de maîtrise de la complexité de l'ensemble aboutira à un édifice incontrôlable et presque impossible à maintenir.

Servez-vous de tout cela avec beaucoup de discernement et de doigté.

Je ne vous encourage pas à éviter la nouveauté dans un réflexe conservateur et passéiste, bien au contraire, sinon je masquerai toutes ces possibilités de Xaml au lieu

de vous les démontrer. Non, mon message est clair : soyez hyper pro jusqu'au bout des ongles, et utilisez tout Xaml. Mais si jamais vous avez un doute sur votre maîtrise du sujet, ne sabotez pas votre projet : évitez ces complications, écrivez du code C# qui sera facile à maintenir. Vous ne passerez peut-être pas pour un héros à la machine à café, mais vous aurez le droit au sommeil du juste, celui qui a fait son travail proprement sans saboter le travail des autres. *C'est peu, mais c'est énorme !*

Le Binding PreviousData

Parmi toutes les formes de binding c'est peut-être l'une des moins connues. De toute façon depuis que nous avons quitté les premiers exemples de `OneWay` et `TwoWay` nous avons laissé depuis longtemps derrière nous la plage bondée de touristes et nous nageons en eaux profondes où seuls quelques gros poissons se risquent à frayer. Certains ont de grandes dents, et c'est pour cela que j'attire à chaque fois votre attention sur la grande vigilance qui doit accompagner l'utilisation de ces formes avancées de binding…

Le mode `RelativeSource PreviousData` est malgré tout un mode de binding très intéressant. Nous avons vu jusqu'à maintenant des modes relatifs qui permettent de jouer avec l'arbre visuel : accès à soi-même (`Self`), ou aux autres contrôles du template, voire à ceux se trouvant encore plus haut dans l'arbre (`FindAncestor`).

Qu'en est-il lorsque nous templatons des objets capables de présenter des données (au sens large), c'est-à-dire des suites d'objets, et que dans le template que nous créons la présentation ou la mise en page (ou tout autre élément du layout) dépend des données *précédentes* ?

Soyons encore plus clair. Imaginez un Bar Chart. Le Chart en lui-même est conçu pour afficher une série de données. Chaque donnée est affichée par un objet qui peut être templaté. Quand on écrit le template des barres on ne l'écrit qu'une fois. Bien entendu il sera appliqué à l'identique à chaque barre. Et au moment de la conception du template nous ne connaissons rien des données qui seront affichées, d'autant qu'elles seront différentes à chaque fois.

Imaginons alors qu'au-dessus de chaque barre nous souhaitions afficher un indicateur de tendance. Une petite flèche orientée vers le haut ou le vers le bas selon que la donnée en cours possède une valeur plus grande ou plus petite que la *précédente* ?

Ici aussi nous pourrions gérer la chose ponctuellement en jouant avec les événements du Chart et en écrivant du code behind (quel que soit le contrôle utilisé pour créer le

Chart). Mais il est vrai que ce n'est pas d'une grande élégance et que cette solution ne sera guère portable d'un Chart à l'autre, sauf à recopier le code ou à le complexifier encore plus pour le rendre générique. Certains iront même à créer un `UserControl` spécifique pour régler ce problème d'encapsulation de code, complexifiant encore et encore la totalité du projet.

L'écriture d'un template applicable par un simple clic à tout objet Chart est de loin la solution la plus propre et la plus satisfaisante intellectuellement et techniquement.

Mais il va falloir se débrouiller pour que le template des barres soit capable de référencer la donnée « précédente », celle de la barre affichée juste « avant » afin de gérer le différentiel et afficher la bonne flèche de tendance.

Ci-dessous une capture du visuel de la démo vous fera mieux comprendre le résultat escompté. En haut de chaque barre du Chart on trouve le différentiel de progression ou de régression relatif à la barre précédente ainsi qu'une flèche verte ou rouge orientée vers le haut ou le bas. C'est pour obtenir cet effet que le mode de binding `PreviousData` va être utile.

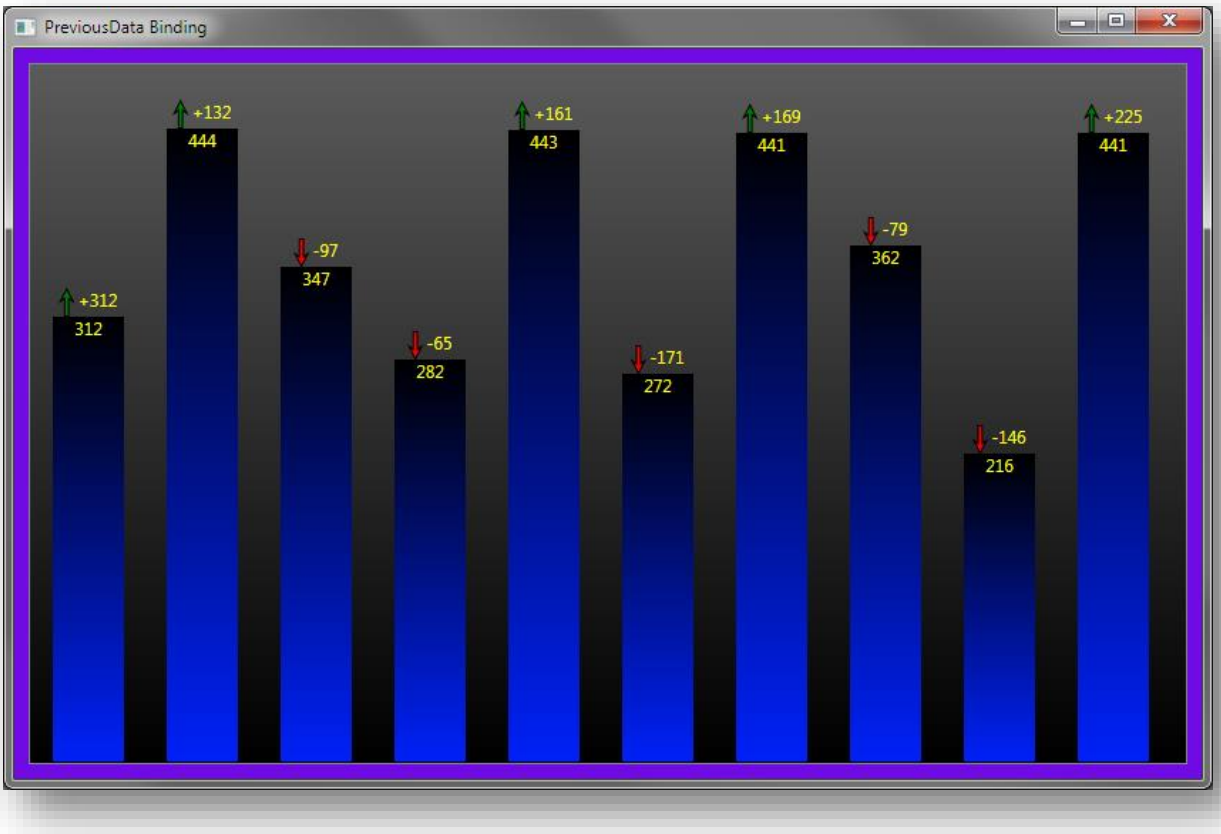


Figure 63 - Binding PreviousData

Le but de cet article étant toujours et encore le binding lui-même et non le code annexe des projets de démo, j'ai tenté jusqu'à maintenant de vous présenter des projets très simples ne montrant que l'essentiel. Ici, même en faisant au plus juste, le code sera un peu plus étoffé puisque nous avons besoin de données, de template et de quelques convertisseurs pour faire marcher l'ensemble.

Premièrement nous n'allons pas utiliser de composant Chart plus ou moins sophistiqué, cela démontrera en plus que les composants de base de WPF et Silverlight sont bien assez riches pour qui a un peu d'imagination. Car en regardant la figure ci-dessus, pouvez-vous imaginer que le composant principal n'est rien d'autre… qu'une simple `Listbox` ?

Passons rapidement sur ce code annexe mais nécessaire au fonctionnement de la démo :

La source de données

Le plus simple est de créer une classe dérivant d'ObservableCollection générant automatiquement une collection de 10 entiers aléatoires :

```
class RecordsCollection : ObservableCollection<int>
    {
        public RecordsCollection()
        {
            var r = new Random();
            for (var i = 0; i < 10; i++) Add(r.Next(200, 450));

        }
    }
```

Code 31 - Source de données de test aléatoire

Il faut déclarer une instance de cette classe dans les ressources de la fenêtre en cours, avec en préambule l'ajout de l'espace de nom dans la balise de la fenêtre :

```
xmlns:Data="clr-namespace:PreviousDataBinding.Data"
```

La ressource sera définie comme suit :

```
<Window.Resources>
        <Data:RecordsCollection x:Key="data"/>
</Window.Resources>
```

La visibilité des flèches : la magie de PreviousData et du Binding Multiple

Les deux flèches (verte et rouge) sont des Path. Pour la simplicité les deux flèches sont affichées l'une sur l'autre, il faut donc, selon le cas, en cacher une et montrer l'autre pour ne voir que celle qui correspond à la tendance.

Le projet `PreviousDataBinding` fourni avec l'article contient bien entendu la totalité du code. Pour l'inspection des ressources, des templates et des Path je vous conseille vivement d'utiliser Expression Blend qui est mille fois mieux adapté pour ce genre de travail.

Pour calculer la visibilité d'une flèche il nous faut connaître la valeur courante ainsi que la valeur de la précédente barre du Chart.

Comme indiqué plus haut le composant que nous templatons ici n'est autre qu'une `ListBox` dont `l'ItemPanelTemplate` a été modifié pour utiliser un `StackPanel` en mode horizontal. Le template qui nous intéresse est celui des items (`ItemTemplate`), c'est lui qui définit les barres et les données affichées (textes et flèches). C'est un `DataTemplate` que nous utilisons (voir le code source complet du projet).

Pour obtenir la valeur courante et la précédente en même temps il faut définir un Binding Multiple que nous avons déjà étudié.

Voici un extrait du `DataTemplate`, au niveau de la définition du `Path` représentant la flèche verte (rappelons que par défaut la rouge est `Collapsed` et que nous ne nous occupons que de rendre la verte visible ou invisible) :

```
<Path.Visibility>
  <MultiBinding Converter="{StaticResource RecordsToVisbilityConverter}">
    <Binding/>
    <Binding RelativeSource="{RelativeSource PreviousData}"/>
  </MultiBinding>
</Path.Visibility>
```

XAML 46 - MultiBinding et Binding Relatif de type PreviousData

Plusieurs choses sont à noter ici : la première est l'utilisation d'un multiple binding que nous avons déjà indiqué. On remarque l'utilisation d'un convertisseur. Ensuite vient le contenu même de ce binding qui est formé de deux valeurs, l'une exprimée

comme un binding à l'item courant de la collection et l'autre utilisant le fameux `PreviousData`.

L'explication est finalement assez simple : En définissant un binding multiple nous faisons en sorte de pouvoir traiter plusieurs valeurs à la fois. Ces valeurs sont au nombre de deux : la valeur courante de la barre et la valeur de la barre précédente. C'est pour cela que nous avons deux bindings dans la balise multiple. Le premier est un binding simple (item courant de la collection) qui représente la valeur courante de l'Item de la `ListBox` (puisque c'est cet Item que nous templatons au travers d'un `DataTemplate`). Rappelons que la source de données de la `ListBox` n'est autre que l'objet collection que nous avons présenté un peu plus haut, objet qui retourne une liste de dix valeurs entières aléatoires. C'est donc une telle valeur entière unique que devra traiter le `DataTemplate`. Dans d'autres cas le `DataTemplate` peut avoir à traiter un objet plus complexe qu'un entier (une fiche article, une facture…). En choisissant un item aussi simple qu'une instance de `int` on ne peut pas simplifier plus l'exemple…

Le second binding utilise enfin le mode `RelativeSource PreviousData` qui retourne la valeur précédente, donc relative à la position de la valeur courante. Donc l'entier précédent dans la collection source de la `ListBox`.

Armés de ces deux bindings qui nous permettent d'obtenir les deux valeurs qui nous intéressent, utilisons la capacité du binding multiple pour fusionner ces dernières en une information différente : la visibilité du `Path`. C'est grâce au convertisseur spécial du Multi Binding que nous pouvons réaliser ce tour de force.

Voici le code de ce dernier :

```
public class RecordsToVisibilityConverter : IMultiValueConverter
{

    public object Convert(object[] values, Type targetType,
           object parameter, System.Globalization.CultureInfo culture)
    {
        int currentValue, previousValue;
        var result = Visibility.Visible;

        if (values[0] == null || values[1] == null) return result;
        if (!int.TryParse(values[0].ToString(), out currentValue))
           return result;
        if (!int.TryParse(values[1].ToString(), out previousValue))
           return result;
        if (previousValue > currentValue)
           result = Visibility.Collapsed;
        return result;
    }
    public object[] ConvertBack(object value, Type[] targetTypes,
                                object parameter,
                        System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }

}
```

Code 32 - Multi convertisseur avec différentiel de données

L'intérêt d'utiliser un multi binding se révèle donc ici : il permet d'obtenir plusieurs valeurs et son convertisseur sait traiter toutes ces valeurs pour en fabriquer une nouvelle (valeur de sortie) qui peut être de tout type. Ici, c'est une valeur de visibilité que nous créons selon le différentiel entre les deux valeurs d'entrées reçues.

Vous aurez noté que le multi binding (voir plus haut) est inscrit dans une balise `Path.Visibility`. C'est donc bien une valeur de visibilité que nous retournons et qui affectera directement la visibilité de la flèche verte.

Vous allez me dire, certes, mais la flèche rouge alors ? On voit bien à la forme des flèches que si elles étaient juste superposées on verrait les pointes de la flèche se trouvant sous la première… Et dans la définition du code Xaml de la première flèche (non concernée par le binding que nous venons de voir) il n'y aucun convertisseur

connecté ni aucun binding, comment peut-elle devenir visible lorsque la flèche verte est cachée (résultat du calcul du binding) ?

On pourrait être tenté d'implémenter le même mécanisme de binding sur la première flèche, il suffirait de passer un paramètre supplémentaire au convertisseur pour lui indiquer de fonctionner dans un sens ou son inverse.

Mais cela serait se compliquer la vie pour pas grand-chose (ainsi que le code qui entrerait alors dans le menu des spécialités italiennes, les fameux code spaghetti !).

Il y a bien plus simple. Il suffit de déclarer maintenant un *Property Trigger* dans notre template en surveillant le changement de la valeur `Visibility` de la flèche verte. Lorsque cette valeur change il ne reste plus qu'à appliquer son contraire à la flèche rouge et le tour est joué ! En réalité c'est même plus simple puisque par défaut la flèche rouge est `Collapsed`. Il n'est donc nécessaire de pister que le changement de visibilité de la flèche verte quand elle passe à `Collapsed`. Puisque lorsqu'elle est visible la rouge est déjà cachée par défaut…

C'est ainsi que dans le `DataTemplate` (celui que nous écrivons en ce moment et qui sera utilisé comme template d'Item pour la `ListBox`) nous trouvons :

```
<DataTemplate.Triggers>
  <Trigger SourceName="GreenArrow" Property="Visibility" Value="Collapsed">
    <Setter TargetName="RedArrow" Property="Visibility" Value="Visible"/>
  </Trigger>
</DataTemplate.Triggers>
```

XAML 47 - Trigger de DataTemplate

Quand la flèche verte voit sa propriété `Visibility` passer à `Collapsed`, alors nous changeons la visibilité de la flèche rouge à `Visible`.

Voilà à quoi peut servir le mode `PreviousData` du binding relatif. Ce n'est qu'un exemple, très simple bien qu'utilisant plusieurs astuces. J'espère que mes explications sont assez claires et que cela est en train de faire bouillonner vos neurones qui se demandent maintenant comment utiliser cela dans un projet !

Le contenu des textes

Le mécanisme est en gros le même pour les textes. Il est même plus simple puisque les deux textes sont toujours visibles, seul le contenu de l'un est obtenu par calcul grâce un multi binding faisant la différence entre la valeur en cours et la précédente.

La technique de binding utilisée étant identique, je renvoie le lecteur au code source où il trouvera tout ce qu'il faut pour disséquer lui-même cette partie de la démo (code Xaml, autre convertisseur `RecordsDifferenceConverter`, etc).

Le Template binding

Cette forme de binding relatif est particulièrement utile. Il s'agit de pouvoir référencer directement une propriété nommée dans l'élément sur lequel le template est appliqué.

La syntaxe conventionnelle :

```
Lapropriété = "{Binding RelativeSource={RelativeSource TemplatedParent},
Path=XXX}"
```

Le `Path` est celui de la propriété visée.

La syntaxe raccourcie :

```
Lapropriété = "{TemplateBinding xxxx} »
```

C'est ce raccourci qui donne d'ailleurs son nom usuel à cette forme de binding, le *template binding*.

Utilité

Il est bon de dire quelques mots sur cette forme de binding car elle joue un rôle très important dans la création des templates bien conçus.

Prenons l'exemple d'un bouton. Créons un template pour changer son look & feel. Retirons tout ce que le bouton contient pour construire quelque chose d'autre. Ici nous allons supposer quelque chose d'extrêmement simple qui du point de vue visuel sera assez laid, mais l'esthétique n'est pas l'objet de notre propos.

Donc pour templater ce bouton, une fois que nous l'avons vidé de tout ce qu'il contient, ajoutons un rectangle que nous remplissons en vert, ajoutons un `ContentPresenter`, et voilà. Un joli bouton rectangulaire, vert, affichant par défaut le mot « `Button` ». Ce bouton ne gère aucun changement d'état visuel mais cela n'a pas d'importance pour la démo.

Vous l'imaginez bien ce template de `button` ? Ok (il faut dire que j'ai vraiment choisi quelque chose de simple !).

Maintenant sur votre fiche, placez un bouton et appliquez-lui le template.

Le bouton devient un rectangle vert avec le texte par défaut affiché. Super.

Maintenant vous vous dites, ce vert est vraiment épouvantable dans mon application qui joue plutôt sur les tons de bleu. Autant ce magnifique look flat rectangulaire me plait, autant, ce vert…

Qu'à cela ne tienne ! Votre bouton étant sélectionné, vous regardez dans ses propriétés et vous décidez d'attribuer un joli bleu issu de votre charte de couleurs à la propriété `Background`.

Rien. Le bouton reste ostensiblement vert. Vous tentez une autre couleur (l'informaticien ne s'avoue pas vaincu au premier essai). Rien. On dirait même qu'il se moque de vous non ? Il est vert et restera vert. Vous, vous devenez rouge de colère avant d'être finalement … vert de rage !

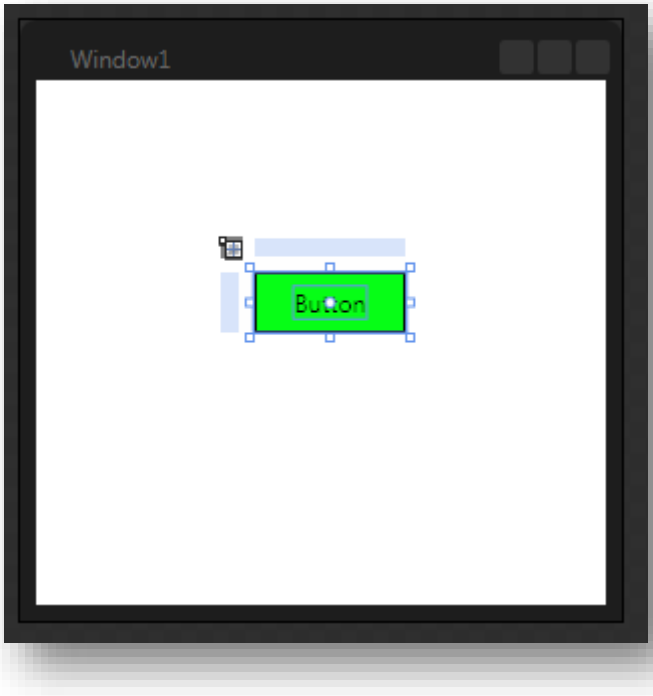


Figure 64 - Template Binding – Création du template sous Blend

Sous Blend en mode templating voici à quoi ressemble notre bouton (ci-dessus).

Posons un second bouton sur la fiche :

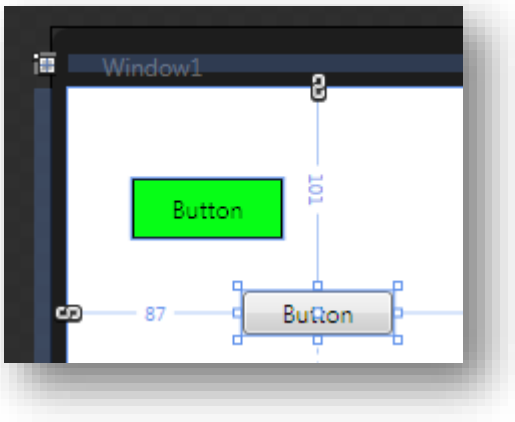


Figure 65 - Template binding - application du template 1/2

Et appliquons-lui le template :

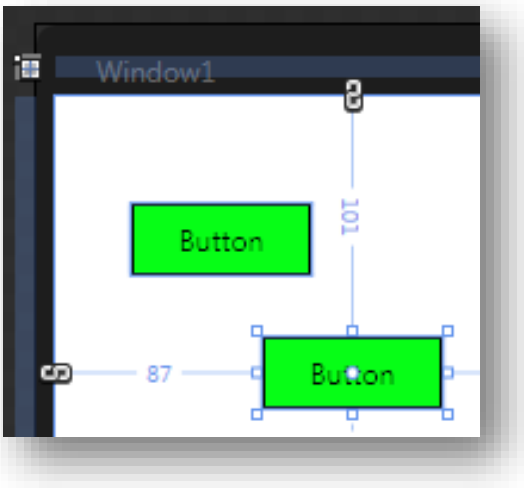


Figure 66 -Template binding - application du template 2/2

Tentons de changer sa couleur :

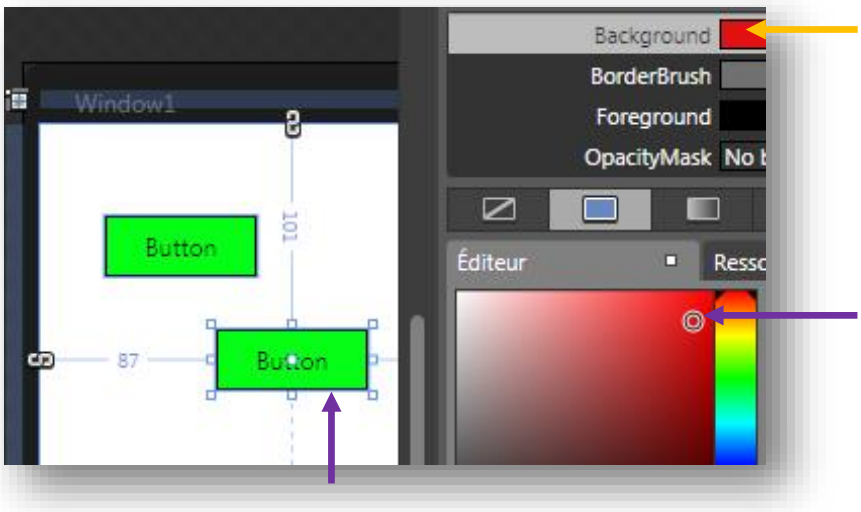


Figure 67 - Template binding - Essai de changement de Background

Il reste vert…

C'est agaçant pour celui utilise le template. C'est frustrant de voir cette belle palette de couleur n'avoir aucun effet sur ce bouton alors même qu'il existe une propriété « `Background` » et qu'on peut visiblement la changer. Mais cela ne sert à rien, *comme si quelque chose était cassé*.

C'est bien ce qui s'est produit. En vidant le template original du bouton *nous avons cassé la mécanique visuelle* ainsi que tous les liens qui existaient entre la classe `Button` et sa représentation visuelle…

De fait, la propriété `Background` de `Button`, si elle existe toujours (la classe `Button` n'a pas changé) ne sait plus à quoi s'appliquer. Comment le bouton pourrait-il « deviner » que le `Background` doit maintenant être appliqué au `Fill` d'un `Rectangle` ? Et notre template est simplissime, dans un vrai template il y a souvent plusieurs objets (pour créer l'effet glossy, pour gérer l'état `disabled`, etc) comment le bouton pourrait-il deviner lequel joue le rôle de `Background` visuel ?

C'est impossible voilà tout. La magie n'existe que dans les spectacles de music-hall.

Il faut une intervention humaine pour expliquer au template qu'on désire que la propriété `Fill` du `Rectangle` soit liée à la propriété `Background` du contrôle en cours de templating.

C'est à cela que sert le mode `TemplateBinding`.

Sous Blend cela est très simple à mettre en place en quelques clics.

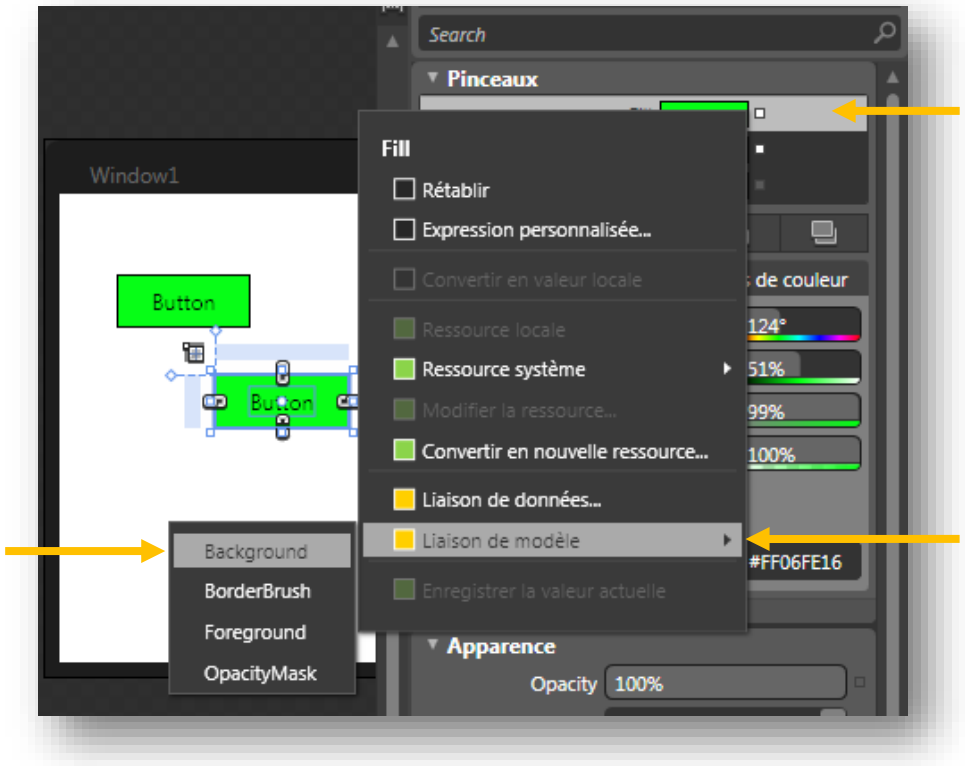


Figure 68 - Template binding - création du lien sous Blend

Comme le montre la figure ci-dessus, Blend permet de définir ce fameux lien entre une propriété de l'un des objets du template avec une propriété compatible appartenant au modèle (la classe sur laquelle s'applique le template).

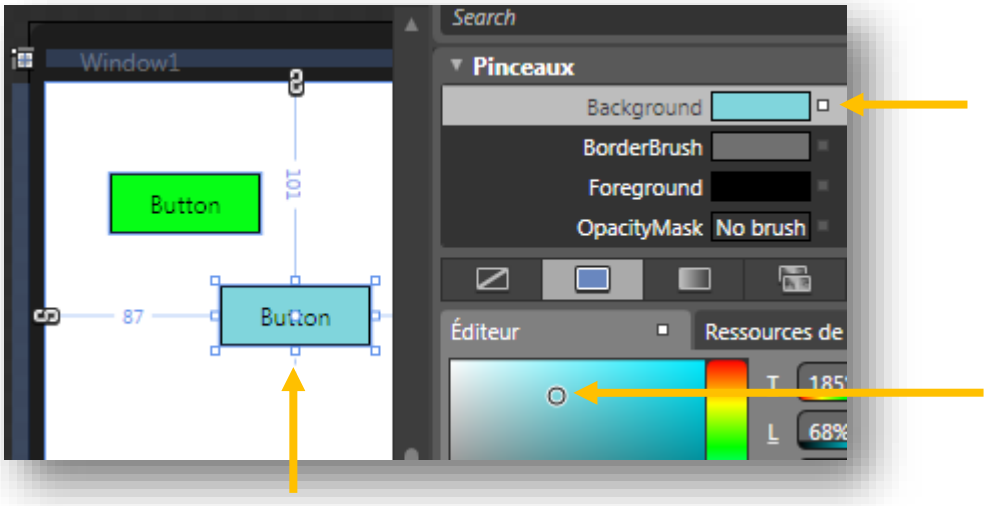


Figure 69 - Le bouton correctement templaté

Maintenant que le template a été correctement conçu il est possible à l'utilisateur de ce dernier de pouvoir changer la couleur du bouton en modifiant sa propriété

Background. Le template sait maintenant comment relier cette propriété à l'effet visuel recherché (ici changer la propriété Fill du Rectangle).

Blend a créé pour nous le code Xaml, le voici :

```
<Rectangle Fill="{TemplateBinding Background}" Stroke="Black" />
```

XAML 48 - Template Binding

C'est la "version courte" que Blend utilise pour créer ce binding et non la syntaxe plus longue de binding relatif.

On comprend d'ailleurs la raison d'être d'une syntaxe raccourcie, c'est que ce binding bien particulier est très utilisé (ou devrait l'être, ce qui fait une nuance, hélas) !

Dans le projet fourni « Templatebinding » vous trouverez deux templates, ButtonControlTemplate1, le template mal conçu qui est accroché au premier bouton (celui qui reste vert) et ButtonControlTemplateCorrigé, le template corrigé qui permet de changer la couleur du bouton.

Le template binding (ou liaison de modèle) est ainsi un outil dont la compréhension est indispensable pour qui veut créer des templates utiles et versatiles.

Dernière question : « Lorsque j'applique un template binding, par exemple au Background comme dans l'exemple que nous venons de voir, je n'ai plus de contrôle sur la couleur de fond de mon objet ! Si je veux qu'il soit vraiment vert, même si après l'utilisateur du template peut changer cette valeur, comment faire pour qu'il y ait une couleur par défaut différente de celle du contrôle original ? »

La question est légitime… La réponse tient en un mot : *Style*. Il suffit de créer un style qui fixe les paramètres comme le Background, style qui englobera le template. Blend le fait automatiquement, tout template est créé avec un style l'englobant. Si on code sous VS ou directement en Xaml « à la main », il ne faut pas oublier l'étape du style.

Pour bien comprendre le rôle de chacun (qui semble être confus pour beaucoup de développeurs), le template sert à modifier la forme, le look & feel, le style sert à fixer des valeurs aux paramètres de l'objet. C'est dans le template qu'on créée un bouton rond au lieu de rectangulaire, mais c'est dans le style (et après application du template binding) qu'on fixe une couleur verte par défaut au bouton.

Le Collection Binding

Son but est assez simple : permettre un binding directement sur l'item courant d'une collection lorsque le `DataContext` est un objet de ce type.

Prenons l'exemple d'une `ListBox` affichant une liste de personnes. La `ListBox` est placée dans la grille principale de la fenêtre, c'est cette dernière qui possède le `DataContext`. Ce dernier pointe une ressource qui est une `ObservableCollection` constituée d'items de type Personne (une classe ayant deux propriétés : `Nom` et `Prénom`).

La `ListBox` possède un `ItemTemplate`, défini sous la forme d'un `DataTemplate` placé dans les ressources de la fenêtre. Ce `DataTemplate` utilise pour sa part un binding que nous avons déjà vu, le binding simple faisant implicitement référence au `DataContext` (syntaxe `{Binding xxx}`). Chaque item affiche ainsi le nom et le prénom de la personne.

Maintenant, nous souhaitons poser sur la même grille (donc partageant le même `DataContext` que la `ListBox`) une zone précisant la sélection actuelle mais sans passer par la `ListBox`, uniquement en se référant à l'élément courant du `DataContext`, donc de la collection sous-jacente.

Regardons le résultat (projet exemple `ListBinding`) avant d'aller plus loin :

Figure 70 - Le Collection Binding

On voit que l'item sélectionné dans la `ListBox` est le second (trait bleu sur le côté gauche) et on peut voir que dans le cadre « Votre sélection » se trouve exactement les mêmes informations.

Les deux `TextBlock` de ce cadre sont programmés comme suit :

```
<TextBlock Text="{Binding /Nom, Mode=Default}" />
<TextBlock Text="{Binding /Prénom, Mode=Default}" />
```

XAML 49 - Default binding

Pour la lisibilité j'ai supprimé tout ce qui concerne la présentation (couleur, position …).

La syntaxe de ce binding particulier est ainsi `{Binding /}` pour se référer à l'élément courant de la collection, ou bien comme utilisé ici `{Bindind /xxx}` où `xxx` est le nom de la propriété à laquelle on souhaite se lier dans l'élément courant.

Si le `DataContext` est un objet plus complexe qui contient une propriété de type collection, on peut encore utiliser cette syntaxe mais en préfixant le slash par le nom de la propriété collection. Par exemple avec un `DataContext` exposant une propriété Personnes (une collection), on écrira `{Binding Personnes/}` pour l'item courant, ou `{Binding Personnes/Nom}` pour se lier à la propriété `Nom` de l'item courant.

Il faut noter que la notion d'item courant n'est pas forcément automatique. Il faut que la position dans la collection soit gérée par « quelqu'un ». Pour se faire on peut utiliser, par code, une `CollectionView` par exemple. Si, comme dans notre exemple, il y a une `ListBox` qu'on considère être le « pilote » de l'item courant, il faut alors positionner sa propriété `IsSynchronizedWithCurrentItem` à `true`. C'est ce que fait notre exemple.

Le Priority Binding

Je croyais sincèrement être arrivé au bout des types de binding et pouvoir aborder gentiment la fin de cet article qui, comme celui sur MVVM est devenu un livre à lui-seul au fil des ajouts (record battu, MVVM ne faisait « que » 70 pages !)… Hélas pour moi, mais heureusement pour vous, ma conscience me poussant à toujours tout vérifier, je suis tombé sur un (je n'ose pas écrire le mot…) dernier (?) type de binding : le Priority Binding.

Ce mode bien particulier propose de réaliser une liste de bindings dans laquelle il existera une hiérarchie, une priorité. Si le binding de plus haut rang retourne une valeur, c'est elle qui deviendra la valeur courante, si ce binding échoue c'est le prochain sur la liste qui sera essayé jusqu'à temps que l'un des bindings de la liste retourne quelque chose. L'ordre dans lequel les bindings sont parcourus pour obtenir une valeur est toujours le même, c'est celui fixé par la liste, l'ordre des priorités. Cette vision est simplificatrice pour expliquer le fonctionnel, en réalité les choses sont plus subtiles nous allons le voir, notamment avec gestion de threads d'arrière-plan.

Xaml ne passe pas son temps à balayer la liste, bien entendu au final le binding actif est choisi dynamiquement mais en respectant les priorités fixées par la liste. Par exemple si un binding de faible priorité retourne une valeur en premier, elle deviendra la valeur active. Mais dès qu'un binding de rang plus élevé retournera à son tour une valeur c'est cette dernière qui deviendra la valeur active. On peut penser par exemple à un binding de faible priorité allant puiser sa valeur dans une ressource locale (donc rapide) et un autre binding de rang plus élevé mais allant chercher sa valeur dans le Cloud, donc plus lent mais peut-être plus à jour. Tant que la valeur du Cloud n'est pas arrivée c'est la valeur locale qui est affichée, dès que la valeur en provenance du réseau est disponible c'est elle qui est affichée.

Un autre exemple est celui de Word. Lorsque vous ouvrez un document, Word affiche le nombre de pages immédiatement, c'est une estimation. Si le document est très gros il faudra un peu de temps pour qu'il soit chargé et analysé, le nombre de pages affiché deviendra alors le « vrai » nombre de pages du document. Si on devait réaliser Word en WPF, on utiliserait le Priority Binding avec deux bindings : le premier pointerait sur une propriété « `estimation nombre de pages` » qui résulterait d'un calcul approximatif, de priorité basse, le second serait retourné par une propriété « `vrai nombre de pages` » de priorité haute retournant la valeur exacte. Aucune programmation, aucun timer ne serait nécessaire pour obtenir l'effet recherché…

Pour résumer, le Priority Binding s'avère très utile dans des situations où des données assez lentes à obtenir ont un impact sur l'interface visuelle. Pour éviter que cette dernière soit dans un état intermédiaire un peu « en friche », on peut, grâce au Priority Binding, proposer des valeurs d'attente, rapidement calculées ou obtenues localement qui seront remplacées par les données définitives lorsqu'elles arriveront.

Bref, vous avez compris l'intérêt je pense.

Mettons tout cela en scène avec un exemple.

Supposons une fenêtre affichant un `TextBlock`. La valeur affichée par ce dernier, un texte, sera puisée de trois sources différentes mixées dans un seul Binding, un Priority Binding. La première source sera très rapide mais de faible priorité, la seconde un peu lente et de priorité intermédiaire, et la troisième très lente mais de très haute priorité.

Pour simuler tout cela de façon simple, c'est l'objet `Window` lui-même qui fournira trois propriétés. Une sera directe, les deux autres utiliseront un `Thread.Sleep()` pour ralentir leur lecture et simuler des sources lentes (gros calcul, accès Web…).

La gestion des threads sera automatiquement prise en compte par le Priority Binding pour peu qu'on indique un paramètre supplémentaire, notamment sur les propriétés lentes utilisant un `Thread.Sleep()`. Cette option est `IsAsync` qu'on passe à `true` si on désire que le binding ainsi marqué soit placé automatiquement en file d'attente dans un thread d'arrière-plan.

La séquence que le projet `PrioriryBinding` vous propose sera la suivante dès que vous l'exécuterez :

Immédiatement vous verrez cet affichage :

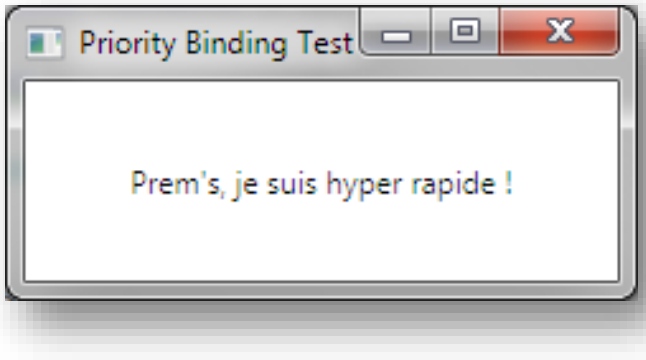


Figure 71 - Priority Binding - 1/3

Au bout d'une seconde et demie l'affichage va devenir le suivant :

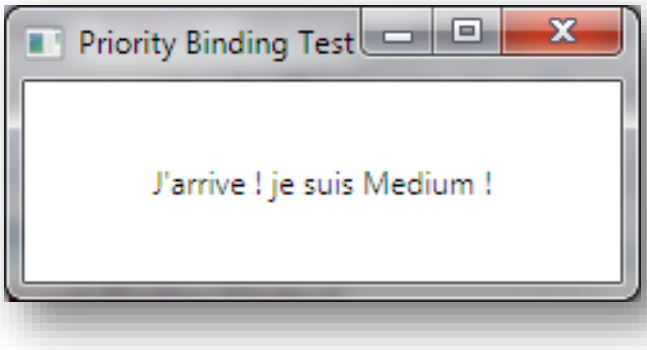


Figure 72 - Priority Binding - 2/3

Au bout de trois secondes (en partant du premier affichage) l'affichage va enfin devenir le suivant et ne changera plus :

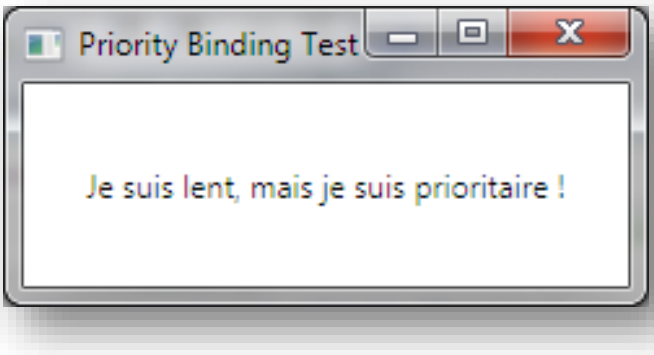


Figure 73 - Priority Binding - 3/3

Fascinating ! (comme dirait M. Spock).

Voici les trois propriétés déclarées directement dans la fenêtre `Window1` :

```
        /// <summary>
        /// Ressource très prioritaire mais très lente à obtenir...
        /// </summary>
        /// <value>The slow.</value>
        public string Slow
        {
            get
            {
                Console.WriteLine("Slow: " +
                                  Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(3000);
                return "Je suis lent, mais je suis prioritaire !";
            }
        }

        /// <summary>
        /// Ressource assez lente à obtenir de priorité intermédiaire
        /// </summary>
        /// <value>The medium.</value>
        public string Medium
        {
            get
            {
                Console.WriteLine("Medium: " +
                                  Thread.CurrentThread.ManagedThreadId);
                Thread.Sleep(1500);
                return "J'arrive ! je suis le Medium !";
            }
        }

        /// <summary>
        /// Ressource très rapide à obtenir mais de priorité basse
        /// </summary>
        /// <value>The fast.</value>
        public string Fast
        {
            get
            {
                Console.WriteLine("Fast: " +
                                  Thread.CurrentThread.ManagedThreadId);
                return "Prem's, je suis hyper rapide !";
```

```
            }
        }
```

Code 33 - Mettre en évidence les propriétés de binding

Comme indiqué, la propriété « rapide » (`Fast`) ne possède aucun délai, les deux autres un `Thread.Sleep()` de 1500 et 3000 ms.

Vous noterez que chaque lecture de propriété est assortie dans le `Getter` d'un affichage console retournant le numéro du thread utilisé, ce qui permet de confirmer que chaque accès à bien lieu dans un thread séparé géré automatiquement par le Prioriy Binding.

Par exemple, pour l'exécution retracée plus haut (voir les trois figures), on trouve dans la console de sortie :

```
Fast: 9
Medium: 11
Slow: 10
```

L'accès à `Fast` s'est effectué via le thread 9, l'accès à `Medium` via le thread 11 et l'accès à `Slow` via le thread 10.

Reste à voir comment tout cela est spécifié sous Xaml. Vous allez voir, c'est très simple :

```
<TextBlock>
  <TextBlock.Text>
    <PriorityBinding>
      <Binding ElementName="MainWindow" Path="Slow" IsAsync="True" />
      <Binding ElementName="MainWindow" Path="Medium" IsAsync="True" />
      <Binding ElementName="MainWindow" Path="Fast" />
    </PriorityBinding>
  </TextBlock.Text>
 </TextBlock>
```

XAML 50 - Priority Binding

Cela ressemble à du Multi Binding, sauf qu'une priorité existe entre les bindings (la plus haute en premier) et que la finalité est une unique valeur, pas une valeur composite. Il n'y a pas non plus de convertisseur global, mais rien n'interdit d'en définir au niveau de chaque binding (qui sont eux des bindings « normaux », ceux

utilisés ici n'étant donc que des exemples). On note malgré tout une spécificité, la présence de `IsAsync` dans les binding lents. Cela n'est pas obligatoire mais sans cette indication sur les sources lentes le Priority Binding perdrait tout son charme. En effet, Xaml étant interprété dans l'ordre d'écriture, le premier binding rencontré serait celui de plus haute priorité mais le plus lent à obtenir. L'application se bloquerait jusqu'à l'obtention de la valeur. Une fois celle-ci acquise, comme elle est de plus haut niveau, les deux autres seraient ignorées. C'est donc tout l'intérêt du « montage » qui tomberait à l'eau sans la magie du multithreading que `IsAsynch` ajoute automatiquement (à condition de préciser cette option).

La question piège est de savoir pourquoi `IsAsync` est une option au lieu d'être systématique. On répondra qu'il s'agit de souplesse car c'est au développeur de décider. Une autre question est de savoir si le dernier binding doit ou non avoir l'option `IsAsynch` à `true` ou pas…

Si la valeur est réellement rapide à obtenir je vous conseille de ne pas utiliser l'option sur le dernier binding. Mais si le dernier binding, censé être le plus rapide, peut lui aussi prendre un peu de temps, pour éviter de figer l'application il semble naturel de le mettre en `IsAsynch true` aussi.

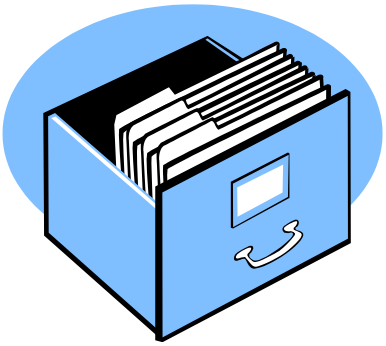
Dans ce cas cela signifie que pendant un certain laps de temps il n'y aura aucun affichage du tout pour le `TextBlock` (ou tout autre objet utilisant le même mécanisme). Cela peut être satisfaisant comme cela peut ne pas être acceptable… Dans ce dernier cas il faudra alors rajouter un nouveau binding de priorité encore plus basse mais capable de fournir immédiatement une valeur. Il n'aura alors pas l'option `IsAsynch` à `true`. A moins que… Non, il faut malgré tout que la chaîne s'arrête à moment où un autre, mais j'entrevois que dans certaines situations ce choix puisse devenir délicat, voire stratégique…

Les propriétés de tous les bindings

Xaml n'est qu'un langage de description qui est interprété et dont la finalité est l'instanciation d'objets ayant des relations entre eux. Cela veut dire que tout ce qui s'écrit en Xaml peut s'écrire en code C# par exemple, rien qu'en instanciant et en initialisant les bons objets dans le bon ordre et en créant les relations nécessaires entre eux.

Il n'est bien sûr pas question de programmer de cette façon ! Mais il faut avoir conscience que toute balise Xaml cache derrière elle une instance d'une classe. Cela

est vrai aussi pour le Binding. Et qui dit classe et instance, dit propriétés et méthodes. Les premières sont accessibles en Xaml, les secondes ne le sont que par code. Il est ainsi parfois nécessaire de récupérer en C# l'objet correspondant à une balise ou un binding pour accéder aux méthodes qu'il expose.



Dans cet article nous nous intéressons uniquement au Binding. Les méthodes de la classe Binding sont plutôt réservées à un usage « interne » du Framework ou à certains bricolages que je préfère ne pas imaginer…

Les propriétés de la classe Binding sont en revanche manipulables en Xaml et concentrent toutes les options qui permettent de façonner un lien. Il est intéressant de noter au passage que l'avènement d'un langage descriptif comme Xaml donne aux propriétés un rôle central qui tranche avec les langages de programmation des générations précédentes qui donnaient encore la priorité aux méthodes. C'est un glissement progressif, du plaisir[7], je ne sais pas, mais vers le descriptif et le fonctionnel, cela est certain.

Voici un récapitulatif des principales propriétés des Bindings et leur signification. Une grande partie a fait l'objet d'exemples dans le présent article. Pour les autres, cette liste attirera votre attention et vous donnera envie (je l'espère) d'en savoir plus ! J'ai ajouté des liens vers des informations complémentaires lorsque cela s'imposait (par exemple vers la définition complète de la syntaxe XPath).

Tableau 1- Propriétés d'un Binding

Propriété	Description
BindingGroupName	Spécifie le nom de BindingGroup. Les BindingGroups ajoutés dans WPF dernièrement (3.5sp1) simplifient par exemple la gestion des validations. Voir : http://msdn.microsoft.com/fr-fr/library/system.windows.data.binding_members.aspx
BindsDirectlyToSource	Lorsqu'on utilise une source dérivant de DataSourceProvider, en plaçant cette propriété à

[7] Les cinéphiles auront noté l'astucieuse référence au film de Robbe-Grillet de 1974…

	true on créé un lien avec l'objet provider plutôt qu'avec ses données.
Converter	Permet de spécifier le convertisseur à utiliser.
ConvertCulture	Permet d'indiquer et de forcer la culture que le convertisseur doit utiliser.
ConverterParameter	Le paramètre optionnel passé au convertisseur.
ElementName	Le nom de l'élément quand on se lie à un élément dans la même portée Xaml. Ne peut pas être utilisé si RelativeSource ou Source est déjà défini.
FallbackValue	La valeur à utiliser quand le binding rencontre une erreur.
IsAsync	A utiliser sur des binding lents afin que la récupération de la valeur soit effectuée automatiquement dans un thread d'arrière plan non bloquant.
Mode	Direction et signification du Binding (voir les différents modes expliqués dans cet article : OneWay, TwoWay, OneTime, …)
NotifyOnSourceUpdated	Force le déclenchement de l'événement SourceUpdated dès qu'une valeur est transférée de la cible vers la source.
NotityOnTargetUpdated	Force le déclenchement de l'événement TargetUpdated quand la cible est mise à jour depuis la valeur source.
Path	Indique la propriété source
RelativeSource	Permet de fixer la source de façon relative à la cible (voir les exemples dans cet article sur le Binding Relatif).
Source	Pointe l'objet qui doit être considéré comme la source du binding. Ne peut pas être utilisé si ElementName ou RelativeSource le sont.
StringFormat	Ajouté dans 3.5sp1 permet le passage d'une chaîne de format évitant parfois l'écriture d'un convertisseur (voir exemple dans l'article).
TargetNullValue	Ajouté dans 3.5sp1 indique la valeur à utiliser si la source est nulle.

UpdateSourceExceptionFilter	Permet de gérer les exceptions du moteur du binding. Il faut alors associer un ExceptionValidationRule au binding (voir http://msdn.microsoft.com/fr-fr/library/system.windows.controls.exceptionvalidationrule.aspx)
UpdateSourceTrigger	Timing de la mise à jour de la source (voir les exemples de l'article). Valeurs possible : Default, PropertyChanged, LostFocus et Explicit.
ValidatesOnDataError	Ajouté dans 3.5sp1 utilise un IDataErrorInfo au moment de la validation des données.
ValidateOnExceptions	Ajouté dans 3.5sp1 permet de traiter les exceptions du binding comme des violations des règles de validation.
ValidationRules	Collection de règles qui permettent de valider les valeurs généralement les saisies de l'utilisateur.
XPath	Requête XPath qui retourne la valeur source à utiliser lorsque la source est de type XML. La syntaxe peut être bien plus complexe que celle utilisée dans les exemples de cet article. Voir MSDN : http://msdn.microsoft.com/fr-fr/library/ms256471(VS.80).aspx

Conclusion

Le binding, cet inconnu… Présenté dans des dizaines de tutoriaux, et pourtant… Ce qui me semble essentiel n'est que très rarement traité : *comment s'en servir correctement et en éviter les pièges bien réels*.

Les tutoriaux et leurs démos sont toujours bien « lisses ». Tout fonctionne si on suit « la recette ». La réalité est bien plus nuancée simplement parce que l'écriture d'une application même de taille modeste n'est pas un long fleuve tranquille. Erreurs de conception qu'il faut corriger, changement de design de dernière minute, ajouts de fonctionnalités au fil de l'écriture, analyse trop succincte (quelle analyse au fait ? ils sont où les documents ?), etc, etc… Et dans cette situation précise qu'est la *réalité*, des tas de difficultés plus ou moins grandes apparaissent.

Mais on en parle que fort peu. Certainement parce que toute démo, tout tutor a avant tout valeur de propagande et que jamais une publicité ne vous parlera de façon relative du produit, il ne peut qu'être beau, facile à utiliser, pratique et faire de vous un héros au quotidien (Microsoft et Borland on même communiqué un temps sur cette image de super héros totalement ringarde à mon goût, heureusement le premier n'a pas insisté longtemps et le second est mort, ceci expliquant en partie cela ? ☺).

Développer professionnellement n'est pas une simple transcription d'une démo ou d'un cours d'école d'ingé. On est souvent plus proche d'Indiana Jones couvert de vase se débrouillant seul dans des marécages putrides et farcis de bêtes bizarres que d'un Tom Cruise tout propre agitant ses datagloves dernier modèle devant des écrans virtuels dans Minority Report !

Heureusement, tel un super héros je viens à votre secours ! (comme l'écriraient mes filles : LOL MDR !)

Sans rire, je vous mets au défi de trouver sur le Web, même en anglais, un article aussi complet sur le Binding…

StringFormat : une simplification trop peu utilisée

Mettre en page des éléments variables ou formatés en XAML peut parfois sembler fastidieux, c'est oublier qu'il existe des astuces de Binding comme le `StringFormat` qui simplifient beaucoup les choses…

De Silverlight à Windows Phone 8 et WinRT

Le principe est le même et les options sont semblables. Cet article s'applique ainsi à toutes les versions de XAML implémentant `StringFormat`. Le nombre de "saveurs" de XAML étant désormais important (WPF, WP7, WP8, Silverlight, WinRT… et chacune variant parfois selon l'historique de ses versions), il sera toujours intéressant de vérifier ponctuellement que telle ou telle option est disponible dans la version de XAML que vous utilisez, plutôt que de vous arracher les cheveux ou de me traiter de tous les noms derrière mon dos 

Le code à la Papa

Je vois souvent du code écrit "à la papa". Il s'agit d'un méfait plus répandu que le célèbre code Spaghetti (Cf. Le Retour du Spaghetti Vengeur) et donc ayant un pouvoir de nuisance presque aussi grand. S'il reste moins "grave" techniquement parlant, puisqu'il respecte en apparence le langage et les bonnes pratiques, il est insidieux et complique la maintenance, voire l'écriture du code, lorsqu'il n'oblige pas à écrire du code là où il en est nul intérêt ni besoin.

Un simple exemple : Afficher un texte du genre "*Bonjour machin*", où "*machin*" est un nom quelconque.

Ce qu'on voit souvent c'est un bricolage du genre :

```
<StackPanel Orientation="Horizontal">
    <TextBlock Text="Bonjour "/>
    <TextBlock Text="{Binding UserName}"/>
</StackPanel>
```

XAML 51 - Mauvais code Xaml

On remarque l'utilisation de deux `TextBlock`, un fixe, l'autre bindé à la source de données. Mais on note aussi la présence sournoise d'un `StackPanel` pour englober tout cela.

Autant d'objets en mémoire, autant de code écrit rend le tout indigeste.

Certes c'est "légal", c'est là toute l'astuce du code « à papa », il a l'air inoffensif et légitime… Son défaut principal ? En être resté à l'époque de la machine à vapeur de sa jeunesse…

StringFormat à la rescousse

Plutôt qu'une bonne grosse poignée de code bien lourde à digérer, on peut résumer en une seule ligne avec un seul objet le code ci-dessus :

```
<TextBlock Text="{Binding UserName, StringFormat='Bonjour {0}'}" />
```

XAML 52 - StringFormat

Forcément, c'est plus élégant… ça prend moins de place dans le fichier XAML, ça rend la lecture plus aisée, donc la maintenance plus sure et plus facile, et en mémoire un seul objet est créé au lieu de trois. Ce qui signifie aussi que la circulation des

évènements (*Bubbling* et *Tunnelling*, l'un ou les deux selon la saveur de XAML) s'en trouve fluidifiée en cas de gestion d'un clic, d'un mouvement de souris, etc.

Rien que des avantages donc. Et grâce à une option du Binding : `StringFormat`.

Il n'y a pas que le nom de l'utilisateur qui se prête ainsi au jeu des simplifications, de très nombreux cas se présentent où les données retournées par le Modèle ou le ViewModel ne se prêtent pas instantanément à une présentation agréable pour l'utilisateur. Imaginez la lecture du capteur de position GPS par exemple, si vous affichez la latitude sans prendre de précaution, des tonnes de chiffres après la virgule viendront s'étendre sur votre belle mise en page pour la ruiner (ce qui est vrai avec tous les `Double` ou même `Float`)…

J'ai vu des développeurs exposer de telles propriétés en `String` dans leurs ViewModels pour avoir la possibilité de les formater correctement. Que dire d'autre que "Beurk !"… Je suis partisan d'utiliser les ViewModel pour adapter les données à la Vue mais uniquement quand cette adaptation est impossible à faire directement dans l'UI ou qu'elle imposerait l'écriture d'un convertisseur de valeur (une solution trop lourde juste pour formater une donnée).

La mise en page c'est la mise en page, le code d'un ViewModel ou d'un Modèle ne doit en rien être adapté pour la mise en page. Les données elles-mêmes peuvent être présentées par le ViewModel dans un format plus pratique à gérer niveau UI, il ne faut pas être psychorigide et appliquer des règles « absolues » mais là c'est toute la fantaisie qu'on s'autorise. C'est à la Vue de faire la mise en page. Donc à XAML.

Ainsi, la fameuse latitude sera bien plus lisible si on restreint son affichage à, par exemple, 3 décimales, comme suit :

```
<TextBlock Text="{Binding MyGeoposition.Latitude,
                 StringFormat=\{0:F3\}}"
/>
```

XAML 53 - StringFormat paramétré

Vous percevez j'en suis certain l'intérêt.

Support multi langue

Le support de toutes les langues du monde est une tâche difficile, généralement un logiciel se limite à deux ou trois traductions possibles. Encore faut-il bien gérer

l'affichage des nombres, des dates et tout ce qui est en rapport avec la culture locale de l'utilisateur…

Pour un américain, une date sera lisible si elle est écrite ainsi ;

Tuesday, December, 11, 2012

Ce qui est ridicule pour un français qui préfère largement :

Mardi, 11 décembre 2012

Mais un espagnol n'y trouvera pas son compte, lui qui écrit :

martes, 11 de diciembre de 2012

Allez-vous vous amuser à prendre en compte tous ces cas particuliers ? Non, et vous le savez, l'une des force du framework .NET est de connaitre ces subtilités-là à votre place, loué soit-il.

Hélas XAML est un peu plus comme ses créateurs américains "*ah bon, tout le monde ne parle pas l'anglais en mâchant du chewing-gum, une pizza dans une main et un saut de popcorn dans l'autre ?*".

Donc XAML a dû sécher les cours de langues car hélas il a un petit problème de synchronisation avec la belle mécanique de localisation mise en œuvre dans le framework .NET.

Pour s'en sortir, rien de compliqué, mais encore faut-il le savoir.

Il suffit d'écrire dans le constructeur de la Vue le code suivant :

```
this.Language =
XmlLanguage.GetLanguage(Thread.CurrentThread.CurrentCulture.Name);
```

Code 34 - Correction du problème de localisation

Il est clair que cette petite synchronisation aurait dû être corrigée depuis longtemps, mais les équipes qui font XAML ont tellement été trimbalées par ci et par là à tout refaire à zéro comme pour WinRT que forcément l'éclatement du même code en différentes saveurs ne facilite pas la consolidation de l'existant. Quand on vous dit qu'il faut travailler proprement pour simplifier la maintenance, ce n'est pas juste pour vous embêter, vous avez ici la preuve éclatante des effets dévastateurs d'un manquement à cette rigueur…

Le mini langage dans le mini langage d'un langage

C'est un peu mon premier reproche quand j'ai abordé Xaml il y a déjà longtemps, je m'y suis fait, mais je reste toujours aussi critique sur certains choix que je trouve un peu gênant. Notamment cette impression de "poupées russes". Xaml est un langage. Le Binding est une astuce proposant son propre mini langage avec ses options, ses variantes, etc. Et `StringFormat` se place dans le Binding en offrant lui aussi ses petites variantes "perso" sans aucune véritable homogénéité avec le reste.

Mais avec le temps on s'y habitue sauf que malgré tout il faut garder une antisèche toujours planquée dans un coin pour ne rien oublier !

`StringFormat` ne se contente pas de placer une chaîne dans une autre avec un marqueur de position "{0}", il possède de nombreuses options bien pratiques !

Chaînes

Par exemple avec les chaînes il est possible d'indiquer un cadrage gauche ou droit dans un espace défini :

```
StringFormat=\{0,10\} donnera "   Bonjour"
StringFormat=\{0,-20\} donnera "Bonjour   "
```

XAML 54 - Exemples de formatage par StringFormat

Le premier `0` indique le paramètre, le chiffre après la virgule est le nombre d'espaces dans lequel le champ doit être justifié, positif pour un cadrage à droite, négatif dans l'autre cas.

Très honnêtement je déconseille fortement cette option, seules les secrétaires de plus de 60 ans qui ont connu les machines à écrire mécaniques s'amusent à cadrer du texte avec la barre d'espace… Sur les traitements de texte on utilise des tabulations, et avec XAML on utilise tout simplement une mise en page correcte !

Mais bon, cela doit pouvoir servir. Ça existe en tout cas.

Nombres

Les options concernant les nombres sont bien entendu plus intéressantes et surtout se justifient pleinement.

Pour un `double` qui possède la valeur `15695,2569` les effets des différents formatages seront les suivants :

```
StringFormat=\{0:C\} : 15 695,26 €

StringFormat=\{0:F\} : 15695,26

StringFormat=\{0:N\} : 15 695,26

StringFormat=\{0:E\} : 1,56952569E+004

StringFormat=\{0:P\} : 1 569 525,69%
```

XAML 55 - StringFormat et les nombres

En ajoutant un nombre derrière la lettre du format on peut indiquer le nombre de décimales voulues (de `0` à autant que cela a de sens). Le nombre est collé à la lettre. Par exemple `StringFormat=\{0:F3\}` pour un formatage de type `F` avec `3` décimales.

On note l'antislash pour "escaper" l'accolade américaine ouvrante et fermante, ce qui est obligé puisque ces accolades sont contenues à l'intérieur d'un Binding qui lui-même les utilise déjà pour autoriser sa propre présence en XAML… (On en revient au langage dans le langage dans le langage…).

Dates

`StringFormat` est bien utile aussi pour les dates, objet complexe dont les conventions d'écriture sont très nombreuses et possédant dans chaque culture ses propres variantes selon qu'on veuille faire court ou long.

Le principe est le même que pour les nombres sauf qu'on ne peut pas passer de chiffre après la lettre de format, puisqu'il n'y a pas de décimales…

Le plus simple des formatages est obtenu avec "`d`" minuscule. Il formate la date sans l'heure en mode compact "`1/1/2013`" par exemple.

La lettre "`D`" majuscule formate la date longue ("`Mardi 11 décembre 2012`").

Le petit "`f`" rajoute l'heure à "`D`" en format court ("`3:25`")

Le grand "`F`" fait comme le petit "`f`" mais l'heure est cette fois-ci en format long ("`16:12:30`")

Le petit "`g`" formate la date en mode court et l'heure en mode court.

Le grand "`G`" formate la date en mode court mais l'heure en mode long.

Le grand "`M`" formate le nom du mois et le jour "`15 Décembre`"

Le grand "`R`" format le nom du jour court, le nom du mois court, et l'heure longue en GMT ("`Ma, 1er Jan 2013 22:15:20 GMT`")

Bien entendu, il reste possible de fabriquer son propre format date et heure en utilisant les traditionnels "`hh` / `HH`, `mm`, …". Cela se justifie dans quelques cas. Hélas il est fréquent qu'ayant oublié les nombreuses variantes exposées ci-dessus on s'adonne à la facilité d'un formatage personnalisé dont les composantes sont plus logiques et faciles à retenir. Mais c'est mal. Si on utilise un formatage existant, il faut utiliser la lettre de format.

D'autres références

Vous trouverez les références complètes sur MSDN bien entendu.

Voici quelques adresses :

Le formatage composite

Le formatage standard des nombres

Le formatage personnalisé des nombres

Le formatage standard des dates

Le formatage personnalisé des dates

Conclusion

`StringFormat` n'est pas l'astuce du siècle, tout juste une façon de vous rappeler que XAML est un langage "à tiroir" et qu'il faut prendre le temps de bien connaitre toutes ses subtilités car la plupart font gagner du temps et rendent le code plus lisible et plus maintenable.

C'est aussi une façon de vous rappeler que le Binding est un langage à lui tout seul (cf. mon article en PDF + code Le Binding Xaml – Maîtriser sa syntaxe et éviter ses pièges près de 80 pages tout de même ! – intégré et mis à jour dans le présent livre PDF).

Un bon code n'est pas fait seulement de grandes théories, mais aussi de petites attentions…

L'Element Binding

L'Element Binding est une feature ajoutée à Silverlight 3 mais qui était déjà présente sous WPF et qu'on retrouve sous WinRT et Windows Phone.

L'Element Binding définit la capacité de lier les propriétés des objets entre eux sans passer par du code intermédiaire.

Pour simplifier prenons l'exemple d'un panneau d'information, par exemple un `Border` avec un texte à l'intérieur. Imaginons que l'utilisateur puisse régler l'opacité de cette fenêtre par le biais d'un `Slider`. (Ci-dessous l'application exemple en Silverlight que vous pouvez tester en live sur Dot.Blog en accédant à l'article par un clic sur le titre de ce chapitre du livre PDF).

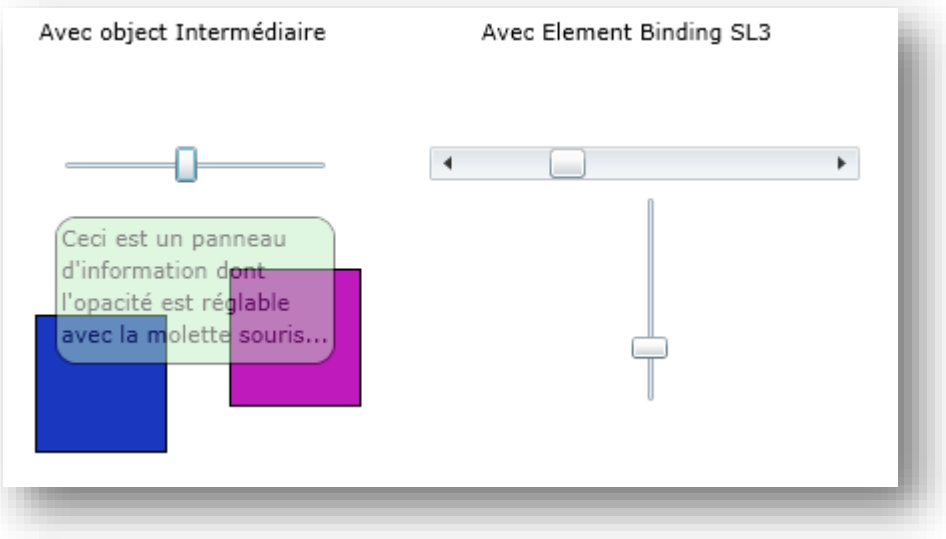


Figure 74 - Element binding en action

On place quelques éléments visuels sous le `Border` (ici des rectangles) afin de mieux voir l'effet du changement d'opacité.

Trois méthodes s'offre à nous pour régler le lien entre le `Slider` et l'opacité du `Border`. J'appellerai la première "*méthode à l'ancienne*", la seconde "*méthode du certifié*" et la troisième "*méthode XAML*". Les trois ont leur intérêt mais si vous êtes très pressé vous pouvez directement vous jeter sur la 3eme solution !

Méthode 1 dite « à l'ancienne »

Le développeur Win32 habitué aux MFC ou à des environnements comme Delphi ou Java aura comme réflexe immédiat d'aller chercher l'événement `ValueChanged` du `Slider` et de taper un code behind de ce type :

```
MonBorder.Opacity = MonSlider.Value ;
```

Ça a l'avantage d'être simple, efficace, de répondre (apparemment) au besoin et de recycler les vieilles méthodes de travail sans avoir à se poser de questions…

Qu'en dire ? C'est à l'ancienne, ça résume tout…

Méthode 2 dite « du certifié »

Ici nous avons affaire à un spécialiste. Son truc c'est la techno, suivre les guidelines et écrire un code qui suit tous les dogmes de l'instant est un plaisir intellectuel. Parfois ses solutions sont un peu complexes mais elles sont belles et à la pointe de la techno !

Conscient que la solution « à l'ancienne » a un petit problème (en dehors d'être trop simple pour être « belle », elle est one way, le slider modifie l'opacité du border mais l'inverse ne fonctionne pas) il va chercher une solution objet élégante répondant à l'ensemble des cas possibles couvrant ainsi le two way, considération technique trop technophile pour le développeur du cas précédent.

Ici forcément ça se complique. C'est techniquement et intellectuellement plus sexy que la méthode « à l'ancienne » mais cela réclame un effort de compréhension et de codage :

Il faut en fait créer un objet intermédiaire. Cet objet représente la valeur du `Slider` auquel il est lié lors de son instanciation. Quant à l'objet `Border`, sa propriété `Opacity` sera liée par Data Binding standard à l'objet valeur.

Voici le code de la classe de liaison :

```
public class ValueBinder : INotifyPropertyChanged
	{
		public event PropertyChangedEventHandler PropertyChanged;
		private Slider boundSlider;
		public ValueBinder(Slider origine)
		{
			boundSlider = origine;
		}

		public double Value
		{
			get { return boundSlider==null?0:boundSlider.Value; }
			set {
				if (PropertyChanged!=null)
					PropertyChanged(this,
					  new PropertyChangedEventArgs("Value"));
				boundSlider.Value = value;
			}
		}
	}
```

Code 35 - Un code correct mais pas utile

Cette classe, ou plutôt l'une de ses instances, servira à représenter la valeur courante du `Slider`. Ce dernier est lié à l'instance lors de la création de cette dernière (voir le constructeur de la classe `ValueBinder` ci-dessus).

Comment utiliser cette classe ?

La première chose est qu'il faut en créer une instance, cela peut se faire dans le code XAML ou bien dans le code behind de la façon suivante (dans le constructeur de la page par exemple) :

```
var valueBinder = new ValueBinder(slider);
```

Maintenant il suffit dans le code XAML de lier les deux objets à la valeur de la classe intermédiaire, par Data Binding :

Côté Slider, le code est :

```
<Slider x:Name="slider" …  Value="{Binding Value, Mode=TwoWay}"/>
```

Côté Border :

```
<Border x:Name="borderInfo" … Opacity="{Binding Value, Mode=TwoWay}" …>
```

Ne reste plus qu'à rendre visible l'objet intermédiaire par exemple en en faisant la valeur courante du `DataContext` de l'objet `LayoutRoot` :

```
LayoutRoot.DataContext = valueBinder;
```

Et voilà ! Ne reste plus qu'à compiler et à vous le plaisir de voir que l'opacité du `Border` change bien lorsque le `Slider` est déplacé.

La chaîne est la suivante : la modification de la position du `Thumb` entraîne dans le composant `Slider` la modification de la valeur de la propriété `Value`. Comme celle-ci est liée par Data Binding TwoWay à la propriété `Value` de l'objet intermédiaire cette propriété va se trouver modifiée dans le même temps. Comme le `Setter` de la propriété notifie le changement de valeur de la propriété (la classe implémente `INotifyPropertyChanged`) et comme la propriété `Opacity` du `Border` est elle aussi liée par Data Binding à la propriété `Value` de l'objet intermédiaire, le `Border` sera « prévenu » du changement de valeur et sa propriété `Opacity` sera immédiatement modifiée pour recevoir la valeur courante de `Value` de l'objet intermédiaire ! C'est pas fun tout ça ? (hmmm j'en vois un qui ne suit pas là bas au fond… !).

Vous allez me dire, TwoWay, on veut bien te croire mais on ne le voit pas là … Pour l'instant cela se comporte exactement comme la première solution, juste qu'il faut avoir un niveau de certifié pour comprendre…

Ce n'est pas faux. C'est pourquoi je vais maintenant ajouter un petit bout de code pour faire varier la valeur de la propriété `Opacity` du `Border`. Le plus simple est de gérer la roulette de la souris dans le `Border` :

```
<Border x:Name="borderInfo" …  MouseWheel="Border_MouseWheel"…>
```

Et dans le code behind :

```
private void Border_MouseWheel(object sender,
System.Windows.Input.MouseWheelEventArgs e)
            {
                  borderInfo.Opacity += e.Delta/1000d;
                  e.Handled = true;
            }
```

Il suffit maintenant de faire rouler la molette au-dessus du `Border` pour en changer l'opacité. Le TwoWay ? Regardez bien : le curseur du `Slider` avance ou recule tout seul… Pour éviter que la page HTML contenant le plugin Silverlight ne se mette à scroller il faut bien entendu indiquer que l'événement est géré (`Handled=true`) ce qui est fait dans le code ci-dessus et qui ne serait pas utile sous WPF par exemple.

Cette solution est élégante, complète mais complexe. Elle reste l'approche à préconiser dans tous les cas du même type car, on le voit bien, si la première solution est simple, elle n'est pas complète. Complexifier par plaisir est un mauvais réflexe, mais ne pas voir qu'une solution est trop simpliste est aussi un défaut qu'il faut fuir !

Bref, si nous revenions à l'époque de Silverlight 2 la solution présentée ici serait une bonne solution. Depuis Silverlight 3 et les autres variantes de XAML supportent le Binding direct entre éléments (comme WPF) et nous allons pouvoir faire plaisir à la fois au développeur du premier cas et à celui du second :

Méthode 3 dite « XAML »

Comment marier le reflexe (plutôt sain) du premier développeur de notre parabole de vouloir faire vite et simple avec l'exigence intellectuelle (tout aussi saine) du second qui implique de faire « complet » ?

En utilisant la méthode dite sobrement « XAML » car c'est tout simplement comme cela qu'il faut faire lorsqu'on connait XAML… Pas besoin d'inventer des solutions trop simples ou trop biscornues, XAML le fait, cela s'appelle l'Element Binding.

Pour illustrer l'Element Binding de façon simple, prenons une `Scrollbar` en mode horizontal et un `Slider` en mode vertical.

Et regardons le code XAML de leur déclaration :

```
<ScrollBar x:Name="scrollA" …
           Value="{Binding Value, ElementName=sliderB, Mode=TwoWay}"/>

<Slider x:Name="sliderB" …
        Value="{Binding Value, ElementName=scrollA, Mode=TwoWay}" />
```

XAML 56 - Element Binding

Et c'est tout ce qu'il y a à faire pour obtenir une solution complète, élégante, sans code behind et très facile à mettre en œuvre ! Merci XAML !

En début de billet vous pouvez jouer avec les deux dernières implémentations (je n'ai pas implémenté la méthode 1).

Vous pouvez aussi télécharger le code du projet (Blend 3 ou VS2008 minimum avec les extensions SL3) : elementBinding.zip (61,43 kb)

Contrainte de propriétés (Coercion)

La contrainte des valeurs est un mécanisme essentiel permettant à la valeur d'une propriété de rester confiner dans des bornes fixées par l'objet ou par l'utilisateur. C'est en réalité un des éléments de base permettant de respecter le paradigme de la programmation objet : l'encapsulation qui veut qu'un objet se protège de toute action externe pouvant le déstabiliser. Hélas, WinRT comme Silverlight n'offrent pas les mécanismes que WPF propose pour les propriétés de dépendances. Nous allons voir comment régler cet épineux problème.

La Contrainte de valeur (value coercion)

Le principe est simple : quand une propriété ne peut pas prendre n'importe quelle valeur il faut mettre en place un mécanisme de protection pour valider et maintenir la valeur dans les bornes acceptées par le contexte. Validation et contrainte sont d'ailleurs deux choses assez différentes qui se traitent parfois ensemble mais qui doivent être comprises comme deux actions distinctes fonctionnellement parlant.

Valider une valeur consiste à vérifier qu'elle appartient à l'ensemble des valeurs "légales". Par exemple un `TextBox` peut être utilisé pour saisir un entier. La validation consiste ici à vérifier que ce qui est tapé est bien un entier et à rejeter la saisie dans le cas contraire. Le Framework propose certaines validations automatiques (ou explicites) et met à disposition du développeur d'autres mécanismes comme des

attributs pouvant décorer les propriétés d'un `DataContract` ou des interfaces spécifiquement créées pour valider le contenu d'une propriété ou l'ensemble des propriétés d'un objet.

La contrainte de valeur est plus limitative, elle considère que la valeur qui lui est donnée appartient bien déjà à l'ensemble des valeurs légales (elle a été validée), ce qui lui importe c'est uniquement de savoir **si la valeur est "acceptable" à l'instant précis où elle est changée**, c'est à dire si elle est conforme à ce que le contexte global de l'objet considéré peut accepter sans que cela ne le déstabilise.

Il ne faut pas voir ici un absolu, cette séparation validation / contrainte est récente et la gestion des contraintes, historiquement, a longtemps été considérée comme faisant partie des validations. La différence faite ici repose sur celle qui est mise en place au sein du Framework principalement sous WPF (et que nous allons mettre en œuvre pour Silverlight ou WinRT).

Il faut bien comprendre que cette problématique émerge surtout pour les objets visuels XAML, les contrôles ou User Control, car ils définissent le plus souvent des propriétés spéciales dites « propriétés de dépendance » (que nous avons étudiées dans un chapitre précédent du présent livre PDF).

La validation et la contrainte des valeurs sur des propriétés classiques dites « CLR » (en référence au Common Language Runtime du framework .NET) s'effectuent dans le *setter* de la propriété ce qui ne pose aucune difficulté. Le fonctionnement des propriétés de dépendance rend ces opérations plus délicates d'où la nécessité d'une stratégie différente.

Coercition et propriétés CLR

On appelle *propriétés CLR* les propriétés déclarées en suivant la syntaxe classique du langage dans lequel on développe (C#, VB.NET...). En C# les propriétés s'écrivent sous la forme d'une déclaration de type et de nom agrémentée d'un *getter* et d'un *setter* comme le montre l'exemple ci-dessous:

```
private int age;
int Age { get { return age;} set { age=value;}}
```

Dans ce petit morceau de code on utilise "`age`" qui est un champ entier privé qu'on appelle un "**backing field**". Il existe des variantes déclaratives que nous n'aborderons pas ici car tel n'est pas le sujet de cet article.

La validation telle qu'expliquée en début d'article n'a pas de sens dans un tel contexte puisque C# est un langage dit fortement typé et qu'il repose sur un Framework ayant la même caractéristique. De fait le setter de la propriété `Age` ne recevra jamais rien d'autre qu'un entier, tout code voulant lui envoyer une valeur d'un autre type sera sanctionné à la compilation. La propriété ne peut ici recevoir qu'une valeur licite (mais ce n'est pas forcément toujours le cas).

En revanche ici la *coercition* de la valeur a tout son intérêt. S'agissant d'un âge, et en supposant qu'il n'existe aucune autre contrainte fixée par le contexte ou le cahier des charges, il semble raisonnable d'interdire les valeurs inférieures à 1 et celles supérieures à disons 150 ans (attention à ne pas recréer le bug de l'An 2000… Ici nous allons interdire les valeurs hors bornes. Si nous limitons à 100 ans nous ne pourrons pas saisir l'âge du doyen de l'humanité, si nous posons 120 comme borne notre logiciel posera un problème dans 10 ans peut-être quand ce record sera battu…).

Pour une propriété CLR l'implémentation de la coercition est évidente, le setter (qui masque une méthode `Set_<nom de la propriété>`) est justement étudié pour cela - entre autres. Ainsi, notre code exemple deviendra:

```
int Age
      { get { return age;}
        set {
              if (age<1 || age>150) return;
              age=value;
            }
      }
```

Code 36 - Une propriété CLR avec contrôle des valeurs

Il s'agit d'une coercition simple, dite par exclusion. On pourrait en débattre longtemps mais *stricto sensu* la valeur n'est pas vraiment contrainte ici; on ne fait que *rejeter* les valeurs interdites. La coercition sous-entend qu'on "bloque" la valeur soumise tant qu'elle est interdite et que la propriété pourra reprendre toute seule la valeur saisie si jamais elle redevient valide. Le mécanisme mis en place dans WPF est d'une grande subtilité.

> *Je traduis ici Coercion par Coercition, ce qui semble raisonnable. Toutefois le verbe lui-même est très peu utilisé, et s'utilise principalement sous la forme d'un adjectif dans l'expression "mesure coercitive" qui a un sens négatif. On pourrait utiliser le verbe "contraindre" et le mot "contrainte" à la place de "Coerce" et "Coercion" mais j'ai préféré conserver la traduction la plus proche, pour une fois qu'il existe un mot français de même sens ou presque...*

Un meilleur exemple pour comprendre cette nuance subtile mais essentielle se trouve dans tous les objets proposant des propriétés de type "mini/maxi" bornant les valeurs possibles d'une troisième propriété (généralement appelée `Value` mais tout dépend de l'objet et de son utilité).

Le contrôle `Slider` illustre parfaitement ce pattern classique. On retrouve les propriétés `Minimum` et `Maximum` ainsi qu'une propriété `Value` qui ne peut prendre qu'une valeur comprise dans les bornes définies par les précédentes.

Imaginons que nous utilisions un `Slider` ayant pour bornes 0 et 10. Sa `Value` étant initialisée à `6`. Changeons le `Maximum` pour le descendre à `4`. En toute logique le changement de valeur de la propriété `Maximum`, en dehors de ses propres contrôles, doit déclencher ceux de la propriété `Value` pour s'assurer qu'elle reste bien confinée dans les bornes définies, ce qui n'est ici plus le cas.

Dans le modèle de propriété CLR la coercition s'exercera dans le setter de la propriété Maximum avec un code du type :

```
// ... setter de Maximum
  maximum = value;
  if (value>maximum) value=maximum;
  // appel de propertychanged de maximum ET de value
// ... suite
```

C'est simple et efficace. Cela fonctionne comme dans l'exemple de la propriété `Age` plus haut, sauf que la coercition est effectuée dans le *setter* d'une autre propriété que celle qui est "bornée". Ici, le changement de `Maximum` à la valeur `4` entraînera le passage de la propriété `Value` à `4` aussi puisque sa valeur courante (`6`) se trouve désormais en dehors des bornes. Bien entendu cela n'exclue pas le besoin d'effectuer un test similaire dans le *setter* de la propriété `Minimum` pas plus qu'il ne faut oublier d'implémenter de même type dans le *setter* dans la propriété `Value` elle-même.

Ce modèle est tellement simple que si nous changeons à nouveau `Maximum` pour mettre sa valeur à `6`, la propriété `Value` restera à `4`. Il n'y a en effet aucun effet "mémoire". Pourtant si la valeur était à `6` et que seuls sont intervenus les deux changements indiqués de `Maximum` (vers `4` puis vers `6`), la propriété `Value` devrait reprendre sa valeur `6`, à nouveau valide et inchangée car sa modification n'a été qu'interne et non pas effectuée sous l'impulsion d'une action ou volonté de l'utilisateur... En effet, puisque l'utilisateur n'a pas modifié volontairement la valeur de `Value` directement, elle ne devrait pas être modifiée définitivement par la coercition que nous avons appliquée qui devrait se comporter comme **une force de compression**, qui **une fois relâchée laisse la valeur « s'étirer » à nouveau**.

Or, avec les propriétés CLR, sauf à développer un code complexe pour chaque propriété, un tel comportement n'existe pas par défaut.

Coercition et propriétés de dépendances

XAML et ses possibilités nombreuses (tel que le **Binding**) a obligé les concepteurs du Framework à réfléchir sur d'autres façons plus sophistiquées de gérer les valeurs des propriétés. Cela a débouché sur la mise en place du *moteur des propriétés de dépendances* (et des propriétés attachées - ou jointes). Une propriété de dépendance (DP : *dependency property*), comprendre "interdépendante", accepte simultanément

plusieurs valeurs qui se trouvent à des niveaux de priorité différents. Par exemple une animation pourra imposer une série de valeurs dans le temps à une propriété, mais lorsque l'animation s'arrêtera la propriété reprendra la valeur qu'elle avait à l'origine, avant le lancement de l'animation. Dans mon article "Le Binding Xaml, sa syntaxe, éviter ses pièges…" (voir dans ce livre pour la version mise à jour) ainsi que dans celui portant sur les propriétés de dépendances "Les propriétés de dépendance et les propriétés jointes" j'ai longuement expliqué ces nuances et j'y renvoie le lecteur qui souhaite approfondir ces notions.

Pour ce qui nous intéresse ici rappelons ainsi qu'une propriété de dépendance se déclare de façon spécifique et qu'elle est généralement "doublée" par une propriété CLR de même nom. Les mécanismes en place pour enregistrer et gérer une propriété de dépendance ne sont pas implémentés dans le langage lui-même mais dans le Framework. De fait, une propriété de dépendance n'est pas "visible" directement pour du code classique. D'où la nécessité de déclarer une propriété CLR qui ne fait que dialoguer avec la propriété de dépendance afin de rendre celle-ci plus facilement accessible au travers du code et conserver une cohérence de nommage et d'accès à la valeur de la propriété. Ci-dessous un exemple simple de déclaration d'une DP avec sa propriété CLR :

```
public class MyStateControl : ButtonBase
{
  public MyStateControl() : basea() { }
  public Boolean State
  {
    get { return (Boolean)this.GetValue(StateProperty); }
    set { this.SetValue(StateProperty, value); }
  }
  public static readonly DependencyProperty StateProperty =
                          DependencyProperty.Register(
                                "State", typeof(Boolean),
                                typeof(MyStateControl),
                                new PropertyMetadata(false));
}
```

Code 37 - Définition d'une propriété de dépendance

> *On ne le dira jamais assez : La propriété CLR déclarée avec une propriété de dépendance est* ***optionnelle****. Elle ne sert qu'à donner accès à la propriété dans le code C#, XAML voit la propriété de dépendance sans cet artifice. De plus, et tout aussi essentiel, le* getter *et le* setter *de la propriété CLR « compagnon » ne doivent JAMAIS rien contenir d'autre que l'appel à* GetValue *et* Setvalue *respectivement de la propriété de dépendance. Tout code de validation, de test ou autre ne sera JAMAIS appelé lorsque les modifications de la propriété de dépendance sont faites depuis XAML (Binding notamment, saisie directe de l'utilisateur…).*

Les propriétés de dépendances gèrent plusieurs couches de valeurs comme nous l'avons dit. Il existe donc déjà un effet "mémoire" comme celui évoqué plus haut. Il est donc naturel que la coercition s'exerce de façon complète et non de façon simpliste comme pour les propriétés CLR. Dans l'exemple du `Slider` que nous avons utilisé plus haut le passage du `Maximum` de `10` à `4` puis de `4` à `6` doit faire varier dans un premier temps la propriété `Value` à `4` puis à `6`, sa valeur originale *redevenue licite*.

Pour rendre cela possible il faut considérer qu'il existe en réalité deux valeurs pour une propriété : sa valeur réelle et sa valeur bornée. En fait le mécanisme fait intervenir une méthode pour en assurer la coercition. Cette méthode est un **Callback** passé lors de l'enregistrement de la propriété de dépendance au travers d'une structure de **métadonnées**. Le reste, notamment l'effet "mémoire", est géré par le moteur des propriétés de dépendance du Framework car il n'y a techniquement pas besoin de conserver une copie de la valeur non bornée. La coercition doit ainsi se comprendre plutôt comme un filtre en lecture, la valeur est lue au travers du filtre ce qui donne l'impression qu'il y a deux valeurs, la « vraie » stockée dans l'objet, la « bornée » retournée par la coercition, supprimons le filtre et la « vraie » valeur semble « revenir » par magie. En réalité elle a toujours été là et c'est l'effet du filtre qui donne l'impression qu'elle a pris d'autres valeurs.

WinRT et Silverlight ne supportent pas la coercition

Hélas pour nous, et de façon assez incompréhensible il faut le dire, le Framework Silverlight pas plus que celui de WinRT ne supporte pas la même structure de métadonnées pour les DP que WPF, il manque le callback de coercition. Certes il

fallait bien couper des fonctionnalités pour passer d'un framework complet de plusieurs dizaines de méga au plugin Silverlight, et ce découpage chirurgical a été majoritairement effectué de main de maître avec une grande subtilité, mais il y a des ratés... On comprend beaucoup moins pourquoi WinRT est si restrictif sur de très nombreux points alors que le framework WinRT est gigantesque et qu'ici on n'était pas à quelques octets près. La suppression du callback de coercition dans Silverlight et WinRT interdit tout bonnement à un objet de type `Slider` de pouvoir borner les valeurs qui lui sont passées ce qui fait un peu désordre (car cela brise l'un des piliers du paradigme objet, l'encapsulation). Ces objets seraient de purs objets C# le problème ne se poserait, c'est leur nature XAML qui oblige la présence de DP qui créée ce problème. Et l'UI est un élément essentiel de l'UX. Comment Microsoft a-t-il pu sacrifier des éléments aussi essentiels notamment sous WinRT reste pour moi un mystère.

Les mauvaises réponses au problème

Vous allez me dire que ce n'est pas vrai, que le `Slider` de Silverlight par exemple fonctionne parfaitement et gère bien la coercition de la propriété `Value`. Prenez un outil comme Reflector et décompilez le code de la classe `Slider`... vous verrez alors la complexité de la solution mise en œuvre pour simuler la coercition !

Microsoft pouvait certes ponctuellement déléguer certains de ces développeurs pour régler le problème dans une seule propriété d'un seul composant, mais la méthode choisie est bien trop complexe pour être industrialisable dans un projet mettant en œuvre plusieurs propriétés de ce type. D'ailleurs Microsoft ne communique pas sur cette « solution » qui n'est documentée nulle part sauf à utiliser un décompilateur et à étudier le code original du Framework Silverlight...

La réponse de Microsoft dans le composant `Slider` est donc une mauvaise réponse au problème car elle n'est pas réutilisable (non industrialisable) et non documentée. De plus, si nous y regardons de plus près, le comportement de la propriété `Value` n'est pas exactement celui qu'il devrait être, il y a un léger bogue dans ce code très complexe... L'application de démo suivante montre le problème.

Pour bien comprendre il faut savoir que la valeur `Minimum` est considérée comme un point de repère car il faut bien choisir une règle au risque de se retrouver dans une situation ingérable d'un point de vue logique. Par exemple si on augmente le `Minimum`, il finira par « pousser » la `Value` jusqu'à l'amener en butée contre le `Maximum`. Et si on continue ce mouvement le `Minimum` entraînera `Value` et `Maximum` vers la droite. Si on maintenant on prend la situation inverse et que l'on diminue le `Maximum`, ce dernier *poussera* bien `Value` qui sera bien amenée en butée contre le `Minimum` mais si on continue le mouvement on sera bloqué : le `Minimum` ne peut pas être descendu en le « poussant » avec le `Maximum`.

C'est arbitraire, les concepteurs du Framework aurait pu prendre la logique contraire (faire du `Maximum` le point de repère immuable). En revanche il serait difficile de considérer `Minimum` et `Maximum` comme des points fixes tout comme n'en considérer aucun comme étant le point fixe. Réfléchissez à la mise en œuvre logique de l'algorithme et vous comprendrez mieux j'en suis certain (c'est un peu prise de tête mais l'exercice est vraiment intéressant, faites-le vraiment !).

Dans l'exemple ci-dessous (*accédez au billet original sur Dot.Blog en cliquant sur le titre du billet dans ce PDF pour jouer avec l'exemple Silverlight*) regardez aussi le comportement de la valeur lorsqu'elle est « compressée » par la monté du minimum ou la descente du maximum puis lorsque que cette « pression » est relâchée : la valeur reprend progressivement sa valeur originale mais avec des « sautes ». C'est le bogue évoqué plus haut.

Figure 75 - Exemple de coercion de valeurs

Quelques explications : ce qui est montré est le comportement des trois propriétés `Maximum`, `Minimum` et `Value` et principalement le mécanisme de coercition à l'œuvre lorsqu'on ne touche pas à `Value` et que l'on augmente ou diminue à l'extrême la valeur d'une des bornes puis qu'on la remplace à sa valeur de départ. Le `Slider` qui est "testé" en quelque sorte par cette démo se trouve tout à droite. Il ne ressemble plus à un `Slider` mais à rectangle de couleur affichant la valeur courante de `Value`. C'est un `Slider` tout à fait "normal" qui a été templaté pour l'occasion. Nous testons son code et non son look. Les trois `Slider` horizontaux permettent de *matérialiser les trois valeurs testées* (maxi, mini et valeur). Chacun d'eux est bindé à la propriété correspondante du `Slider` templaté et reflètent donc son état courant. Comme ces bindings sont en *two-way*, on peut agir sur les `Slider` pour modifier l'état du `Slider` testé, ce qui, en retour, peut faire varier la position des `Slider` horizontaux qui réagissent aux changements d'états des propriétés auxquelles ils sont liés. La procédure de test consiste donc ici à augmenter le `Minimum` jusqu'à l'amener en butée à droite, puis à le refaire descendre et observer le comportement du `Slider Value` durant ces opérations (ainsi que celui de `Maximum`). La deuxième étape consiste à refaire la même chose avec `Maximum` (en le poussant en butée à gauche cette fois-ci). Vous noterez que lorsque vous remettez progressivement le maximum ou le minimum à leur valeur d'origine, `Value` va elle aussi descendre ou monter pour reprendre progressivement sa position d'origine. Dans l'implémentation de la coercition du `Slider` de Silverlight il y a un bogue qui fait que `Value` n'a pas un déplacement graduel et continue, elle effectue des bonds. Je n'entrerai pas ici dans les explications de ce bogue qui dépasse le cadre de l'article.

Une autre mauvaise réponse classique à ce problème consiste à modifier le *setter* de la propriété CLR pour implémenter la coercition.

Effectivement lorsqu'on modifiera la valeur *par code*, puisque ce dernier va généralement passer par la propriété CLR, la valeur sera bien bornée puisque passant par le *setter* de la propriété (bornée mais pas contrainte avec mémoire de la valeur saisie, c'est l'exemple de la propriété `Age` en début d'article).

Hélas c'est bien mal comprendre comment les DP fonctionnent ! Seul le code (C# ou autre) peut voir la propriété CLR, lorsque le changement de valeur s'effectue par un mécanisme XAML tel que le Binding par exemple, **la propriété CLR est totalement ignorée** et le code de coercition avec... la propriété CLR déclarée en doublon d'une DP n'est là que pour en simplifier l'accès, *son getter et son setter ne doivent rien*

contenir d'autres que les appels à Set/GetValue de la DP. C'est donc une (très) mauvaise réponse au problème, pire que celle de MS dans le `Slider` car là, cela ne fonctionne même pas totalement... Cette mauvaise solution pose d'ailleurs, et en plus, les mêmes problèmes que la suivante.

La troisième façon classique de se tromper devant ce problème épineux consiste à utiliser le *callback* de changement de valeur de la propriété de dépendance qui lui existe aussi bien sous Silverlight et WinRT que sous WPF. Erreur grossière... En effet, dans le code en question nous sommes dans un "*changed*" et non pas un "*changing*". Les règles de nommage dans le Framework ont fait l'objet d'un long travail de la part de l'équipe qui l'a écrit. Je vous conseille d'ailleurs la lecture de "*Framework Design Guidelines. Conventions, Idioms, and Patterns for Reusable .NET Libraries*" écrit par ceux qui ont fait le Framework .NET et qui expliquent comment et pourquoi les choses sont ce qu'elles sont. Les créateurs du Framework y avouent aussi leurs mauvaises idées, leurs échecs, ce qu'ils auraient dû faire autrement, c'est un retour sur expérience incroyablement riche et instructif (mais en anglais...). Bref l'utilisation du passé dans les noms d'événements est une norme indiquant que ce qui est notifié *à déjà eu lieu* là où le présent indique une action en cours pouvant éventuellement être annulée. "*changed*"' implique donc que la valeur est déjà enregistrée par la DP, l'événement n'est là que pour nous permettre d'en prendre note si cela nous intéresse.

Implémenter la coercition à cet endroit aura un premier effet négatif : modifier la valeur dans le *changed* fera entrer le code dans un mode récursif fatal. Mieux vaut éviter ce genre de montage risqué qui fait pédaler dans le vide le logiciel ce qui, au fil des erreurs de ce type le rendra peu réactif (en dehors d'autres ennuis comme un plantage total). Mais ce n'est pas tout... Pour une DP un Binding est au niveau "*valeur locale*" (Cf. mon article sur les DP) et une valeur fixée par code est aussi considérée comme une valeur locale. Vous commencez à comprendre le malaise ? Non ? ... En fixant par code une valeur locale le moteur de DP va effacer et remplacer la valeur locale en cours puisque sur un niveau de priorité donné d'une DP il ne peut y avoir qu'une seule valeur. Moralité *si un Binding était présent il sera perdu définitivement* ! Impossible de le retrouver sauf à relancer le logiciel ou bien à implémenter un code redondant recréant le Binding original. Implémenter la coercition de valeur dans le callback *changed* est donc une erreur qui débouche sur des bogues graves et sur la perte des Bindings en place.

Une solution ?

J'en ai vu quelques-unes, toutes sont plus complexes les unes que les autres et en réalité aucune ne fonctionne correctement, ce sont des bricolages plus ou moins sophistiqués. Et toutes ont le même défaut que l'implémentation de Microsoft dans le `Slider` : être trop complexes et totalement ingérables sur de nombreuses propriétés (et présentant le plus souvent des écueils du même type).

J'ai aussi beaucoup réfléchi à ce problème car rien ne me choque plus que de ne pas pouvoir développer un contrôle capable de protéger ses propriétés, c'est-à-dire renoncer au 3ème millénaire et en utilisant les outils à la pointe du progrès à toute une partie de ce qui fait les *bases in-négociables de la programmation objet*. Mes réflexions m'ont amené à tester divers modèles plus ou moins satisfaisants. Au final on s'aperçoit que la seule vraie solution est de simuler la fonctionnalité manquante, notamment en modifiant les métadonnées des propriétés de dépendance de Silverlight ou WinRT pour y ajouter le fameux *Callback* manquant (et réussir à interférer avec le moteur pour prendre en compte ce nouveau Callback, ce qui n'est pas une mince affaire).

Mais on ne « tripatouille » pas les méandres du moteur de DP XAML comme ça ! Il s'agit d'un exercice de style assez complexe. De plus, il semble souhaitable d'adopter une solution compatible avec WPF, c'est-à-dire capable de détecter sous quel environnement elle est compilée pour exploiter les facilités de WPF et les simuler sous Silverlight et WinRT. Produire un code portable WPF/Silverlight/WinRT est un objectif qu'on ne doit jamais perdre de vue. Les impératifs du cross-plateforme dont j'ai parlé souvent ici en étant parfois mal compris par certains lecteurs est une réalité avérée désormais. Même en restant « pur Microsoft » il faut aujourd'hui faire un code portable et cross-plateforme d'abord pour le protéger des humeurs et changements inopinés dans les stratégies parfois confuses de Microsoft et ensuite pour s'assurer que le code pourra atteindre l'utilisateur quels que soient son OS. Microsoft aime se moquer de Google et Android et de la « fragmentation » de ce dernier. C'est oublier que Windows Phone 7.x a été totalement deprecated du jour au lendemain, que Windows Phone 8 n'est pas tout à fait compatible avec le WinRT de Surface ni avec celui du PC, qu'il existe un WinRT ARM, un autre pour Intel, et que le bureau classique Windows 7 est majoritaire chez les utilisateurs… Un foisonnement d'OS là où Microsoft voulait vendre de l'unité avec Windows 8 et WinRT il faut dire que cela fait

désordre. Julie Larson Green qui remplace Sinofsky avec aussi peu de talent vient de s'en rendre compte et a annoncé (novembre 2013) qu'à terme (lequel ?) Microsoft allait fusionner tous ces OS et qu'il n'en resterait que deux (lesquels ?). Bref, si vous tenez à ne pas tout redévelopper au gré des humeurs de la Direction de Microsoft (qui en plus va être changée d'ici l'été 2014 avec le départ annoncé de Ballmer), vous avez intérêt à écrire du code portable !

Malgré les différences dans les Frameworks il est bien plus intelligent de se forcer un peu pour écrire un contrôle portable une fois (et maintenable au sein d'un seul et même code) que de s'apercevoir qu'il faut le récréer totalement (et avoir deux codes à maintenir) pour l'utiliser dans l'un ou l'autre des environnements…

Toujours à la recherche d'une solution « propre » à ce gros problème je fouine régulièrement sur le Web pour voir si quelques solutions nouvelles sont apparues. Comme je le disais plus haut je déchante assez vite. Mais dernièrement j'ai découvert un code (récent) qui, s'il a lui aussi une certaine complexité (puisqu'il faut suppléer une fonction de base du Framework) répond à toutes les exigences : cohérence du fonctionnement, simulation exacte du comportement attendu, portabilité WPF/Silverlight /WinRT et simplicité d'utilisation.

> *On notera que la portabilité WinRT est théorique, c'est-à-dire qu'à la vue du code rien ne semble s'y opposer. A l'heure où j'écris ces lignes je n'ai pas eu le temps de refaire l'exercice sous WinRT pour m'en assurer. Le lecteur est averti… Et si l'un d'entre vous effectue ce test avant que je ne me décide à le faire, ses commentaires seront les bienvenus sur Dot.Blog !*

C'est cette solution, plus aboutie que mes propres essais, que je vais vous présenter maintenant.

Pourquoi alors avoir terminé le titre de ce paragraphe par un point d'interrogation plutôt qu'un point d'exclamation « Une Solution ? »… C'est que tant que Microsoft n'aura pas implémenté cette fonction essentielle, tout ce qu'on pourra faire pour la suppléer ne sera que bricolage, plus ou moins savant, plus ou moins pratique. La solution que je vous propose répond au besoin, reste assez simple à mettre en œuvre car l'auteur fournit des snippets pour écrire le code automatiquement, mais cela n'est

pas une vraie solution définitive. Toutefois elle fonctionne et permet même de régler le bogue visible dans le `Slider` de SL. C'est déjà pas mal et c'est pour cela que je vous la présente.

Mais pour mieux comprendre tout ceci je vous propose de "jouer" avec le même exemple SL que celui présenté plus haut mais réécrit en utilisant le code de coercition de « Dr WPF », c'est ainsi qu'il se fait appeler sur son blog (et impossible d'en savoir plus sur ce mystérieux personnage puisque même le nom de domaine de son site est déposé via tiers d'anonymisation ! Si cela se trouve il travaille chez Adobe ou Apple et ne veut surtout pas être démasqué ☺).

Testez les mêmes mouvements des trois `Slider` pour voir la différence de traitement (rendez-vous sur la page de l'article original sur Dot.Blog, vous trouverez les exemples live sous Silverlight directement utilisables en ligne).

Le Fonctionnement

Il s'agit d'un code très subtil dont les mécanismes sont un peu longs à disséquer en totalité. Si vous avez un niveau de compréhension assez élevé de C# et du Framework, alors il est préférable d'étudier le code par vous-même. Et si vous n'avez pas ce niveau mes explications vous embrouilleraient certainement encore plus…

Mais de très haut, car il faut bien en dire quelque chose sinon cela serait vraiment frustrant, le principe repose sur l'idée que j'ai déjà évoquée dans ce billet : remplacer la classe de métadonnées permettant de définir une DP par une classe plus complète intégrant le callback de coercition. Le tout en mimant la syntaxe de WPF pour la portabilité.

Bien entendu cela n'est pas réellement suffisant puisque Silverlight ou WinRT ne sauraient quoi faire des informations supplémentaires. Donc la librairie initialise une sorte de hook qui lui permet d'avoir la main pour traiter le callback. Le cheminement réel est un poil plus compliqué que cela, regardez le code, il n'est pas très gros, la difficulté réside dans son niveau d'astuce plus que dans sa taille…

La solution est « clean » elle ne fait aucune entorse ni « bricolage » coupable qui pourrait poser des problèmes. Toutefois je ne saurais vous offrir plus de garanties que l'auteur lui-même, c'est-à-dire en de tels cas : aucune (voir la licence d'utilisation).

Le code

Vous trouverez le code à télécharger sous deux formes, un projet de démonstration (les trois `Slider` version corrigée) ainsi qu'un ensemble de snippets à installer dans Visual Studio. Ces dernières contiennent à la fois le code de la classe principale (mais vous pouvez tout simplement reprendre le fichier qui se trouve dans le code de la démonstration) et de nombreuses variantes de déclaration de DP, ce qui est beaucoup plus intéressant puisque cela évite une frappe fastidieuse tout en gérant (ou non, c'est au choix) le fameux callback de coercition.

Le billet présentant l'ensemble se trouve ici : http://drwpf.com/blog/2010/05/05/value-coercion-for-the-masses/

Il est accompagné d'une vidéo d'un quart d'heure qui illustre bien la solution proposée (attention, c'est en anglais et le son est tellement compressé que la voix de robot qui en résulte peut être difficile à comprendre si vous n'êtes pas réellement *fluent* en English…). Cela n'est certainement pas le fruit du hasard, Dr WPF ne semble pas souhaiter pas qu'on reconnaisse sa voix non plus...

Vous trouverez aussi le billet suivant : http://drwpf.com/blog/2010/05/08/metadata-options-for-silverlight/ qui dévoile un peu plus les dessous de la solution adoptée (notamment dans le support des flags d'options utilisés pour la déclaration des métadonnées).

Pour ceux qui ne comprennent vraiment pas assez l'anglais pour s'y retrouver voici le lien direct de téléchargement de la démo : http://drwpf.com/blog/Portals/0/Samples/CoercionDemo.zip et celui de la librairie de snippets (exécuter le fichier, il y a un expert d'installation assez simple à comprendre qui se lance) : http://drwpf.com/blog/Portals/0/Reference/DrWPFSnippets.zip

Conclusion

Il est grand temps que l'équipe XAML nous fournisse une vraie solution (elle existe, c'est celle de WPF) pour nous éviter d'avoir recours à de telles extrémités. D'un autre côté, tant que cette solution intégrée au Framework n'existe pas il est hors de question de concevoir des contrôles qui ne sont pas capable de répondre aux exigences les plus élémentaires de l'encapsulation. Alors remercions le mystérieux Dr WPF pour ce code de très bon niveau qui apporte une solution viable et industrialisable tout en étant compatible avec le code original de WPF. Il est étrange

que Dr WPF se cache à ce point et cherche à masquer son identité jusqu'au dépôt de nom de domaine de site, jusqu'à maquiller sa voix sur la vidéo. Mais c'est son choix.

Restons sur ce mystère et surtout le plaisir de disposer d'une solution exploitable à cet énorme problème que pose la coercition de valeur sous Silverlight et les autres profils XAML moins puissants que WPF !

Custom Control vs UserControl

La différence est importante entre `Custom Control` et `UserControl`, et en même temps il est souvent difficile de la comprendre ! Quand créer l'un ou l'autre est une question récurrente. Comment choisir entre l'un ou l'autre dans ses développements ?

A l'origine…

A l'origine le développeur génial créa Visual Basic. Un EDI simple, fondé sur ce qu'on appellera plus tard le "RAD" (*Rapid Application Development*). Dans ce nouveau monde incroyable on cassait le modèle de programmation triste qui prédominait alors. Au lieu de tout faire en code sous un éditeur de texte, comme une sténodactylo, on pouvait enfin concevoir des applications Windows en posant des "briques visuelles", des "*composants*". Boutons, checkbox, etc, devenaient des *objets visuels* pour le développeur aussi !

> *Visual Basic a fait mieux qu'un mage, un dieu ou un guru :* ***il a rendu la vue aux développeurs !*** *Et bien que je n'aie jamais été un fan de cet outil à cause de son langage – je lui préférais Delphi et son Pascal à l'époque – il est important de rendre à César ce qui lui revient.*

Aveugles tâtonnant "au jugé" dans leurs éditeurs de texte, les programmeurs ont d'un seul coup vu la lumière et sont devenus des développeurs ! Enfin sous leurs yeux écarquillés le bouton était réellement un bouton qu'il suffisait de placer visuellement là où on voulait qu'il soit ! *On voyait enfin ce que l'utilisateur verrait plus tard.*

Nous sommes dans les années 80, le génial inventeur s'appelle Alan Cooper. Il montre son projet, "*Tripod*", à un certain Bill Gates. Ayant le sens des affaires qu'on lui sait, Bill négocie le concept et l'achète le 6 mars 1988. Le projet s'appelle désormais "*Ruby*" chez Microsoft.

C'est là qu'un autre grand futé, Anders Hejlsberg - qui sera plus tard le père de C# - alors chez Borland où il avait créé le Turbo Pascal eut l'idée de faire la même chose. Delphi 1.0 sortira en 1995 comme un "VB Killer". Et même si Delphi dépassa de loin VB sous Win32 (techniquement, mais jamais en nombre d'utilisateurs) il ne s'agissait que d'une copie presque parfaite de VB au langage près. Ce sont dans les petites nuances que l'avantage technique de Delphi allait lui permettre de jouer les outsiders (et par son langage le Pascal que je préfère mille fois au Basic de VB).

Pourquoi je vous parle de ces vieilleries ?

Parce que c'est là que se trouve l'origine de la question du jour !

Composants visuels "à créer non visuellement"

Sous Visual Basic Win16 puis Win32 les fameux "composants visuels" qu'on plaçait (au moins jusqu'à VB 6 et avant VB.NET) sur la surface de design devaient être écrits en C++. Un comble pour un langage (VB) qui était et est surtout utilisé par des informaticiens "formés sur le tas" ou des amateurs (hobbistes). Les universitaires ont toujours eu le Basic et plus encore VB en horreur (pour des raisons techniques valables mais aussi parce que finalement cela rendait le développement peut-être trop simple ce qui les obligeait à descendre un peu de leur piédestal !).

Comme un pied de nez, ce fut ces mêmes C++istes qui firent le vrai succès de VB au départ. En effet, pour gagner du temps sur les concurrents, certains s'étaient rendu compte que VB permettait de présenter très vite aux clients potentiels une maquette fonctionnelle ou au moins interactive… Commercialement cela donnait un coup d'avance, le client pouvait « voir » à quoi l'application ressemblerait pendant que les puristes C/C++ se noyaient en explications et petit crobars beaucoup moins vendeurs pour décrocher une commande… Mais en réalité aucun de ces développeurs n'utilisait VB pour développer : une fois le prototype vendu, ils développaient l'application finale avec un "vrai" langage, le C++, et les affreuses MFC (affreuses, à utiliser et dont, comme par hasard, la version 1.0, la pire, a d'ailleurs été écrite par Sinofsky. No comment).

Ce faisant ils firent le succès de VB et renforcèrent l'envie de Hejlsberg de faire Delphi !

Sous VB il fallait donc être un super développeur C++ pour créer des composants, tâches complexe, ingrate et absolument pas visuelle.
Et surtout situation totalement paradoxale puisque ceux qui développent en VB n'ont

en général aucune connaissance en C++. Je parle bien sûr de ceux qui développent vraiment en VB et non les C++istes qui s'en servaient originellement pour le seul maquettage de leurs applications.
Une brèche dans laquelle Delphi allait s'engouffrer…

L'idée géniale de Delphi fut de tout en faire en Delphi. "Delphi est fait en Delphi" fut l'un des premiers arguments techniques avancés pendant longtemps par Borland. Les composants étaient écrits dans le même langage que l'EDI et que les programmes créés par le développeur. Cela permit l'explosion des composants sous Delphi, en grande partie responsable du succès du produit.

Mais même sous Delphi, créer un composant restait une affaire de spécialistes. Ceux qui écrivaient des composants (et encore plus s'ils étaient *orientés données*) avaient une aura de "super pro". Une certaine hiérarchie sociale se créa à cette époque : les jeunots, les débutants étaient mis à la création des états, les confirmés au développement des écrans, et les caïds adulés et mieux payés à la création des composants. Une hiérarchie qui perdure encore dans certaines de SSII d'ailleurs…

De ce point vue Delphi ne révolutionna rien, il entérina plutôt une situation : la bleusaille à la génération d'état, les bons aux composants.

Les composants de VB et de Delphi étaient malgré tout une idée fantastique.

Quand .NET fut créé avec C# (par le père de Delphi donc, passé de Borland à Microsoft), le rêve ne s'arrêta pas en si bon chemin. .NET reprit bien entendu la notion d'IDE "RAD" avec Visual Studio dont le nom indique bien ses intentions RAD – l'accent donné au Visuel en rappel au développement visuel spécifique des environnements RAD.

Les composants Windows Forms ou ASP.NET se bâtissaient comme ceux de VB et Delphi : non visuellement en réclamant un niveau de qualification un cran au-dessus du développeur "de base". Ce que Delphi appelait "Components" (composants), Visual Studio les appela "Controls" (contrôles). Mais en dehors de cette différence mineure on restait dans les mêmes règles du même jeu.

Apparition de la dualité

Mais très vite, les Windows Forms (et ASP.NET) offrirent une autre façon de créer des contrôles : **les User Control**. L'idée, fort simple - à avoir mais pas à réaliser puisque cela a attendu tout ce temps ! - était de **rendre le développement des composants**

totalement visuel. Briser cette frontière entre les compétences de base pour créer des applis et *super compétences* pour créer des contrôles.

.NET offrit ainsi rapidement un nouveau niveau d'abstraction : le **User Control**. On les construisait visuellement, comme des "mini applications" autonomes, on pouvait ensuite les utiliser comme les "vrais" composants fournis de base avec l'EDI. **Un progrès immense.**

Mais voilà, tout ne pouvait se faire visuellement, créer un contrôle, un composant, c'est comme une application : c'est un travail très "ouvert" on peut faire tout de ce qu'on veut. Hélas concevoir un environnement graphique qui par drag'n drop offrirait autant de souplesse que du code a toujours été un challenge jamais atteint (jusqu'il y a peu, je vais y arriver).

De fait, malgré la bonne volonté de vouloir simplifier les choses, cette époque restera marquée par la complexification de la situation en créant deux types de contrôles : les "vrais", les durs les tatoués, ceux créés entièrement par code qui peuvent tout faire et dont la conception est réservée à l'élite des développeurs, et les "faux", les User Controls, créés facilement sans besoin de compétences très pointues mais ne pouvant couvrir toutes les situations…

Pendant des années cette dualité *Custom Control / User Control* domina le monde des composants. Renforçant les castes évoquées plus haut, entretenant la confusion chez les débutants.

L'enfer est souvent pavé de bonnes intentions, c'en est une illustration parfaite.

Et Dieu créa la flemme…

Le bon développeur est un fainéant. Et comme la création de "vrais" contrôles étaient une tâches réservées aux cadors du clavier, il fallait bien qu'un jour ils se lassent de se taper le job le moins rigolo. Eux aussi voulaient faire joujou avec le drag'n drop !

Dans un retournement de situation de type arroseur/arrosé, les "bons" se trouvèrent coincés derrière leur clavier, avec des machines moins puissantes que les jeunots développant à coup de drag'n drop des User Control sur de beaux écrans couleurs (à cette lointaine époque c'était un luxe !). C'est tout juste si certains patrons trop pingres n'allaient pas leur supprimer la souris pour faire des économies vu que de toute façon créer de "vrais" contrôles se faisait exclusivement au clavier ! J'exagère à peine.

Une telle situation ne pouvait s'éterniser, c'est une évidence. Il fallait que l'Ordre revienne, que les Hiérarchies retrouvent leur place comme "au bon vieux temps". Hélas pour les réactionnaires l'avenir n'a pas pris cette forme !

… et Microsoft Blend…

Blend, ce produit magnifique, fut créé et mis sur le marché à peu près en même temps que la première version de Silverlight (même si Blend traitait aussi bien WPF comme il traite désormais Windows Phone ou WinRT).

Blend ne semble pas avoir été acheté à un développeur génial comme VB, en tout cas l'histoire « officielle » est que ce produit est un développement Microsoft dont le nom de code fut *Sparkle*. Mais qu'importe, ce qui nous intéresse c'est ce qu'est Blend et ce qu'il est devenu en quelques années.

Autant le dire tout de suite, n'étant qu'un outil par-dessus le Framework, Blend ne révolutionne pas les concepts sous-jacents : .NET et ses langages principaux supportent toujours, même sous WPF 4.5, la double notion ambigüe de Custom Control / User Control.

Le génie de Blend est d'offrir une plateforme graphique dont **la puissance permet enfin d'aborder la création des "vrais" composants de façon totalement graphique** comme les User Controls.

Le principe reste un peu différent, et le fonctionnel reste du code, mais au final le visuel est enfin créé visuellement !

Il s'agit là d'une révolution qui a connu une gestation de plus de vingt ans ! Difficile pour les plus jeunes lecteurs de s'en rendre compte et d'en mesure la portée, forcément.

La création d'applications Windows visuellement, c'est VB en 1988.

La création des composants dans le même langage que les applications, c'est Delphi 1.0 en 1995.

Puis plus rien jusqu'à la fin de la première décennie du 3eme millénaire…

En 2006 le projet *Sparkle* devient finalement "Blend" au lieu de "Microsoft Expression Interactive Designer". C'est plus court. Mais c'est en 2007 que la première CPT publique sera releasée.

Blend 3 en 2009 marque une première étape de maturité confirmée par Blend 4 et les versions suivantes désormais accolées à Visual Studio – la suite Expression ayant été arrêtée.

Plus de vingt ans à attendre pour que l'idée de composant, unique au départ, ne redevienne unique et que ne cesse la dualité Control / User Control.

L'intérêt de créer des Custom Controls plutôt que des User Controls ?

Il faut malgré tout aborder cette question qui vous brûle certainement les lèvres depuis le début, je vous ai assez fait attendre !

Vous l'aurez deviné, si je m'évertue à replacer dans leur contexte historique et technique les Custom Controls et les User Controls c'est bien parce qu'ils sont différent *au final*.

S'il ne s'agissait que d'histoire cela manquerait d'intérêt pratique et s'il ne s'agissait que de technique cela serait d'un ennuyeux épouvantable…

Car nous n'avons pas soulevé la question que sous-tend cette différence : L'intérêt de développer des contrôles, quels qu'ils soient.

L'intérêt principal des contrôles visuels c'est d'être des briques de construction d'un niveau très élevé puisqu'ils encapsulent en un seul concept à la fois la forme et la fond, à la fois le visuel et le code utile. Il est donc simple de comprendre qu'en manipulant de tels objets on puisse construire plus vite un édifice logiciel complexe et performant qu'en devant créer ponctuellement et de façon répétitive un code dans sa totalité pour chaque projet.

De cet intérêt premier en découle un second : la réutilisabilité. Ne pas réinventer la roue à chaque projet, ne pas réécrire le même code en créant de nouveaux bogues est en soi un but recherché depuis toujours par les informaticiens. L'étape de la programmation objet puis celle des bibliothèques d'objets ont été décisives dans cette quête. Les contrôles visuels sont le dernier maillon de celle-ci pour les applications modernes où le visuel joue à part égale avec le code fonctionnel et réclame donc une attention et une quantité de code qui justifient pleinement d'appliquer le concept de réutilisabilité comme on le faisait avant avec les bibliothèques de code pur.

Mais tout cela se comprend au niveau des généralités qui expliquent la recherche d'une unification du visuel et du code dans des méta-objets, les composants visuels, la naissance du concept de RAD et tout ce qui nous a amené à créer des applications attractives, interactives et plaisantes. Belle évolution depuis le code binaire des premiers ordinateurs se programmant avec des interrupteurs !

Techniquement, pour un développeur XAML aujourd'hui, pourquoi développer des contrôles sous une forme ou une autre ? Quelle est la motivation pratique ?

En dehors de la réutilisabilité du code, en dehors de l'encapsulation du visuel et du code dans des entités manipulables par drag'n drop, en dehors de toutes ces raisons déjà suffisantes, il y en a une autre d'importance égale : la personnalisation de l'interface visuelle.

XAML est livré avec peu de composants et beaucoup de puissance. La première chose qui surprenait en regardant Blend au départ était son dénuement ! `TextBlock`, `TextBox`, `Rectangle`… une poignée de composants là où des environnements comme VB ou Delphi avaient des palettes entières de composants divers et variés !

C'est que XAML est *résolument différent* des autres environnements justement. Sa « pauvreté » n'est qu'apparente, tout se joue au niveau de la *sophistication de ce langage*, dans sa capacité unique à travailler en mode vectoriel en temps réel et à *permettre toutes les personnalisations qu'on désire*.

Bien entendu le Framework .NET lui-même est un socle indispensable à cette puissance dont il est le moteur. Prenons par exemple un environnement comme Delphi, si vous disposiez d'un composant tel que le bloc de texte (le `TLabel`) ou de saisie de texte (le `TEdit`) ces composants ne savaient pas se lier à du code ou d'autres composants, le Binding .NET n'existait pas, seuls les événements permettait de réagir aux changements des composants. De plus ils étaient totalement fermés et la seule personnalisation possible était de jouer avec les propriétés exposées. De fait si désirait obtenir un composant se liant aux données par exemple la seule chose qu'on pouvait faire était de dupliquer les composants existants pour les lier à une source de données bien particulière, un SGDB généralement. Ainsi le `TLabel` devait avoir son pendant dans le `TDBLabel` pour qu'un texte puisse être lié à une source de données, le `TEdit` devait avoir son double *orienté données* dans le `TDbEdit`, etc, etc. Tout cela doublait le nombre de composants.

Pire, comme je le disais les composants étant « fermés » (c'est-à-dire non personnalisables en dehors des propriétés qu'ils exposaient), si vous vouliez un bout

de texte ayant un look particulier il fallait réécrire un nouveau composant… Et si vous vouliez qu'il puisse se lier à des données, il fallait en créer une autre version orientée données !

On comprend vite pourquoi les palettes de composants s'entassaient rapidement : chaque look & feel d'un composant réclamait l'écriture d'un composant particulier qui lui-même devait se doubler d'une version orientée données si on voulait pouvoir le lier et l'utiliser autrement que par ses propriétés et sa gestion d'évènements…

XAML a balayé tout cette lourdeur. XAML a totalement réinventé le concept de composant visuel.

De base tous les composants, les contrôles, sont « orientés données » puisque tous exposent des propriétés qui, sauf rares exceptions, sont des propriétés de dépendance offrant un Binding immédiat. Le concept de « données » sous .NET a été « explosé », sous Delphi ce concept sous-tendait l'utilisation d'une source de données de type SGBD, sous .NET le concept de source de données a été étendu à « tout objet » ou liste d'objets.

Mais plus loin encore, XAML a apporté le *templating*, c'est-à-dire la possibilité incroyable et unique (ne cherchez pas l'équivalent ailleurs il n'y en a pas) de pouvoir *totalement changer la forme d'un contrôle sans avoir besoin de le recompiler*, d'adapter ou modifier son code !

XAML a aussi apporté avec les propriétés de dépendance des possibilités nouvelles comme les *propriétés attachées* qui permettent à une classe de décorer ses enfants visuels de nouvelles propriétés sans avoir à modifier ces contrôles.

XAML a apporté tellement de bouleversements qu'il faut il est vrai fournir un effort pour aborder ce langage et en comprendre les mécanismes.

Mais pourquoi donc développer de nouveaux contrôles puisse que ceux qui existent sont si versatiles ?

Il existe une multitude de niveaux d'enrichissements et de personnalisations disponibles sous XAML. En faire une hiérarchie est assez difficile car tout dépend du but et donc du point de vue. Mais malgré tout il est possible de dresser une suite d'actions, des plus simples au plus sophistiquées, qu'un développeur peut décider d'entreprendre pour personnaliser une interface. C'est en regardant de plus près cette hiérarchie qu'on répondra le mieux à la question :

Niveau 0 – Réutiliser un contrôle existant

On prend dans la palette des composants le contrôle qui convient à ce qu'on veut faire, par exemple je veux afficher un texte, je prends un `TextBlock` et je le dépose sur l'interface à l'endroit où j'en ai besoin.

Niveau 1 – Utiliser les propriétés des contrôles

En toute logique le premier niveau de personnalisation consiste à utiliser les propriétés offertes par les contrôles pour les personnaliser. Je souhaite que le `TextBlock` que je viens de poser utilise la fonte `Arial` en corps 12 rouge et qu'il affiche « *bonjour* » pour cela je modifierai en conséquence et respectivement les propriétés `FontFamilly`, `FontSize`, `ForeGround` et `Text`.

Niveau 2 – Modifier le Template visuel et le Style

Je désire maintenant que mon `TextBlock` présente son texte dans un rectangle bleu transparent à bords arrondis. Cela n'est pas prévu par le contrôle, mais grâce à XAML c'est possible : je vais modifier son *Template* visuel, insérer un rectangle derrière le texte, arrondir les bords du rectangle, lier la couleur de remplissage de ce dernier à la couleur d'arrière-plan du `TextBlock` par un *Template Binding*. Je créerai ensuite un `Style` et dans ce `Style` je vais fixer la couleur d'arrière-plan à bleu transparent. Le nouveau `Style` est applicable à tout `TextBlock` à volonté. Toute modification du `Style` et du `Template` qui est lié s'appliquera automatiquement à tous les `TextBlocks` faisant usage de ce `Style`.

Niveau 3 – Les contrôles composites : les User Controls.

Je souhaite désormais que mon `TextBlock` soit accompagné d'une case à cocher (`CheckBox`). C'est grâce à la création d'un User Control que je vais pouvoir associer ces deux contrôles dans une même entité qui sera nommée.

Si je le désire (ce qui est souvent le cas pour des raisons fonctionnelles) je vais intervenir sur le code de ce User Control pour exposer de nouvelles propriétés (par exemple l'état du `CheckBox`, le texte à afficher dans le `TextBlock`) ou pour gérer la dynamique entre les contrôles du composite (par exemple basculer la couleur du `TextBlock` selon que la case est cochée ou non).

Un User Control est tellement versatile qu'une application peut être entièrement construite avec des User Controls jouant le rôle de pages à afficher (c'est le cas sous Silverlight par exemple même s'il existe d'autres types de conteneur de page).

Il sera facile dans le même projet de réutiliser un User Control. Je pourrais ainsi placer une dizaine de « texte avec case à cocher » dans un conteneur ou un autre User Control et piloter leurs propriétés.

Toutefois, si je peux modifier simplement le `Template` d'un `TextBlock` ou d'une `CheckBox`, je m'aperçois rapidement que je ne peux pas « templater » mon User Control… Si cela était possible je risquerai de casser son fonctionnement. En effet, le code que j'ai ajouté pour gérer le clic de la `CheckBox` pour changer la couleur du `TextBlock` par exemple, et bien ce code ne fonctionnerait plus si l'un ou l'autre de ses contrôles changeait de nom ou disparaissait…

Un User Control ne peut pas être utilisé comme un simple contrôle, il ne possède pas de `Template`. Pire, il peut utiliser tout ou partie d'autres déclarations XAML, Styles, Templates, dictionnaire de ressources… et même s'appuyer sur d'autres portions du code du projet pour fonctionner. Je m'aperçois ici que la réutilisabilité de mon User Control est assez limitée…

Niveau 4 – Créer des propriétés attachées

A partir d'ici le numéro d'ordre du niveau devient plus difficile à fixer. Jusqu'au niveau 3 tout pouvait se faire visuellement par des mécanismes purement XAML. Au niveau 3 on pouvait ajouter du code mais cela était optionnel. A partir de maintenant il faudra mélanger XAML et code pour pousser plus loin la personnalisation.

C'est le cas des propriétés attachées qui permettront d'ajouter des propriétés à tout enfant visuel d'un conteneur. Tout contrôle placé dans ce conteneur disposera alors de nouvelles propriétés qui n'auront d'intérêt et de sens que dans le contexte du conteneur en question. Par exemple le contrôle `Canvas` contient la définition de propriétés attachées conférant à ses enfants visuels les propriétés `X` et `Y` permettant de les positionner sur la surface du `Canvas`.

Niveau 5 – Les behaviors

Les Behaviors (comportements) sont traités dans ce livre, je ne vais pas m'étendre trop. Mais supposons que je désire qu'un effet de tremblement se produise lorsque la souris passe au-dessus de certains contrôles.

Je pourrais bien entendu templater tous les contrôles en question pour modifier l'état `MouseOver` par exemple. Mais on le sent bien, ce serait une solution affreusement longue à mettre en place, brouillon et définitivement peu réutilisable par essence même.

C'est là que je créerai un `Behavior` qui apportera à tout contrôle sur lequel je le déposerai ce fameux comportement de tremblement.

Comment le faire est une autre question débattue ailleurs dans ce livre, mais le principe est là : les behaviors permettent d'ajouter des comportements nouveaux, donc un haut niveau de personnalisation, à des contrôles existants sans même posséder ni voir ni modifier leur code source. Et cela de façon simple et réutilisable à l'infini…

Niveau 6 – Les Custom Controls

Je reprends mon User Control créé au niveau 3. Et je constate qu'en effet il est très lié au projet dans lequel il a été créé, donc qu'il est difficilement réutilisable et qu'en plus il est « verrouillé » et que je ne peux pas le « templater » pour l'adapter à d'autres contextes.

L'ultime étape dans la personnalisation de l'interface de mon application consiste donc à créer un Custom Control, c'est un vrai Control, mais « custom » puisque je vais le créer pour en façonner tous les aspects et le rendre enfin « templatable ».

Généralement je créerai mon Custom Control dans une bibliothèque de contrôles, même s'il y est tout seul, afin de le rendre parfaitement réutilisable dans tous mes projets.

Arriver à ce niveau de personnalisation de l'interface visuelle tout est permis, tout est possible, et le résultat est un Control semblable à ceux fournis de base, s'insérant dans les palettes de Blend ou Visual Studio, utilisable par simple drag'n drop, et personnalisable via le Templating.

Faisons le point !

La question, pas si innocente que cela on s'en rend compte, qui était posée était de savoir pourquoi il faudrait créer des Custom Controls ou des User Controls.

Les raisons sont nombreuses et les 6 niveaux de personnalisation présentés ici permettent de mieux les appréhender.

Maintenant que nous savons « pourquoi », nous pouvons voir « comment »…

La création visuelle des Custom Controls

Les User Controls, c'est facile, il suffit de faire "nouveau User Control" dans les menus de Blend ou VS pour en créer. Ensuite on dispose d'une surface graphique pour y dessiner ce qu'on veut et d'une page de code pour y placer le fonctionnel.

Mais les Controls, les "vrais". Comment fait-on alors ?

Avec Visual Studio les choses changent peu. C'est avant tout un environnement de codeur, pas de designer.

Mais avec Blend, l'impossible devient réalisable…

La méthode

Elle est bien simple… On ajoute une nouvelle Classe. Comme on le ferait pour une classe métier ou autre.

Aucun visuel n'est bien entendu attaché à ce code. Pas de page XAML cachée derrière. Juste une classe.

Commençons par la faire héritée d'un `Control`, par exemple `ContentControl`. Comme toujours le choix de la classe mère est essentiel dans la construction d'une nouvelle classe, mais c'est un sujet que nous n'aborderons pas ici tellement il nous éloignerait du but.

Un `Control` possède des états visuels gérés par le *Visual State Manager*. Il suffit de décorer la classe des bons attributs :

```
1: [TemplateVisualState(GroupName = CommonStates, Name = StateA)]
2: [TemplateVisualState(GroupName = CommonStates, Name = StateB)]
3:
4: public class MonControl : ContentControl { ... }
```

Code 38 - TemplateVisualState

Dans l'exemple ci-dessus on créé un seul groupe d'état (dont le nom réel se trouve dans une constante chaîne contenu dans le code de la classe) "`CommonStates`" et deux états dans ce groupe "`StateA`" et "`StateB`",peu importe leur rôle et signification nous restons ici dans le principe. La classe `MonControl` se trouve ainsi dotée de deux états contenus dans un groupe d'états, le tout sera accessible dans le panneau du VSM sous Blend.

Le fonctionnel

Le fonctionnel c'est du code, nous n'en sommes pas à la création de programmes par commande vocale ou lecture des pensées… Donc le fonctionnel est codé "normalement" tout dépend de ce qu'on veut faire. Cela peut être très simple ou ultra sophistiqué, l'étendue est telle qu'il est même impossible d'en dire plus. On trouve des propriétés CLR, des propriétés de dépendance, des méthodes.

Bien entendu on retrouvera la logique qui permet de changer d'état (entre `StateA` et `StateB` dans l'exemple ci-dessus).

Mais le visuel alors ?

En fait le visuel ne compte pas beaucoup pour le fonctionnel… et cette **déconnexion totale** est une avancée incroyable dont le crédit est à mettre aux équipes de développement Microsoft attachées au Framework, à Blend et à Xaml.

En fait, votre code fonctionnel se moque royalement de la "tête" que le contrôle aura. La seule chose qui lui importe c'est de pouvoir y trouver ***certains éléments*** dont il va avoir besoin. Par exemple un bouton, une zone de titre, etc.

Pour déclarer ces éléments il n'y a qu'à décorer encore un peu plus notre classe avec d'autres attributs :

```
1: [TemplatePart(Name = PanelContent, Type = typeof(FrameworkElement))]
2: [TemplatePart(Name = DoItButton, Type = typeof(FrameworkElement))]
3:
4: [TemplateVisualState(GroupName = CommonStates, Name = StateA)]
5: [TemplateVisualState(GroupName = CommonStates, Name = StateB)]
6:
7: public class MonControl : ContentControl { ... }
```

Code 39 - TemplatePart

Le code ci-dessus montre les deux "`TemplatePart`" ajoutés. Notre code aura besoin d'une zone pour un contenu, "`PanelContent`" qui pourra être de tout type dérivé de FrameworkElement (ici on balaye large !) et d'un bouton pour lancer le fonctionnement, "`DoItButton`" que nous voyons aussi comme un descendant de FrameworkElement.

Pourquoi voir "si large" ? Le bouton pourrait être limité à un descendant de `ButtonBase` par exemple. Le panneau de contenu pourrait descendre de `Panel`.

En réalité, et sauf cas vraiment particulier, le concepteur d'un `Control` se doit de rester le plus large possible dans les contraintes qu'il impose au "*template parts*" (parties de template). Pourquoi brider la créativité de celui qui utilisera le contrôle et essayera de le templater ?

Par exemple pour le bouton, notre code a besoin (c'est un pur exemple) d'un bouton pour déclencher certaines actions de `MonControl`. Nous nous moquons totalement qu'au final le bouton soit remplacé par un carré ou une image PNG… Tous ces éléments descendent de FrameworkElement qui supporte les événements de souris…

Ainsi, s'il s'agit vraiment d'un bouton, nous programmerons son "`Click`", mais s'il s'agit de n'importe quoi d'autre nous nous contenterons de son `MouseLeftButtonDown` par exemple.

Bien entendu cette stratégie n'est qu'une parmi d'autres. Chaque Custom Control réclame comme une application beaucoup de réflexion et de planification.

Comment relier le visuel au code ?

C'est très gentil tout çà, mais où est le visuel ?

Pour l'instant il n'y en a pas. Et notre classe pourrait très bien s'en contenter. Visuellement elle ne donnerait rien c'est sûr… Il faut donc aller un cran plus loin.

La première étape consiste d'abord à relier le code au visuel.

Comment ? Hein ? Je viens de dire qu'il n'existait pas ? Oui. Et non.

Le visuel à proprement parlé n'existe pas encore, c'est vrai, mais rappelez-vous, nous l'avons complètement conceptualisé ce visuel… totalement dématérialisé. Donc d'un certain point de vue il existe bel et bien : ce sont les attributs `TemplatePart` qui donnent naissance "*virtuellement*" aux éléments graphiques dont notre code aura absolument besoin pour tourner. C'est très abstrait, j'en conviens. Mais il n'empêche, à ce stade du développement de notre `Control` le visuel est déjà "présent" uniquement grâce à ces attributs. Si vous me demandez "*graphiquement*" si le visuel existe, la réponse est non. Mais "*conceptuellement*" oui.

Comment le code peut-il se servir de concepts abstraits ?

Il y a une astuce, par force. Il faut bien obtenir à un moment ou un autre des pointeurs vers les instances des éléments graphiques si on veut pouvoir les manipuler par code, réagir à des événements.

Quelles instances ? … Peu importe…

C'est agaçant non ? ☺

En fait il "existera" bien des instances à un moment donné, ce moment n'est pas arrivé dans le cycle de conception, mais il ne va pas tarder. Pour l'instant les attributs décrivant les *Template Parts* nous suffisent. Mais pour lier le concept au réel, nous allons tout de même devoir ajouter un peu de code en surchargeant la méthode `OnApplyTemplate` du `Control` :

```
 1: public override void OnApplyTemplate()
 2: {
 3:     base.OnApplyTemplate();
 4:     doItButton = GetTemplateChild(DoItButton) as FrameworkElement;
 5:     content = GetTemplateChild(PanelContent) as FrameworkElement;
 6:     if (content != null)
 7:         content.MouseLeftButtonDown += content_MouseLeftButtonDown;
 8:     if (doItButton is ButtonBase)
 9:         (doItButton as ButtonBase).Click += doItButton_Click;
10:     else if (doItButton != null)
        doItButton.MouseLeftButtonDown += doItButton_MouseLeftButtonDown;
11: }
```

Code 40 - OnApplyTemplate

Regardez bien car c'est là que la magie s'opère…

`OnApplyTemplate` est la porte d'entrée que les concepteurs du Framework ont créée pour que le développeur d'un contrôle puisse prendre la main. Cette méthode est en fait appelée par `ApplyTemplate` qui, pour simplifier, est invoquée avant qu'un `Control` ne soit affiché. Nous n'entrerons pas dans ces détails aujourd'hui.

Or, que voit-on dans ce bout de code exemple ?

On voit que nous récupérons dans des variables privées (mais ce n'est qu'une possibilité) les fameuses instances des parties de template. Par exemple la ligne 4 essaye de récupérer la partie "`DoItButton`", la ligne 5 le "`PanelContent`". Des parties visuelles que nous avons déclarées dans les attributs `TemplatePart`.

Une fois ce "minimum syndical graphique" récupéré par le `Control` ce dernier peut agir sur les éléments. Par exemple, en lignes 6 et 7, le `Control` accroche un gestionnaire d'événement de bouton gauche de la souris à ce qui joue le rôle de "`PanelContent`", peu importe par quoi il est représenté graphiquement.

Les lignes 8 à 10 font la même chose avec l'élément qui joue le rôle de "`DoItButton`" (qui serait un bouton pour lancer une action du `Control`). S'il s'agit d'un descendant de `ButtonBase`, alors nous pouvons programmer son `Click` ce qui est parfait. Mais si jamais ce n'est pas un `ButtonBase`, qu'importe, nous programmons le bouton gauche de la souris.

Bien entendu, derrière ces gestionnaires événements se cache du code fonctionnel, le bouton "`DoItButton`" fait "*quelque chose*". Peut-être lancer une impression, accéder à des données, peu importe, "`MonControl`" n'est pas un vrai `Control` c'est un squelette qui permet d'illustrer mon propos. Dans la réalité il n'y aura peut-être pas de gestionnaires d'événements, ou d'autres, c'est totalement "open" c'est vous qui voyez… selon ce que vous voulez faire de votre contrôle (son utilité finale).

Et la grande révolution elle est où ?

Elle se trouve d'abord dans le fait que nous venons d'écrire un `Control` entièrement par code (Youpi ! comme il y a vingt ans ! – Pfff … vous n'êtes que des grincheux !).

Il y a tout de même une *grosse différence* : ce code utilise et manipule de l'interface, des graphismes, de la 3D sous WPF, des transformations, des effets, des animations mais *tout cela n'a pas besoin d'exister réellement* !

Le **découplage fort entre code et design** prend ici tout son sens. Du code il y en aura toujours. Du moins tant que les ordinateurs ne se programmeront pas tous seuls, ce qui n'est quand même pas demain la veille.

En revanche, permettre au développeur de faire son travail sans se prendre les pieds dans la réalisation graphique, laisser même le droit au changement, à l'erreur, ça c'est nouveau.

Mais ce n'est pas tout !

La vraie révolution vient du fait que vous allez pouvoir maintenant ajouter la partie visuelle à votre contrôle, et ce, **visuellement**, grâce à Blend !

Un Contrôle comme nous venons de le voir n'a pas de Xaml accroché au code. Mais il en faut bien un peu pour donner figure à ce qui ne resterait que virtuel. Mais au lieu d'un objet Xaml de type `Page` par exemple, notre Control se nourrit d'autre chose : de ***ressources***.

Quelles ressources ? Au moins une, un **Template de Control**. Par habitude ce template est stocké dans un dictionnaire de ressource. Cela peut être `App.Xaml` mais

ce serait relier le contrôle à une application donnée. Pas très malin. Donc le template en question sera stocké dans un dictionnaire de ressource séparé. Souvent appelé "`Generic.xaml`" et stocké dans un sous répertoire "`Theme`".

Ce template définit un *style* dont le `TargetType` est la classe de notre contrôle. Ce style définit ensuite un Template qui donne le visuel.

Voici le début d'un tel dictionnaire :

```
 1: <ResourceDictionary
 2:   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 3:   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4:   xmlns:local="clr-namespace:MaSociété.MesControls"
 5:   xmlns:vsm="clr-namespace:System.Windows;assembly=System.Windows"
 6:   >
 7:     <Style TargetType="local:MonControl">
 8:         <Setter Property="Template">
 9:             <Setter.Value>
10:                 <ControlTemplate TargetType="local:ContentPanel">
11:                     <Grid>
12:                         <vsm:VisualStateManager.VisualStateGroups>
13:                         ....
```

XAML 57 - Exemple de définition de Style dans un dictionnaire de ressources

Un dictionnaire de ressource Xaml tout ce qu'il y a de plus classique donc, avec un template de contrôle lui aussi très classique comme tous les templates qu'on peut créer avec Blend.

Ce visuel "de base" est l'aspect qu'aura votre contrôle "par défaut". Certains l'écrivent totalement à la main (si c'est très simple pourquoi pas), mais franchement mieux vaut utiliser le bon outil : Blend.

Comment créer le Style par défaut visuellement ?

Notre Control est écrit, il se compile bien. Ajoutons juste un projet de test à la solution, référençons notre librairie de contrôles, et plaçons une instance de notre contrôle sur la page principale du projet de test.

On ne voit rien, ou presque. Normal, le pauvre contrôle n'a aucun visuel !

Un clic dans l'arbre visuel pour sélectionner le contrôle, puis on va dans le menu Objet et on choisit "Edit Style" (*Modifier le Style* dans la VF je pense). Cela ouvre un nouveau style qu'on prendra soin de placer dans un dictionnaire de ressource à part

qu'on appellera par exemple "`Generic.xaml`"... Une fois dans le **Style** ne reste plus qu'à créer le **Template** (clic droit, modifier le template, et choisir d'en créer un à partir de rien).

Et vous voici **en visuel** sur la création totalement libre du visuel de votre Control !

Accrocher les wagons au train

Vous allez placer ce que vous voulez graphiquement, mais rappelez-vous, votre contrôle a besoin de certains objets particuliers pour fonctionner. Ce sont ceux définis dans les attributs `TemplatePart`. Blend offre un panneau "`Parts`", là on peut voir toutes les parties dont a besoin le contrôle pour fonctionner, et on peut savoir si on a déjà affecté quelque chose ou non à chacune d'entre elles.

Par exemple, je pose un bouton sur le visuel et je me dis "*je veux que ce bouton joue le rôle de la partie "`DoItButton`"*". Comment faire ? Très simple : je clique sur le bouton pour le sélectionner, et clic-droit "*Make into part of MonControl*", ce qui ouvre un sous-menu dans lequel je vois toutes les parties, dont "`DoItButton`". Je clique sur ce dernier et voilà ! Le bouton est maintenant **associé** à la partie "`DoItButton`". Partie qui sera donc récupérée dons le `OnApplyTemplate` du Control, code que nous avons vu plus haut... La boucle est bouclée, les wagons graphiques sont accrochés au train du code...

Ce n'est pas magique ça ?

Extraire les ressources

Maintenant nous disposons d'un projet de test possédant un fichier `Generic.xaml` qui contient tout le template visuel de notre contrôle. Son "look" par défaut.

Il suffit de copier ce fichier dans la librairie où se trouve `MonControl`, de vérifier que le `Style` a bien pour `TargetType` la classe `MonControl`. Peut-être faudra-t-il vérifier et accorder le namespace. Rien de bien méchant.

On peut maintenait supprimer le projet de test, recompiler la librairie de `MonControl`.

Créez une nouvelle application, référencez la librairie en question, poser une instance de `MonControl` sur la page : voici un joli Control avec un look par défaut... A ce stade vous devenez designer utilisateur du contrôle, et vous pouvez le (re) templater pour personnaliser son aspect, voire tout changer !

Conclusion

Créer de “vrais” contrôles étaient jusqu’à il y a peu une tâche ingrate, totalement non visuelle.

Grâce au Framework .NET et à Expression Blend – et avec un poil de ruse je l’avoue - il devient possible, pour la première fois depuis vingt ans, de rendre la création graphique d’un “vrai” contrôle totalement visuelle.

C’est une avancée énorme, une révolution silencieuse. Etrangement les gens sont plus ébaubis par le support des vidéo HD. C’est peut-être plus facile à voir tout simplement…

En tout cas, entre les vidéos HD ou tout un tas de choses sympathiques et la création visuelle de “vrais” contrôles, moi je n’hésite pas un seul instant, la vraie révolution technique est bien dans cette possibilité jusqu’à lors un pur rêve. Delphi, Java, PHP, et tous les autres environnements plus récents sont conceptuellement largués. Même les « génies » de Apple ne sont pas capable de fournir des langages aussi haut de gamme, tout dans le blabla mais techniquement du vide. Que dire de Android qui se programme visuellement comme du HTML à grand coup de PNG à la noix et de nine-patches, si ce n’est que c’est pitoyable autant de retard technologique ? Il ne suffit pas d’envahir le monde avec un Linux modifié comme le fait Android, il ne suffit pas de vouer un culte à un pseudo gourou mort et à une marque comme le fait Apple, tous ces gens sont des vendeurs de savonnettes, c’est sympa une savonnette bien parfumée, mais ce n’est pas l’informatique, seul Microsoft a été capable d’aller aussi loin dans les concepts et dans la qualité du tooling. Il est dommage que des maladresses de gestion et de marketing n’aient pas permis à XAML d’avoir la reconnaissance qu’il mérite. Mais un jour son heure viendra, forcément. Avec le foisonnement des form-factors, le vectoriel est forcément la stratégie la plus adaptée. Entre temps il reste WPF, WinRT, Windows Phone et dans une moindre mesure aujourd’hui Silverlight pour se faire plaisir avec de vrais OS, de vraies plateformes, de vrais langages et un tooling qui relève de la SF par rapport à la concurrence…

J’espère que ce billet vous aura permis de vous en rendre compte et vous aura donnée l’envie d’en savoir plus. Je n’ai pas la prétention ici d’avoir fait le tour de la conception des contrôles. Juste d’un point crucial qui marque l’histoire du développement même si personne n’en parle.

Ecrire un UserControl composite

Les command links sont ces petites boîtes qu'on a vu apparaître les pages de Vista puis de Windows 7 et qui comportent généralement une icône et deux lignes de texte. Elles permettent de naviguer entre les pages de la documentation, des options de l'OS, etc. C'est en tout cas une base parfaite pour parler des contrôles composites !

Command Link à quoi cela ressemble ?

Figure 76 - Visuel d'un Command Link

Ci-dessus un extrait de MSDN montrant ce qu'est un *Command Link*. On reconnait facilement ces boîtes ayant un filet bleu clair, une icône en haut à gauche, une ligne de titre et, en corps plus fin, juste en dessous, des explications. Lorsqu'on clique sur le titre on navigue vers la page dont il est fait état. Cela peut être une page Web externe ou bien une page de l'application elle-même.

Si on désire disposer d'un contrôle encapsulant ce visuel et ce comportement il faut créer un contrôle composite. Les User Controls sont parfaits pour obtenir rapidement un résultat satisfaisant. Ils ont aussi l'avantage de se créer visuellement.

Prétexte parfait pour parler de la création des User Controls donc !

On verra qu'on pourrait choisir de développer un Custom Control, pour tous les avantages que ce type de contrôle apporte mais c'est un sujet que j'ai longuement traité dans « *Custom Control vs User Control* » quelques pages plus haut…

La technique est la même qu'on soit sous WPF, Silverlight, Windows Phone ou WinRT. L'exemple développé ci-dessous utilise Silverlight, il fallait bien choisir et que j'ai toujours eu un faible pour ce dernier, encore aujourd'hui.

Créer le UserControl

Le plus simple est de créer une nouvelle application et d'y ajouter un nouveau `UserControl` :

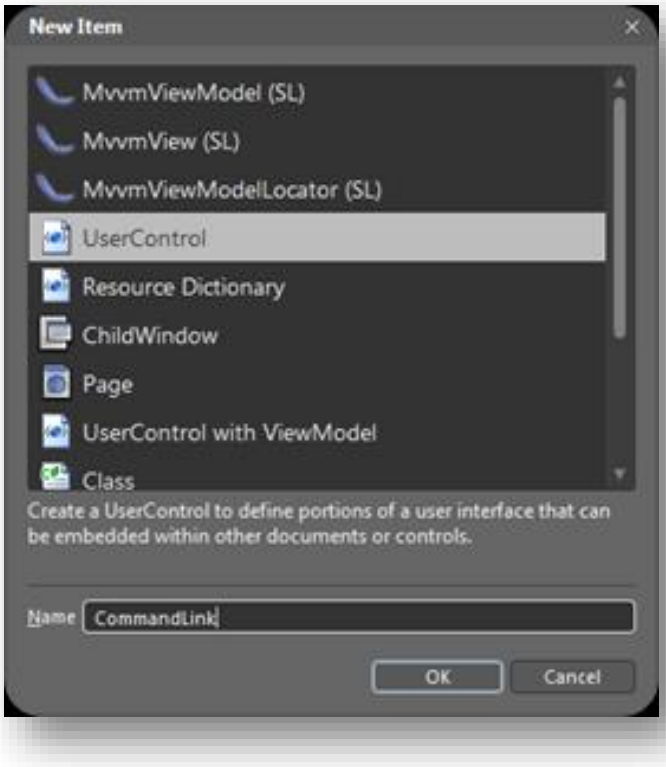


Figure 77 - Création d'un UserControl sous Blend

Créer le look

Le design d'un tel élément n'est pas très compliqué et nécessite juste de positionner une grille :

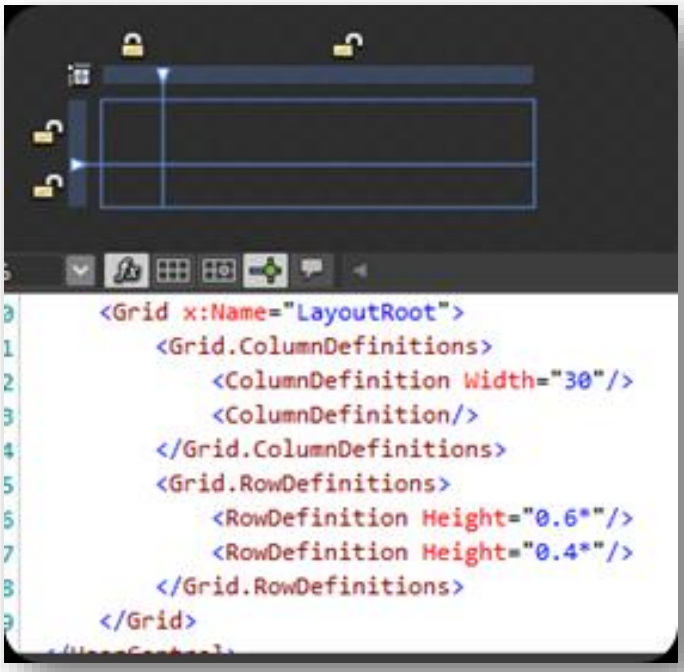


Figure 78 - Définition du visuel d'un UserControl

On ajoute au projet, dans un répertoire qu'on nommera `Images` de préférence, l'icône qui servira d'accroche. Vous pouvez mettre ce que vous voulez. Dans une version plus sophistiquée du `UserControl` il devrait être possible de placer l'icône de son choix. Mais un `UserControl` ne permet pas vraiment ce genre de chose. Il faudrait concevoir un Custom. Pour l'instant je vais choisir une petite flèche verte :

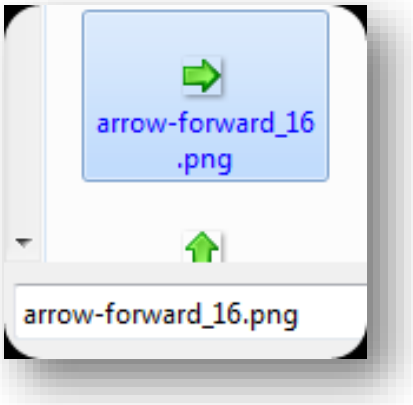


Figure 79 - Icône pour Command Link

L'icône sera positionnée dans le coin supérieur gauche de la grille que nous avons créée. Cette dernière va maintenant être entourée par un border :

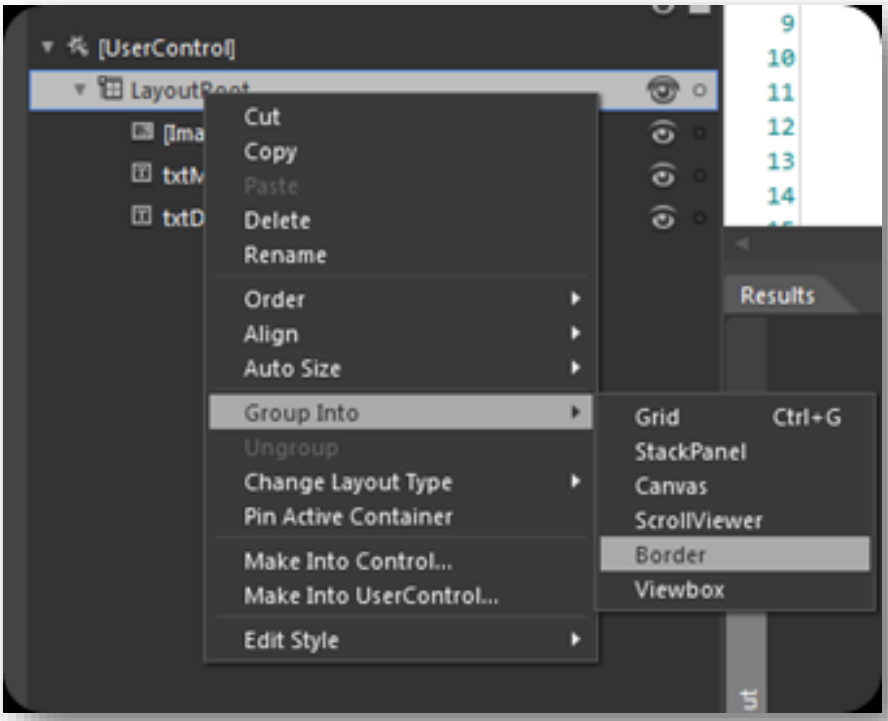


Figure 80 - Finalisation du visuel du UserControl

Deux `TextBlock` seront placés dans la grille. Un premier avec un corps assez gros, celui du dessous en plus petit (pour la taille c'est vous qui voyez, il faut que cela soit lisible sans être ni trop gros ni trop petit…).

Le résultat donne pour le moment :

Figure 81 - L'aspect du Command Link

C'est sobre et efficace.

Pour donner un peu de vie, nous allons créer une animation "`MoveArrow`" qui va déplacer la flèche vers la droite et la ramener à sa position (avec un *easing* pour que le mouvement soit plus agréable) :

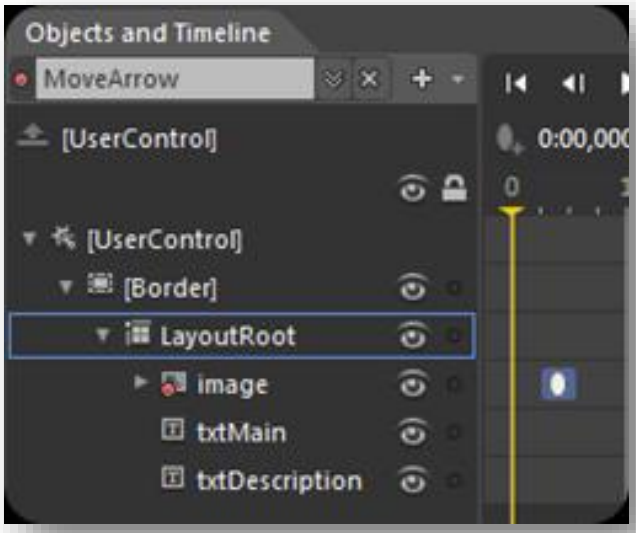


Figure 82 - Animation et Timelime

L'animation est placée en mode autoreverse comme cela nous n'avons qu'une moitié d'animation à créer et nous sommes sûr qu'à la fin de celle-ci la flèche aura retrouvé sa position de départ.

On notera que les couleurs de fond et d'avant plan sont reliées, par *Template Binding*, aux propriétés équivalentes du `UserControl`. De cette façon l'utilisateur de ce dernier pourra modifier les couleurs sans entrer dans le code du `UserControl`. Car bien entendu, comme je l'ai souligné plusieurs fois un User Control ne se template pas. Il faut donc prévoir dès le départ les façons externes de modifier les éléments principaux du look & feel et cela se fait en exposant des propriétés, de dépendance de préférence afin qu'elles soient utilisables avec Binding XAML.

Utiliser les behaviors

Grâce aux différents behaviors disponibles (selon les profils XAML) il est possible de "dynamiser" notre `UserControl` sans perdre de temps en code ou en animations supplémentaires.

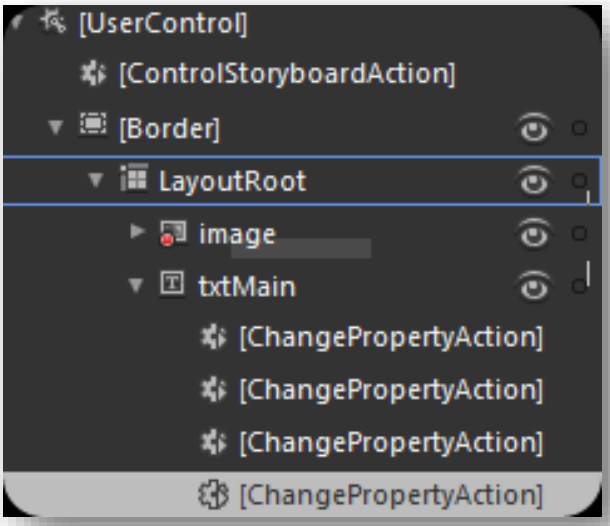


Figure 83 - Mise en place des behaviors

Ici nous voyons un premier behavior : "`ControlStoryBoardAction`" qui est placé sous le `UserControl`. Dans la version finale j'ai préféré l'accrocher au `Border`. Ce behavior sera programmé pour se déclencher sur le `MouseEnter` (du `Border` donc) et déclenchera l'animation "`MoveArrow`". A chaque fois que la souris entrera dans notre User Control, la petite flèche s'agitera quelques instants pour montrer le titre de notre `CommandLink`.

On voit ensuite quatre `ChangePropertyAction`. Il s'agit d'une action (comme le précédent behavior) c'est à dire un `Trigger` (donc une condition de déclenchement). Et comme c'est un `ChangePropertyAction`, l'action réalisée quand le trigger sera déclenché est une modification de propriété.

Nous utilisons 4 instances car nous avons 2 changements à opérer, chacun avec un déclenchement d'entrée et de sortie (1 `Behavior` pour chaque). C'est là que le développement d'un User Control devient un peu moins « beau » pour les puristes. Un peu de code ajouté plus proprement ferait peut-être mieux l'affaire de ce point de vue, mais ce que nous offre le User Control c'est justement la rapidité de création pour un résultat tout aussi fonctionnel qu'un Custom Control. C'est un avantage important dans l'optique d'une meilleure productivité, et si cela ne compte pas pour le technicien, cela importe pour son patron !

Le premier groupe concerne le titre. Je voulais mettre en évidence qu'il s'agit d'un lien et pas seulement d'un texte. Pour cela il suffit d'ajouter un souligné sous le titre, la plupart des gens sont habitués au Hyperliens qui se présentent comme cela. Il y a

donc un `Behavior` déclenché sur le `MouseEnter` qui ajoute un souligné au texte, et un autre sur le `MouseLeave` qui supprime le souligné.

Le second groupe concerne la couleur du titre qui va devenir plus claire à l'entrée de la souris et reprendre sa couleur en sortie. L'effet est bon mais il annule un peu la précaution d'avoir établi un *Template binding* sur le `Foreground` du contrôle. Dans une version plus élaborée il faudrait éviter cette collision sur la propriété couleur du `TextBlock` en jouant non pas directement sur sa couleur (choisie par le développeur utilisateur normalement) mais sur sa transparence ou un autre effet qui ne "bricole" pas la même propriété. Il s'agit là d'une erreur de conception pour une version de production, heureusement ce n'est qu'un exemple ! A vous de créer la V 2.0 ☺

Le code

Nous avons pu concevoir le look & feel totalement sous Blend, mais notre contrôle doit faire des choses, et là il faut du code…

Avant tout il lui faut des propriétés. Des propriétés de dépendances seront utilisées pour le titre, la cible de navigation, la description, etc. Les DP sont utilisables dans des bindings ce qui rendra notre UC plus versatile que l'utilisation de simples propriétés CLR.

Certes les DP sont plus longues à écrire, mais il suffit d'avoir un snippet adapté sous VS et le tour est joué !

Ces propriétés seront complétées d'une description et d'une catégorie, sous Blend (ou VS) cela est plus pratique pour le développeur utilisateur :

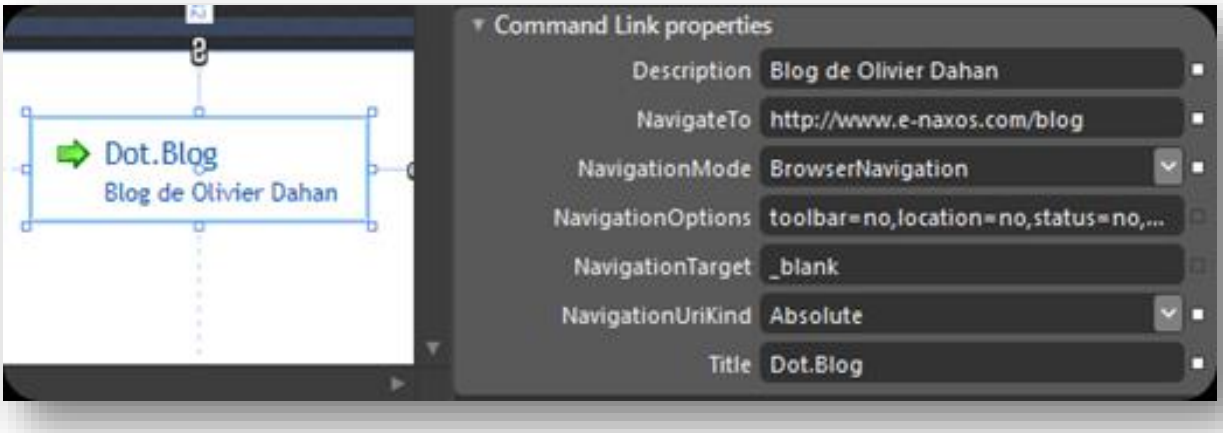


Figure 84 - les propriétés de notre UserControl sous Blend

On voit ci-dessus la catégorie "Command Link properties" ainsi que les diverses propriétés du contrôle.

Une propriété de dépendance s'écrit de la façon suivante :

```
[Category("Command Link properties")]
[Description("First line of text (link title)")]
public string Title
{
    get { return (string) GetValue(TitleProperty); }
    set { SetValue(TitleProperty, value); }
}

public static readonly DependencyProperty TitleProperty =
    DependencyProperty.Register(
        "Title", typeof (string), typeof (CommandLink),
        new PropertyMetadata("Title", OnTitleChanged));

private static void OnTitleChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)
{
    var cl = d as CommandLink;
    if (cl != null) cl.txtMain.Text = e.NewValue.ToString();
}
```

Code 41 - Une propriété de dépendance

Ici on voit que le code de la propriété `Title` modifie la propriété `Text` de `txtMain` qui est un `TextBlock`, celui utilisé pour le titre.

Le fonctionnel

Notre Command Link est assez malin. Il expose une propriété basée sur l'énumération `NavigationType` qui peut prendre les trois valeurs suivantes : `PageNavigation`, `BrowserNavigation` et `EventDriven`.

Le mode PageNavigation

Si le `CommandLink` est utilisé dans une application Silverlight utilisant la navigation SL, c'est à dire basée sur les classes `Frame` et `Page`, il pourra être utilisé pour changer de page dans l'application. La propriété `NavigateTo` indiquera l'URI de la page à atteindre. Dans ce mode notre UC cherche la page parente et utilise ses méthodes de navigation.

Le mode BrowserNavigation

Ici, le `CommandLink` sera utilisé pour appeler une page Web. `NavigateTo` sera donc une adresse Internet, les propriétés `Target` et `Options` permettent de fixer la cible, '`_blank`' pour une nouvelle page par exemple. Les options permettent d'afficher ou non la barre de statut, les menus de la page appelée, etc. Il s'agit là de pur HTML.

Le mode Event Driven

Il se peut que l'application souhaite utiliser le `CommandLink` dans un autre contexte ou d'une autre façon. Dans le mode `EventDriven`, un clic sur le titre déclenchera l'événement `OnNavigate` qui passera en argument une instance de `NavigationEventArgs`, une classe créée pour l'occasion qui contiendra toutes les propriétés utiles du `CommandLink` ainsi que la propriété `Tag` du composant (un contexte quelconque qui sera utilisé par l'application par exemple).

La méthode de navigation

Elle se présente comme suit :

```
 1: private void navigate()
 2:         {
 3:             switch (NavigationMode)
 4:             {
 5:                 case NavigationType.PageNavigation:
 6:                     {
 7:                         DependencyObject p = this;
 8:                         do
 9:                         {
10:                             p = VisualTreeHelper.GetParent(p);
11:                         } while ((p != null && (!(p is Page))));
12:                         if (p == null || (!(p is Page))) return;
13:                         var pp = (Page)p;
14:                         pp.NavigationService.Navigate(
                              new Uri(NavigateTo, NavigationUriKind));
15:                     }
16:                     break;
17:                 case NavigationType.BrowserNavigation:
18:                     {
19:                         if (NavigateTo == null) return;
20:                         HtmlPage.Window.Navigate(new Uri(NavigateTo,
            NavigationUriKind), NavigationTarget, NavigationOptions);
21:                     }
22:                     break;
23:                 case NavigationType.EventDriven:
24:                     {
25:                         var n = OnNavigate;
26:                         if (n == null) return;
27:                         n(this, new NavigationEventArgs
                              { Destination = NavigateTo,
                                Options = NavigationOptions, Tag = Tag,
                                Target = NavigationTarget,
                                UriKind = NavigationUriKind });
28:                     }
29:                     break;
30:                 default:
31:                     throw new ArgumentOutOfRangeException();
32:             }
33:         }
```

Code 42 - Méthode de navigation du UserControl

Selon le mode de navigation choisi le `UserControl` exécutera une séquence de navigation différente.

Conclusion

Créer des `UserControl` n'est pas une opération complexe, certains détails doivent être soignés et j'ai souligné dans ce billet les endroits où il faudrait passer un peu plus de temps pour rendre l'ensemble plus propre.

Nous avons vu aussi quelques limitations de ce mode de création de "composant" : ils ne sont pas "*templatables*", et ne sont pas facilement modifiables par l'utilisateur (par exemple ici changer l'icône n'est possible qu'en modifiant le contrôle lui-même).

C'est pour cela qu'il est bien plus intéressant de développer des Contrôles, ce qui a été abordé plus haut dans ce livre.

La mer, les propriétés de dépendance et les user controls…

La mer… un plaisir toujours trop court… Mais pour prolonger ce plaisir nous avons XAML !

Et comment mieux rendre grâce à la Grande Bleue qu'en fabriquant un User Control la mettant en scène ? Et bien c'est ce que nous allons faire, ce qui permettra ludiquement d'aborder un problème épineux concernant les propriétés de dépendance sous certains profils XAML. Mais d'abord le visuel :

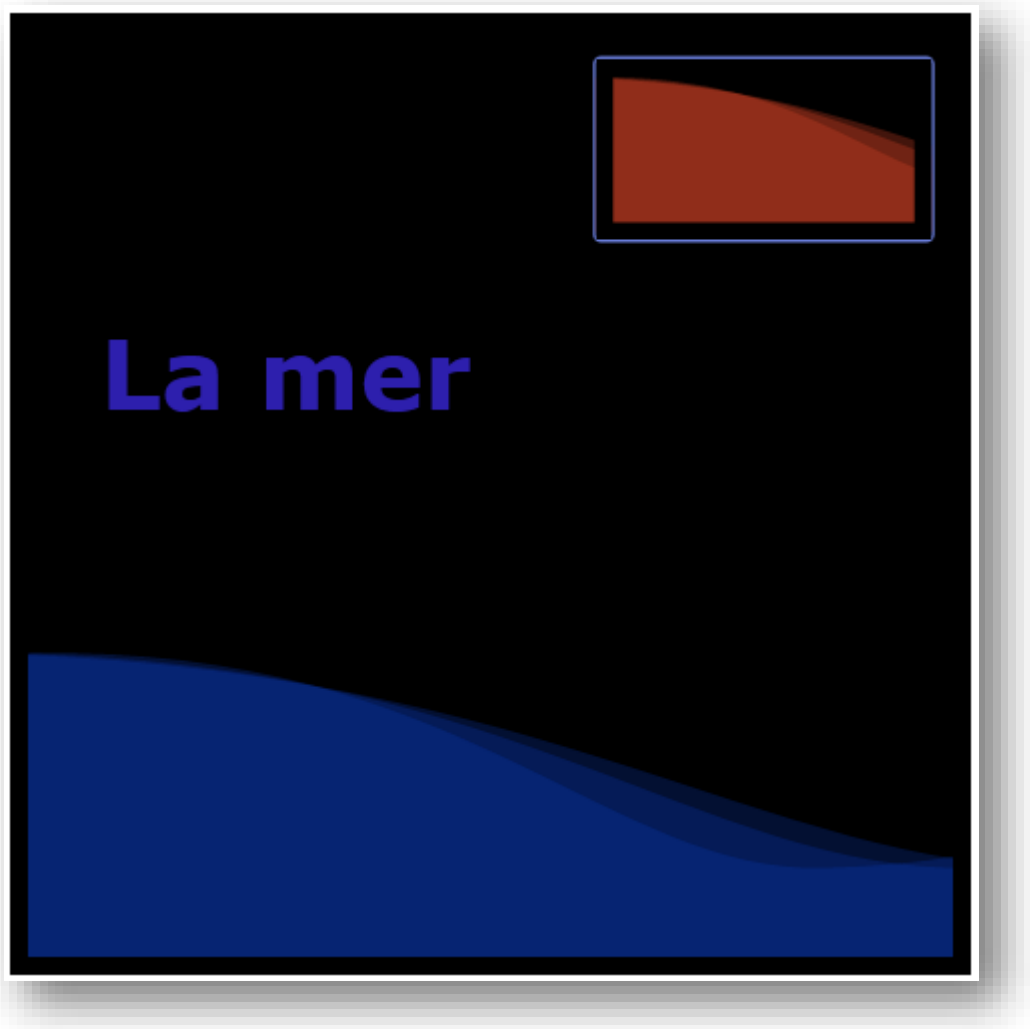


Figure 85 - Capture écran de l'animation "la mer"

La mer bleue, ou la mer rouge, au choix... et c'est bien ce choix qui va poser problème.

> *Bien entendu pour mettre en scène l'exemple sur Dot.Blog j'ai choisi Silverlight qui permet au lecteur de voir fonctionner le code sans besoin d'installer ou télécharger quoi que ce soit, et comme ce contrôle est animé une capture fixe ne permet pas vraiment de rendre justice au visuel. Vous pouvez accéder au billet original en cliquant sur le titre de ce chapitre et ainsi avoir accès à l'exemple live.*

La base du UserControl

Concernant le composant lui-même il s'agit d'un `UserControl` vide créé pour l'occasion. A l'intérieur un `Path` dessiné avec l'outil plume de Blend (une courbe de Béziers donc). Ce `Path` est dupliqué deux fois (ce qui donne donc 3 exemplaires au final). Les deux copies sont modifiées : l'une tassée en largeur, l'autre agrandie sur le même axe.

Un `StoryBoard` complète le tout : les trois vagues sont déplacées de gauche à droite, l'animation est mise en mode Autoreverse et boucle infinie. Pour un mouvement plus doux en début et fin j'ai ajouté un ease in/out.

Le tout est englobé dans une `Grid` pour bénéficier du *clipping*, le `LayoutRoot` étant une `ViewBox`, mais cette partie-là de la cuisine interne du composant n'est pas forcément la plus subtile. Pour terminer j'ajoute un rectangle sans bordure qui est placé en mode `Stretch` au fond du Z-Order, il servira à définir un éventuel background.

Jusqu'à là rien de bien compliqué, juste un peu d'imagination est suffisant.

Là où ça se complique c'est lorsqu'il faut gérer la couleur des vagues et celle du rectangle de fond...

Héritage et propriété de dépendance

Pour terminer correctement le `UserControl` il faut en effet penser à son utilisation. C'est à dire à ajouter des propriétés qui permettront à l'utilisateur du contrôle (le développeur ou l'intégrateur sous Blend) de modifier ses caractéristiques sans avoir besoin, bien entendu, de bricoler le code source du contrôle lui-même.

Ici, nous souhaitons pouvoir modifier la couleur des vagues et celle du fond. Parfait me direz-vous, cela tombe bien, un `UserControl` descend de `Control` qui lui-même définit deux propriétés on ne peut plus à propos : `Foreground` et `Background`. Il "suffit" de les surcharger.

En effet, "il suffit de".

Première chose, ces propriétés sont dites de dépendance (*dependency property* – sujet traité en début de ce livre). Ces propriétés ne sont pas définies comme ce qu'on nomme pour les différencier les "propriétés CLR", les propriétés habituelles.

Dès lors, et telles que fonctionnent ces propriétés spécifiques de XAML, pour les surcharger il faut passer par un système de métadonnées autorisant l'affectation d'une méthode *callback*. Dans cette dernière il est facile de répercuter sur le visuel les changements de valeur de la propriété. L'override d'une propriété de dépendance est donc assez simple. Sous WPF. Et c'est là qu'est le problème.

En effet, tous les profils XAML ne sont pas aussi complets que l'est WPF. Ce dernier est la version de référence, complète. Tous les autres profils à ce jour sont des sous-ensembles, de Silverlight à WinRT. Par exemple sous Silverlight qui sert à illustrer cet article tout à l'air tellement merveilleux qu'on en oublie que si tout le Framework .NET pouvait tenir dans quelques méga octets d'un plugin on se demande bien pourquoi l'installation du dit Framework pour une application classique (desktop) réclamerait des dizaines et des dizaines de méga octets... Y'a un truc. Y'a même une grosse astuce je dirais : forcément tout n'y est pas. En clair, le Framework Silverlight est un découpage chirurgical de haute précision pour donner l'impression que tout fonctionne tout en évitant 90% du code du Framework. Et il y a des petits bouts qui manquent, et parfois des gros ! Le problème se pose aussi avec Windows Phone ou WinRT parfois.

Concernant les propriétés de dépendance, l'équipe de Silverlight a implémenté le principal mais a laissé de côté les subtilités. Les métadonnées sont par exemple moins sophistiquées. Mais il n'y a pas que les données qui ont été simplifiées, les méthodes aussi. Et de fait il manque aux propriétés de dépendance la possibilité de les surcharger comme sous WPF.

Aie ! Comment réutiliser `Foreground` et `Background` définies dans `Control` et accessibles dans le `UserControl` s'il n'est pas possible de modifier les métadonnées et d'enregistrer notre propre *callback* ? J'ai longuement cherché car le problème est loin d'être évident à résoudre. Certains préconisent même face à ce problème de redéfinir vos propres propriétés. C'est tellement horrible comme solution que je m'y suis refusé. Comment avoir le courage de définir une couleur de fond et une couleur d'avant plan au sein d'un composant visuel qui affichera fièrement de toute façon `Foreground` et `Background` qui n'auront, hélas, aucun effet ? Quant à faire une réintroduction de ces propriétés (avec le mot clé "`new`"), n'y pensez pas, cela ne marche pas sur les propriétés de dépendance (en raison de leur déclaration bien particulière).

On peut se demander pourquoi l'équipe Silverlight, qui fait la chasse au gaspi un peu partout, s'est amusée à définir ces deux propriétés dans la classe `Control` si on ne

peut pas en hériter, sachant que `Control` ne sert à rien d'autre qu'à créer des classes héritées ? C'est assez mystérieux même si je devine qu'il s'agit d'un problème de compatibilité avec WPF.

L'Element binding n'est pas utilisable non plus, à moins de donner un nom au `UserControl` (je veux dire à l'intérieur même de la définition de celui-ci). Ce qui n'est pas acceptable car si l'utilisateur du composant change ce dernier, le binding est cassé. Pire si l'utilisateur tente de placer deux instances sur une fiche, il y aura un conflit de nom. Solution inacceptable donc. J'ai fait le tour de toutes les combines possibles, mais j'ai enfin trouvé celle qui fonctionne !

La Solution

La piste de l'Element binding n'était pas mauvaise. Le problème c'est qu'en XAML cela réclamait de pouvoir indiquer le nom de la source (ici le `UserControl`) alors même qu'à l'intérieur de la définition de notre contrôle il n'était pas question de lui donner un `x:Name` figé.

Mais en revanche, ce qui est possible en XAML l'est tout autant par code (et souvent inversement aussi d'ailleurs). Par chance la classe permettant de définir un Binding n'utilise pas les noms pour la source ni le destinataire. Elle utilise les noms des propriétés mais là on les connait et ils ne changeront pas. Du coup, en définissant le Binding dans le constructeur (ou plutôt dans le gestionnaire de l'événement `Loaded`) on peut référencer "`this`", c'est à dire l'instance du `UserControl`, sans connaître son nom. On peut donc créer un lien élément à élément entre la propriété `Fill` des `Path`'s et la propriété `Foreground` du `UserControl` (idem pour le `Fill` du rectangle et la propriété `Background` du `UserControl`).

Lorsqu'on compile tout ça et qu'on pose un composant "`LaMer`" sur une fiche on peut modifier la propriété `Foreground` et les vagues changent de couleur.

```
1:          public LaMer()
2:          {
3:              // Required to initialize variables
4:              InitializeComponent();
5:              Loaded +=
                  new System.Windows.RoutedEventHandler(LaMer_Loaded);
6:          }
7:
8:
9:          private void LaMer_Loaded(object sender,
                                      System.Windows.RoutedEventArgs e)
10:         {
11:             // l'astuce est là !
12:             var b = new Binding("Foreground")
                    { Source = this, Mode = BindingMode.OneWay };
13:             Vague1.SetBinding(Shape.FillProperty, b);
14:             Vague2.SetBinding(Shape.FillProperty, b);
15:             Vague3.SetBinding(Shape.FillProperty, b);
16:
17:             var bb = new Binding("Background")
                    { Source = this, Mode = BindingMode.OneWay };
18:             rectBackground.SetBinding(Shape.FillProperty, bb);
19:
20:             VaguesAnim.Begin();
21:         }
```

Code 43 - Binding par code

Pour le `Background` on fait pareil avec le rectangle. Mais, allez-vous me dire (si si, vous y auriez pensé, un jour !), pourquoi aller mettre un rectangle pour obtenir une couleur de fond alors même qu'il y a déjà une grille en dessous ? La grille possède aussi une propriété `Background`. Pourquoi, hein ?

La réponse est simple, j'ai forcément essayé, et ça fait un magnifique plantage avec une erreur dont le message fait peur en plus (du genre "anomalie irrémédiable dans cinq secondes tout va sauter !"). Le message d'erreur étant assez peu clair quant aux raisons du plantage j'ai fini par abandonner. Ce qui marche une ligne avant pour la propriété `Fill` des rectangles avec la propriété `Foreground` du `UserControl` ne fonctionne pas du tout pour le `Background` de la grille liée au `Background` du `UserControl`. Là, ce n'est pas une feature, je penche sérieusement pour un gros bug qui a peut-être disparu depuis que ces lignes ont été écrites, à tester donc.

Cela étant donné, j'ai donc ajouté un rectangle en fond pour qu'il puisse justement servir de ... `Background`. Et là ça passe. Je n'aime pas ce bricolage mais ça passe.

Ouf !

Certains profils XAML comme Silverlight présentent des différences mineures avec WPF. Mineures en quantité ou qualité apparente mais qui peuvent mener à des casse-têtes pour le développeur. Il faut alors explorer toutes les voies possibles pour trouver la parade. On y arrive souvent, au prix de quelques heures de recherche. La versatilité de XAML et du framework .NET est telle qu'il y a toujours un moyen.

Le code source du projet : LaMer.zip (62,54 kb)

Ecrire des Behaviors (composants comportementaux)

Blend est aujourd'hui un produit majeur associé à Visual Studio qui permet de travailler le visuel d'une application XAML (et même HTML pour WinRT). Depuis la version 3 Blend tend vers une plus grande simplicité de manipulation notamment pour les infographistes qui ne veulent / peuvent s'investir dans du code trop complexe. Cette ouverture ouvre de nombreuses voies aussi aux développeurs.

Les notions de Behaviors (comportements), Actions et Triggers (déclencheurs) sont des avancées significatives vers une programmation totalement visuelle. Le but étant bien de pouvoir ajouter des comportements (donc du code) visuellement sans programmation. Avec les outils d'il y a 20 ans comme VB Microsoft (enfin Alan Cooper qui vendit VB à Microsoft) inventait la notion de composant visuel qui se place sur une fiche par Drag Drop. Un bon en avant quand on connaît les méthodes qui ont précédé et quand on sait avec le recul quel succès a eu cette technologie et ses clones (la mode du RAD, l'EDI Delphi... jusqu'à .NET) ! Avec Blend (un vrai produit MS cette fois-ci) Microsoft innove dans une voie qui trace forcément ce que sera l'avenir : le "composant comportemental". On applique le "RAD" (mot plus très en vogue mais qui garde tout son sens) non plus seulement au visuel mais aussi au code.

Ainsi, aujourd'hui je vais vous parler des Behaviors (comportement).

Qu'est qu'un Behavior ?

Un Behavior est un **comportement**, c'est à dire un bout de code qui peut faire à peu près n'importe quoi. Avec ça, vous êtes bien avancé allez-vous dire ! Mais imaginez

tout ce que peut faire un bout de code autonome (ne dépendant pas d'états externes) et faites un joli paquet cadeau autour afin qu'il apparaisse dans une palette, comme un composant visuel et vous obtenez un Behavior.

Un Behavior est donc avant tout une sorte de *code snippet* qui porte un nom et qui est listé dans Blend comme le sont les composants.

Mais il s'agit d'une première approximation très simplificatrice. Car les Behaviors sont bien plus : Ils sont certes du code autonome encapsulé dans un composant apparaissant dans les palettes d'outils, mais surtout ils sont conçus spécifiquement pour apporter des comportements nouveaux à des objets visuels sans modifier ni même connaître le code de ces derniers !

Un Behavior existe en tant que composant mais pas pour lui-même, il n'a de sens que marié à un composant visuel, un `Rectangle` ou `Button` par exemple. Un Behavior s'accroche à un composant existant et c'est en jouant le poisson pilote vis-à-vis de ce composant qu'il lui amène le fameux comportement.

Un petit exemple pour clarifier : Imaginons qu'on souhaite pouvoir déplacer un objet par Drag'n Drop, il faudra gérer le `MouseDown`, le `MouseUp` et le `MouseMove` selon un scénario classique que je ne développerai pas. Imaginons maintenant que nous souhaitions apporter le même comportement à un autre objet. Il faudra de nouveau programmer aussi tous ces événements. Si maintenant nous encapsulons tout cela dans un Behavior, il suffira de déposer le dit Behavior sur tous les composants à qui on souhaite conférer ce comportement qui, *ipso facto*, en seront dotés sans aucune programmation. La réutilisation du code est totale. Côté utilisation de Blend cela permet même de développer une application sans être développeur pour peu qu'on dispose d'une bonne bibliothèque de Behaviors, d'Actions et de Triggers (je reparlerai des deux derniers dans de prochains billets).

Créer un Behavior

Créer un nouveau Behavior est aussi simple que de créer une nouvelle classe dérivant de `Behavior<T>`. Il faut néanmoins à trouver la classe mère... Selon les profils .NET / XAML celle-ci peut se trouver dans des assemblages différents voire des espaces de noms différents. On utilisera MSDN pour connaître la façon d'atteindre la classe mère en fonction de la cible visée (WPF, Silverlight, Windows Phone, WinRT...). Par exemple pour WPF ou Silverlight il faut ajouter un assemblage livré avec Blend alors que sous Windows 8.1 en WinRT il faut installer un paquet Nuget « Behavior SDK (XAML) ».

Une fois la classe créée, il ne reste plus qu'à surcharger deux méthodes : `OnAttached()` et `OnDetaching()`. C'est à partir de là qu'on peut programmer le comportement. Le Behavior peut connaître l'objet auquel il est attaché : `AssociatedObject`.

Un Behavior Exemple : RotateBehavior

Pour illustrer le propos j'ai cherché quel genre de petit Behavior je pourrais bien développer. Les classiques du genre ne me disaient rien (faire en flou quand la souris entre sur l'objet, ou bien changer sa transparence, etc). Mais, restons réalistes, pour un petit exemple on ne pouvait faire beaucoup plus compliqué. Je me suis donc dit qu'une transformation de type rotation accompagnée d'une petite animation serait parfaite pour illustrer le propos.

Pour d'évidente raison, cet exemple a été écrit en Silverlight afin qu'il puisse être utilisé directement sur Dot.Blog par les lecteurs du blog. En accédant au billet original (en cliquant sur le titre de ce chapitre plus haut) vous pourrez accéder à l'application exemple et jouer avec pour vous rendre compte du résultat visuel. Ici, une capture écran ne donne pas grand-chose…

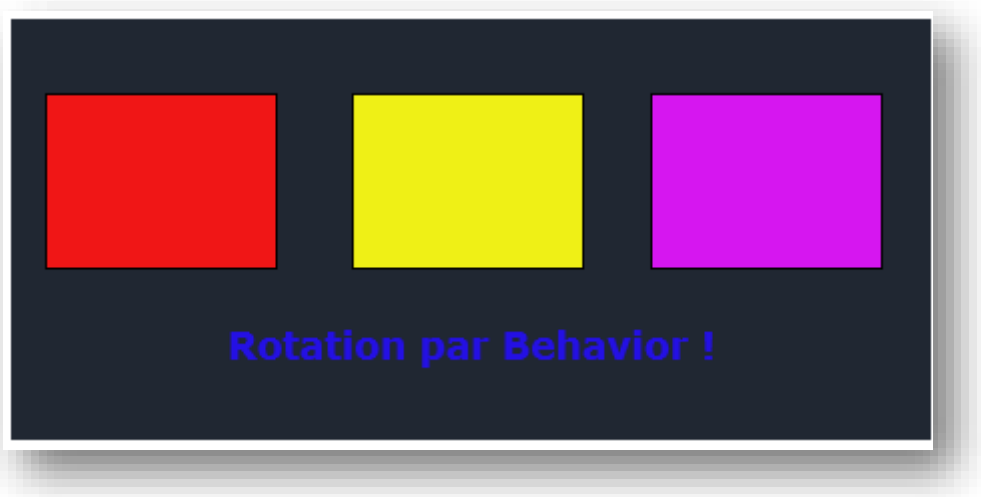


Figure 86 - Capture écran du RotateBehavior

A défaut de pouvoir tester visuellement l'effet directement dans le présent livre on peut observer l'arbre visuel dans Blend. On peut y voir le `RotateBehavior` accroché aux rectangles et à l'objet texte :

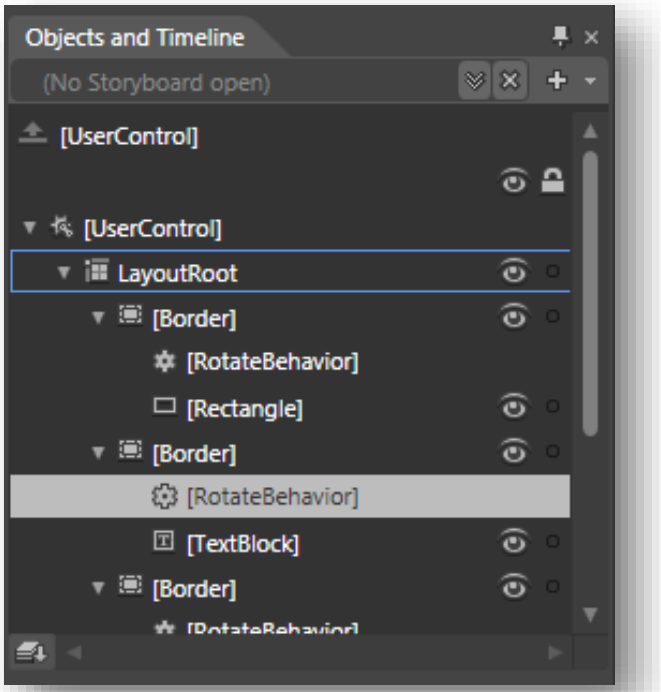


Figure 87 - RotateBehavior sous Blend

De même, pour le behavior sélectionné on peut voir les propriétés suivantes :

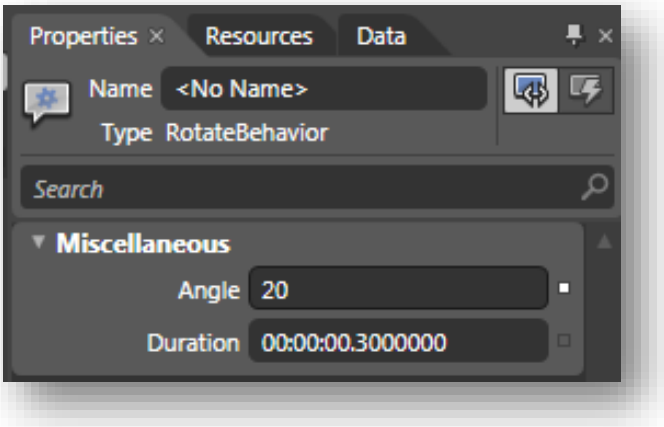


Figure 88 - Propriétés du RotateBehavior sous Blend

On note la propriété `Angle`, en degrés, et la propriété `Duration` qui définit la longueur de l'animation. Bien entendu, chaque `RotateBehavior` placé sur chacun des objets est paramétrable de façon séparée.

Le code exemple

Pour étudier l'exemple le mieux est de le faire fonctionner vous-mêmes et de vous amuser à éventuellement en changer le comportement. Je vous propose ainsi de télécharger le projet exemple (Blend 3 ou Visual Studio 2008 minimum) : Behavior_base.zip (61,84 kb)

Toutefois un peu de code ne peut pas nuire… voici la classe RotateBehavior :

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Interactivity;

namespace Behavior_base
{
    public class RotateBehavior : Behavior<UIElement>
    {
        public RotateBehavior()
        {
        }

        private RotateTransform rotate = null;
        private TransformGroup group = null;
        private Transform saveOriginal = null;
        private double angle = 0d;
        private Storyboard sbEnter = null;
        private Storyboard sbLeave = null;
        private TimeSpan duration = new TimeSpan(0,0,0,0,300);

        public double Angle
        {
            get { return angle; }
            set { angle = value; }
        }

        public TimeSpan Duration
        {
            get { return duration; }
            set { duration = value; }
        }
```

```
protected override void OnAttached()
{
    base.OnAttached();
    saveOriginal = AssociatedObject.RenderTransform;
    rotate = new RotateTransform {Angle = 0d };
    group = new TransformGroup();
    group.Children.Add(rotate);
    AssociatedObject.MouseEnter += AssociatedObject_MouseEnter;
    AssociatedObject.MouseLeave += AssociatedObject_MouseLeave;
}

void AssociatedObject_MouseLeave(object sender, MouseEventArgs e)
{
    if (sbEnter!=null) sbEnter.Stop();
    if (sbLeave!=null) sbLeave.Begin();
}

void AssociatedObject_MouseEnter(object sender, MouseEventArgs e)
{
    if (sbLeave!=null) sbLeave.Stop();
    AssociatedObject.RenderTransform = group;
    rotate.CenterX = (double)AssociatedObject.GetValue(
                          FrameworkElement.WidthProperty) / 2;
    rotate.CenterY = (double)AssociatedObject.GetValue(
                          FrameworkElement.HeightProperty) / 2;
    var da = new DoubleAnimation();
    sbEnter = new Storyboard();
    var du = new Duration(duration);
    da.Duration = du;
    sbEnter.Duration = du;
    sbEnter.Children.Add(da);
    da.To = angle;
    Storyboard.SetTarget(da,rotate);
    Storyboard.SetTargetProperty(da,
        new PropertyPath(RotateTransform.AngleProperty));
    sbEnter.Begin();

    var dal = new DoubleAnimation();
    sbLeave = new Storyboard();
    var dul = new Duration(duration);
```

```
            dal.Duration = dul;
            sbLeave.Duration = dul;
            sbLeave.Children.Add(dal);
            dal.From = angle;
            dal.To = 0d;
            Storyboard.SetTarget(dal, rotate);
            Storyboard.SetTargetProperty(dal,
              new PropertyPath(RotateTransform.AngleProperty));
        }

        protected override void OnDetaching()
        {
            base.OnDetaching();
            AssociatedObject.RenderTransform = saveOriginal;
            AssociatedObject.MouseEnter -= AssociatedObject_MouseEnter;
            AssociatedObject.MouseLeave -= AssociatedObject_MouseLeave;
        }
    }
}
```

Code 44 - RotateBehavior

La stratégie de développement du `Behavior` est ici clairement visible. Puisque notre Behavior va effectuer une rotation animée de l'`UIElement` auquel il sera associé, et comme nous désirons que ce comportement puisse être facilement paramétré il est normal de trouver la déclaration de deux propriétés, l'une pour la rotation à appliquer en degrés, `Angle`, et l'autre pour la rapidité de l'animation exprimée comme une durée (TimeSpan), `Duration`.

Ces propriétés ne déclenchent pas de notification de changement car cela est inutile, le Behavior se déclenchera quand la souris passera au-dessus de l'élément décoré, si nous changeons la durée ou l'angle de rotation, cela ne pourra de toute façon que s'appliquer à la prochaine action, celle en cours étant déjà lancée ou terminée. Et à chaque nouvelle action le code prend en compte la valeur courante des propriétés.

Autre aspect essentiel cette fois-ci : la programmation de `OnAttached` et de `OnDetaching`.

Le `OnAttached` est automatiquement appelé dès lors que le Behavior se voit lié à un composant visuel. C'est donc là l'occasion rêvée de détourner les événements de ce dernier dont nous avons besoin pour gérer notre nouveau comportement.

Il faut aussi prendre soin d'être en mesure de rétablir totalement le composant visuel ainsi « vampirisé » dans l'état où il se trouvait avant que le Behavior soit attaché.

On conserve donc toutes les valeurs qui seront changées pour être à même de les rétablir dans le `OnDetaching`, sinon gare aux bogues !

Le `RotateBehavior` va jouer sur la rotation du contrôle, donc sur son `RotateTransform`. De ce fait nous sauvegardons le `RenderTransform` de l'objet pour le rétablir si le Behavior est détaché par la suite.

Une fois cette sauvegarde effectuée nous pouvons créer un nouveau groupe de transformations que nous utiliserons pour effectuer la rotation.

On profite aussi de ce moment particulier pour s'abonner aux événements qui nous intéressent ici, à savoir le `MouseEnter` et le `MouveLeave`. On utilise des méthodes et non des expressions Lambda car nous avons besoin de pouvoir nous désabonner et pour cela il faut conserver le nom d'une méthode.

Dans le `OnDetaching`, après avoir appelé la méthode de base (c'est une surcharge) nous rétablissons le `RenderTransform` original de l'objet. Nous n'oublions pas non plus de nous désabonner des événements placés sous écoute dans le `OnAttached`. (Le code ci-dessus montre bien cette étape essentielle qui est absente, affreux bogue, du projet à télécharger, n'oublier pas de le rajouter !).

Le reste n'a rien à voir avec la programmation d'un Behavior en général mais uniquement de ce Behavior en particulier. Le `MouseEnter` est utilisé pour créer les animations d'entrée et de sortie de souris. L'une sera lancée à la suite, l'autre sera lancée uniquement sur le `MouseLeave`. Deux animations sont utilisées puisque la première, à l'entrée de la souris, va effectuer une rotation qu'il faudra bien « annuler » par son inverse quand la souris sortira de l'espace de l'objet visuel.

Conclusion

Les Behaviors offrent une nouvelle façon d'étendre et de personnaliser les contrôles visuels, toujours sans avoir à connaître ni modifier leur code source. Ils permettent aussi aux développeurs de fournir au graphiste des comportements, donc du code, exploitable sans connaissance du développement.

Plus loin, les Behaviors repoussent encore plus loin les limites de la réutilisabilité du code en proposant une stratégie adaptée non pas au code pur mais aux objets visuels.

S'il ne faut pas abuser des Behaviors, comme de toutes les bonnes choses, s'en passer totalement serait se priver d'un moyen simple de personnaliser les UI.

Les projets WPF ou Silverlight ont fait une grande consommation de Behaviors, les intégrant parfois même dans la logique de personnalisation des applications plutôt que d'écrire de nouveaux contrôles. Absent de WinRT dans Windows 8, le Behavior SDK apporte à Windows 8.1+ les mêmes possibilités. Utilisez-le pour enrichir vos interfaces facilement !

Un Behavior ajoutant un effet de réflexion (avec WriteableBitmap)

Dans le précédent billet je vous proposais d'écrire un Behavior effectuant une rotation animée d'un élément visuel, aujourd'hui je vous invite à la réflexion...

Je veux parler de l'effet de réflexion tant à la mode. WPF sait parfaitement le faire grâce notamment au brosses visuelles qui permettent facilement de "peindre" un contrôle avec la copie visuelle d'un autre objet (ou arbre d'objets). Hélas, les autres profils XAML ne possèdent pas forcément de brosse de ce type et pour obtenir l'effet désiré il faut dupliquer le ou les objets sources et leur faire subir une rotation, un flip, et l'ajout d'un masque d'opacité. Tout cela est assez fastidieux il faut bien l'avouer.

Mais grâce aux Behaviors il est possible de rendre tout cela automatique et avec l'aide des **WriteableBitmap** il n'est même pas nécessaire de dupliquer tous les objets !

J'ai déjà détaillé dans le billet précédent la technique de création d'un Behavior, je vous renvoie ainsi à cet exemple pour la base. Pour l'effet de réflexion, le mieux est de regarder ci-dessous l'exemple live réalisé en Silverlight (cliquez sur le titre de ce chapitre pour accéder au billet original et jouer avec l'exemple) :

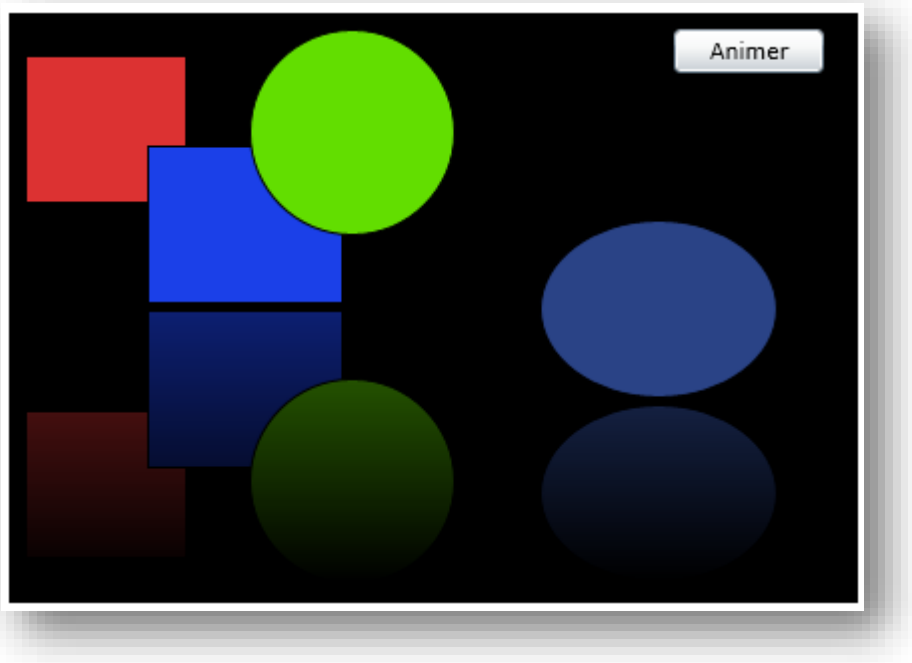


Figure 89 - Capture du behavior de réflexion visuelle

Pour créer cet effet nous allons utiliser la classe `WriteableBitmap`. Cette classe permet de créer des images de toute pièce.

Non seulement les WriteableBitmap permettent de créer un bitmap par code, mais en plus elle possède une méthode de `Render` qui est capable de faire le rendu de tout objet ou arbre visuel et de le placer dans le bitmap. Nous allons utiliser cette possibilité pour remplir un bitmap auquel nous appliquerons ensuite quelques transformations (renversement, échange gauche/droite, translation). Un masque d'opacité est ajouté pour l'impression de dégradé d'opacité.

Le Behavior fonctionne sur des sources de type `Canvas` uniquement, cela permet d'illustrer comment limiter un Behavior à une classe ou une branche de classes donnée, libre à vous de modifier cette limitation (en prenant en compte les implications). Une fois l'image du reflet créé elle est ajoutée en tant qu'élément au `canvas` (après avoir pris soin de supprimer la précédente image s'il y en a déjà une).

Le Behavior n'a pas de rendu en mode design sous Blend, c'est un petit défaut. De même l'effet de réflexion est créé une fois pour toute et n'est modifié que si le canvas source voit sa taille changer. Si vous créez dynamiquement des objets sur celui-ci, le reflet ne changera pas. Bien entendu on pourrait penser connecter le Behavior à l'événement de rendu du canvas `LayoutUpdated`. Cela eut été une solution élégante. Hélas on créerait une boucle infinie puisque notre Behavior ajoute un objet au canvas

ce qui déclenche le renouvellement de son rendu. On pourrait améliorer les choses en ne faisant pas le rendu dans le canvas source et en ajoutant une propriété au Behavior qui serait un objet de type `Image` (non enfant de la source) qu'on utiliserait comme sortie de rendu de l'effet de reflet. Mais cela poserait d'autres problèmes.

Toutes les idées sont envisageables et le code proposé n'est qu'un exemple. A vous de l'améliorer sans oublier de venir ici nous dire ce que vous avez fait !

Côté paramétrage le Behavior permet de modifier les points de départ et d'arrivée du dégradé du masque d'opacité ainsi que l'offset sur l'axe Y. Sans ce dernier le reflet serait collé au canvas source ce qui n'est pas forcément très beau. Par défaut un décalage de 3 pixels vers le bas décolle légèrement le reflet de l'original.

Le code source du projet complet que vous n'avez plus qu'à télécharger ici : ReflectionBehavior.zip (62,91 kb)

Behavior Collection

Les Behaviors sont des outils précieux dans le développement d'applications XAML, j'en ai parlé au chapitre précédent. On en trouve beaucoup sur le Web, surtout pour Silverlight mais pas toujours bien débogués et non centralisés. Il m'est venu à l'idée que tout ce potentiel ne pouvait être utilisé que si j'y mettais un peu d'ordre. Et comme je suis partageur je vous offre deux projets : une collection de behaviors que j'ai contrôlés, et une application de présentation originale, le tout avec le source.

My Behavior Collection

L'application exemple montre quelques un des behaviors, classés par catégorie. L'application de démonstration est simple à utiliser : en haut à gauche et droite se trouve une flèche, celle de droite pour avancer dans la démo, celle de gauche pour revenir en arrière. Les pages forment une boucle infinie.

Le test c'est ici : MyBehaviorCollection

Portabilité

La collection et son programme exemple sont écrits pour Silverlight. La raison est simple et toujours la même : les lecteurs de Dot.Blog peuvent ainsi tester directement l'exemple sans avoir besoin de télécharger ou installer quoi que ce soit. Bien entendu

dans le présent livre cet intérêt pratique disparait, mais en visitant le blog vous pourrez jouer immédiatement avec l'exemple…

XAML étant XAML cela ne change en rien l'intérêt de l'exemple présenté. Certains profils XAML comme WPF ou Silverlight utilisent juste une bibliothèque de code fournie avec Blend pour autoriser la création de Behaviors, WinRT par exemple utilise un paquet Nuget (le Behavior SDK). Il y a quelques nuances entre toutes ces versions de XAML dont il faudra tenir compte si vous désirez adapter le code à WinRT ou Windows Phone par exemple. Mais cela ne demandera pas un gros effort le principe et l'essentiel du code restant identique sauf éventuelles restrictions ponctuelles liées au profil XAML cible choisi.

Les behaviors

Comme je le disais, tous ne sont pas intégrés à la démonstration, soit par faute de temps, soit parce tous ne sont pas facilement démontrables de cette façon.

L'espace de nom `Od.Behaviors` est contenu dans un projet séparé qu'il suffit d'ajouter à une application Silverlight pour être en mesure d'utiliser tous les behaviors déclarés. Pour d'autres versions de XAML on repartira du code source fourni pour l'adapter.

Les Behaviors sont classés par thème ou catégorie :

Animation

Tous les behaviors (ou actions) de type animation.

- `AnimateVisibilityBehavior`; anime la propriété Visibility
- `BouncingPlaneBehavior`, un plan qui rebondit
- `RandomChildrenAnimationBehavior`; anime de façon aléatoire les enfants d'un conteneur
- `ScaleAnimationAction`; déclenche un changement d''échelle
- `ShakeBehavior`; Applique un petit tremblement à un objet
- `ShowHideFlipAction`; une page à deux faces
- `SwivelAction`; un autre effet visuel

Appearance

Tout ce qui concerne l'apparence des objets.

- `BringToFrontBeavhior`; amène un objet en avant plan
- `CenterAndScaleBehavior`; centrer et redimensionne un contrôle

- `ClipToBoundBehavior`; un classique bien utile pour clipper automatiquement
- `FadeInOutBehavior`; effectue un fade in ou out
- `ResizeBehavior`; affiche des poignées pour redimensionner l'objet
- `ScaleAction`; changement de taille
- `ToggleOpacityAction`; changement d'opacité entre deux niveaux sélectionnés
- `ToggleVisibilityAction`; changement de l'état Visibility
- `TransparencyBehavior`; Changement de l'opacité quand la souris est sur l'objet
- `WaterMarkBehavior`; Un watermark utilisé pour les zones de saisie

Commanding

Behaviors relatifs à la gestion de commande

- `CommandOnEnterBehavior` ; déclenche une commande sur la touche Enter
- `ConfirmCommandbehavior`; demande de confirmation pour l'exécution d'une commande
- `ExecuteCommande`; exécution d'une commande
- `UnloadedBehavior`; Déclenche une commande quand un objet est retiré du layout

Effects

Effets visuels.

- `BlurOverbehavior`; floute l'objet quand la souris est dessus
- `FlipAction`; effet de page à deux faces
- `Loop3DAction`; une animation de type 3D
- `MouseOver3DAction`; animation de type 3D sur le mouseOver
- `MouseProjectionAction`; modification de la projection
- `RotateBehavior`; permet la rotation de l'objet par l'utilisateur
- `ShiverBehavior`; effet visuel
- `Slide3DToMouseBehavior`; effet visuel

Functions

Des behaviors plutôt orientés fonction.

- `DragBehavior`; Support du drag
- `EndCueAction`; déclenche une action sur un marqueur placé en fin d'un media element
- `InstallOOBAction`; gestion de l'installation Out Of Browser (spécifique Silverlight)

- `OpenComboboxBehavior`; Ouvre la liste déroulante d'une combo sur la passage de la souris
- `PersistentPropertyBehavior`; permet de rendre une propriété persistante (choix utilisateur par ex.)
- `SelectOnFocusAction`; Sélection sur la prise de focus
- `SetFocusAction`; Prise du focus
- `ShowMessageBoxAction`; affiche un dialogue
- `SnappingSliderBehavior`; limite les positions d'un slider
- `ToggleFullScreenAction`; passer en mode plein écran (spécifique Silverlight)

Navigation

- `NavigationAction`; Navigue vers une URL sur le déclenchement du Trigger

Utilities

Quelques classes bien pratiques utilisées par certains behaviors :

- `AnimationMaker`; Créée et joue une animation sur un item donnée et une propriété double définie.
- `BindingListener`; ajouter du binding à des éléments non FrameworkElement
- `DelegatingCommand`; fournit une commande `ICommand` (dans l'esprit MVVM)
- `StoryboardHelper`; Fabrique une animation de changement de taille
- `TransitionHelper`; Fabrique une transition visuelle pour la Visibility
- `VisualTree`; Retourne les enfants d'un élément visuel, tous ou un en particulier (avec son nom). Récursif.

Le code

Télécharger le projet exemple et la collection de Behaviors ici :

BehaviorCollection.zip

Conclusion

Rien d'exceptionnel, mais ce qui est intéressant c'est d'avoir d'un seul coup tout le code de tous ces behaviors, normalement réputés débogués, de pouvoir les étudier et s'en inspirer pour créer les siens propres, quelle que soit la version de XAML cible choisie. Cette collection prend ainsi toute sa dimension par son côté pédagogique, chaque behavior pris à part n'ayant rien de bien exceptionnel.

Utiliser des éditeurs de valeurs dans les behaviors

Blend fournit un certain nombre de `ValueEditors` personnalisés dans l'inspecteur de propriétés pour simplifier l'utilisation des Behaviors. Mais il y a un petit bonus : l'équipe de Blend a aussi introduit certaines extensions afin de permettre l'utilisation de ces éditeurs dans les Behaviors tiers.

Plus précisément, le `ElementPicker`, le `StateNamePicker` et le `StoryboardPicker` peuvent être réutilisés en plaçant le `CustomPropertyValueEditorAttribute` sur la propriété que vous souhaitez utiliser avec l'éditeur.

Comme toujours avec XAML, vérifiez que les classes évoquées ici sont présentes dans le profil que vous utilisez. Si sur les bases XAML est le même, dans les sophistications on trouve des variations plus importantes…

L'énumération `CustomPropertyValueEditor` fournit les options disponibles :

Appearance	CustomPropertyValueEditor	Expected Property Type
SourceName button ...	Element	string
Storyboard Storyboard1	Storyboard	Storyboard
StateName VisualState	StateName	string

Figure 90 - CustomPropertyValueEditor

Notez que le sélecteur de `ElementPicker` et de `StateName` s'attend à être utilisé sur les propriétés de type `String`, et que le `StoryboardPicker` s'attend à être utilisé sur une propriété de type `Storyboard`. Il appartient à votre contrôle ou à votre Behavior de savoir quoi faire avec la valeur que vous obtiendrez en retour.

Voici un exemple d'implémentation utilisant ces attributs :

```
public class MyBehavior : Behavior<DependencyObject>
{
        [CustomPropertyValueEditor(CustomPropertyValueEditor.Storyboard)]
        public Storyboard MyStoryboard
        {
                get;
                set;
        }

        [CustomPropertyValueEditor(CustomPropertyValueEditor.Element)]
        public string MyElementName
        {
                get;
                set;
        }

        [CustomPropertyValueEditor(CustomPropertyValueEditor.StateName)]
        public string MyStateName
        {
                get;
                set;
        }
}
```

Code 45 - Utilisation de l'attribut CustomPropertyValueEditor

Un mot sur les métadonnées de conception

On voit ici que les attributs sont ajoutés directement aux propriétés et à la classe de runtime. C'est pratique, rapide, et cela fonctionne correctement.

Toutefois du point de vue conceptuel mais aussi purement pratique il ne s'agit pas d'une méthode à conseiller. En effet, les informations de design ne doivent pas être ajoutées aux classes de runtime. D'abord cela ajoute du code inutilisé au runtime et tout code mort doit être supprimé, ensuite cela entraine la liaison de certaines bibliothèques qui alourdissent inutilement l'application. Conceptuellement la bonne séparation du fonctionnement design / runtime d'une classe s'impose raisonnablement comme la séparation du code d'interface de celui du code métier par exemple.

En réalité il faut donc absolument séparer le code design. Cela se pratique depuis l'existence des composants et les premiers EDI offrant un design visuel (Visual Basic ou Delphi) et cela doit toujours se pratiquer aujourd'hui. Il y a certaines choses que le temps qui passe et que les plus grands bouleversements laissent inchangées, en voici une…

Les métadonnées de design doivent ainsi être stockées dans une DLL à part, dont l'extension est "`.Design`". Je traiterai certainement ce sujet d'ici quelques temps mais en attendant, et pour ceux que l'anglais ne rebute pas, vous pourrez trouver un article sur la question à cette adresse : [Writing a design time experience for Silverlight]

control. L'article est même complété d'un template pour Blend créant un projet DLL + design experience. Il s'agit d'un article couvrant Blend 3, le contenu est toujours juste techniquement, le template, en revanche, peut ne pas fonctionner avec les versions plus récentes de Blend (à tester !).

Vous trouverez aussi des informations intéressantes dans l'article suivant : A sample that shows the new Blend design surface extensibility. L'article traite l'exemple sous WPF mais cela est transposable à Silverlight car il s'agit là bien plus de Blend et de son EDI que de la cible Xaml choisie.

Conclusion

Ecrire des composants et des Behaviors n'est pas forcément très compliqué, mais les écrire bien, c'est à dire en fournissant un code propre et documenté, un fichier d'aide et des mécanismes en simplifiant l'utilisation au design demande plus de travail. Le résultat est plus "pro" aussi…

Xaml : lier des boutons radio à une propriété Int ou Enum

Le binding a ses raisons que la raison ne saurait connaître dirons certains… Il est vrai que parfois des choses très simples semblent impossibles à faire. Mais en y réfléchissant un peu la souplesse de Xaml et de C# permettent toujours de s'en sortir.

Le cas d'utilisation

Vous exposez dans un ViewModel une propriété de type `Int` (entier) ou `Enum` (énumération), directement, ou par le biais d'une Entité qui est elle-même exposée. Exemple simple : Le ViewModel de la page de gestion des utilisateurs d'une application expose un objet "*Utilisateur*" dont l'une des propriétés indique le niveau d'autorisation d'accès. Dans la base de données ce champ est représenté par un entier, le modèle Entity Framework le fait "remonter" comme tel. Il peut prendre les valeurs `0` (lecture seule), `1` (lecture / écriture), `2` (administrateur). C'est un exemple bien entendu.

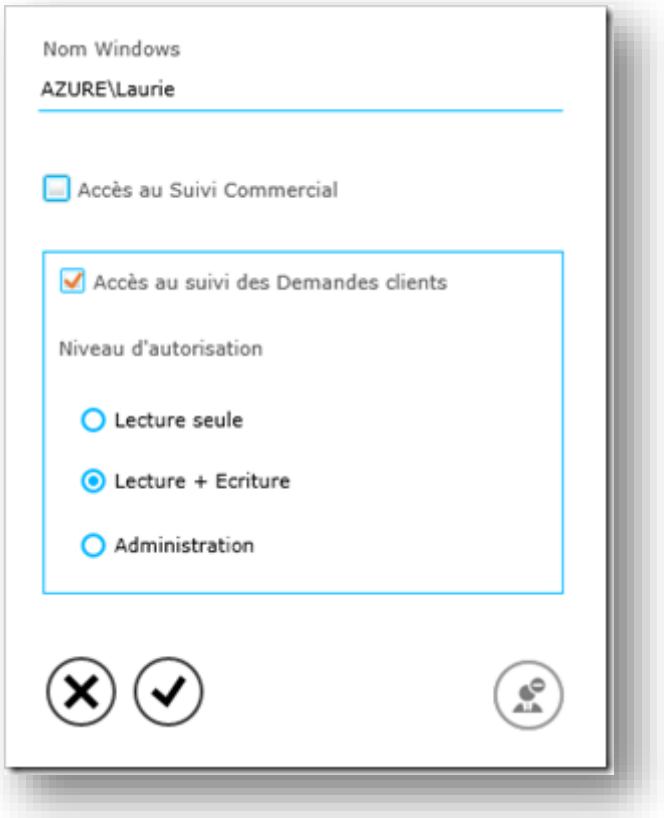


Figure 91 - Exemple de fiche saisie avec boutons radios liés à un Int

Le problème

L'exemple ci-dessus est un cas réellement simple et ultra classique donc. Que le modèle EF soit utilisé directement depuis une application WPF ou via un service WCF au sein d'une application Silverlight ou Windows 8 ne fait aucune différence. Qu'on utilise MVVM ou non, non plus.

Car notre problème n'est pas d'ordre architectural ni même lié à un framework ou un autre. Non, il est lui aussi d'une grande simplicité et parfaitement légitime : offrir une Expérience Utilisateur (UX) de bonne qualité.

Ainsi, cette propriété `Int` de l'objet `CurrentUser` de notre ViewModel pourrait fort bien être bindé à un `TextBox`. L'utilisateur n'aurait qu'à taper `0`, `1` ou `2` pour indiquer le niveau d'accès. Il lirait le niveau d'accès aussi sous cette forme « codée ».

Cela marchera mais ne sera pas satisfaisant pour l'UX : d'une part l'utilisateur devra connaître le transcodage ce qui rappelle l'informatique d'il y a 30 ans ou chaque opérateur disposait à côté de lui de listings de transcodage pour faire les saisies, et d'autre part puisque le champ sera directement connecté à l'objet visuel de saisie il

sera difficile d'effectuer des contrôles sur les valeurs introduites. On pourra certes utiliser l'annotation des données pour tenter de restreindre la plage de saisie ou d'autre mécanisme de validation, mais tout cela est bien lourd et bien peu ergonomique.

> *La première règle qu'il faut avoir en tête c'est qu'une UI bien programmée évite beaucoup de code !*

Si l'utilisateur n'a pas d'autres choix que de saisie les 3 possibilités de notre exemple rien ne sert d'aller bricoler les annotations des objets ou de mettre en place des mécanismes de validation complexes… Dans un tel cas un simple contrôle sera effectué au niveau de l'INSERT dans la base de données car c'est là que, *in fine*, les données doivent être validées. Si on laisse le loisir à chaque application de faire ses validations on prend le risque que deux applications divergent sur ces dernières. *La base de données doit jouer son rôle à part entière*. Mais en revanche *laisser l'utilisateur saisir des données qui seront refusées plus tard* au moment de leur envoi vers la base de données *n'est pas une bonne UX*…

> *Donc la règle devient : sécurisation des données au niveau de la base de données, "au cas où", mais surtout en même temps bonne conception de l'UI de l'application pour éviter que l'utilisateur ne saisisse des données erronées.*
> C'est ce couple de bonnes décisions qui forme le socle d'une UX réussie et d'une application correctement sécurisée.

Dans notre exemple chaque designer pourra trouver sa propre réponse originale, pour faire simple et efficace je vais choisir d'afficher la donnée `Int` sous la forme de trois radio boutons (voir la capture plus haut, issue d'un cas réel) puisqu'il s'agit de choix non cumulables. Les boutons radio sont parfaits pour ce scénario, à la fois pour représenter les données existantes (l'état de celui qui est `IsChecked=true` est bien visible) et pour saisir de nouvelles données puisqu'un clic sur l'un des boutons est évident et sans risque de mauvaise saisie (il annule tout autre choix).

Seulement nous voici arrivé au cœur du problème : *comment lier par un binding trois radio buttons à une même propriété de type* `Int` *(alors que les boutons disposent d'un* `IsChecked` *de type booléen) ?*

Il est bien entendu exclu de proposer une ComboBox avec les trois valeurs 0,1,2 dans la liste déroulante. Cela reviendrait à faire supporter le transcodage et sa signification à

l'utilisateur. De même que traduire ces valeurs en texte impliquerait de créer un mécanisme de conversion dans l'application entre texte et champ Int. Ce qui interdirait de plus de faire un binding direct entre la propriété de l'Entité source et les objets visuels… A moins d'ajouter, en plus, un convertisseur. Autant le faire de façon plus directe !

Les convertisseurs sont nos amis

Les premières fois que j'ai vu des applications exemples écrites en WPF par Microsoft à la sortie de cette technologie (mère de tout ce qui se fait aujourd'hui en Xaml rappelons-le) j'ai été très surpris et un peu décontenancé par un répertoire très fourni que toutes possédaient et qui s'appelait en général "*Converters*". Une foule de petits codes apparaissait ici…

C'est que Xaml et .NET avec leur Data Binding ouvraient la porte à tellement de puissance et de variabilité infinie qu'il fallait bien inventer un mécanisme permettant d'adapter les données aux besoins des contrôles disponibles pour leur affichage… Et comme ces contrôles étaient eux-mêmes en nombre potentiellement infinis (grâce à la grande facilité de création de nouveaux contrôle ou de contrôle utilisateur), il fallait bien trouver une solution efficace pour régler tous les cas de figure. Les convertisseurs de valeur ont bien un rôle essentiel à jouer dans la programmation Xaml…

Pour les données les plus simple et les scénarios les plus classiques, il s'avère qu'on peut heureusement écrire des applications entières sans presque jamais avoir à produire des tonnes de convertisseurs comme dans ces premiers exemples qui me reviennent à l'esprit et que j'évoquais plus haut. Mais dès lors qu'on sort de ces limites l'écriture d'un convertisseur de données s'avère souvent la meilleure solution.

> *Avec MVVM on peut parfois se passer de convertisseurs en "préparant" ou en "adaptant" les données au niveau des getters et des setters des propriétés exposées par le ViewModel, mais cela n'est pas toujours souhaitable même lorsque cela est possible. Une raison parmi d'autres : il n'est pas cohérent de faire manipuler par un VM des types qui n'ont de sens que pour l'affichage. Par exemple exposer une propriété de type Visibility est contraire à la séparation idéale entre code et UI, on doit exposer un booléen et l'UI doit utiliser un convertisseur.*

Que faut-il convertir ?

Dans le cas dans notre exemple nous devons convertir un entier en un booléen.

Cela n'est pas possible sans poser des règles d'interprétation propres à l'application. Quand un nombre donné sera-t-il considéré comme faux ou vrai ?

Il n'y a pas de réponse générique bien entendu. C'est une question de contexte.

Mais le problème est plus complexe qu'il n'y parait… Nous ne devons pas faire une simple conversion. Prenons un moment un autre exemple d'une telle conversion : le convertisseur classique `BoolToVisibility`, ou autre nom, qui consiste à convertir une propriété booléenne exposée par un objet ou un VM en une énumération `Visibility` pouvant être bindée à un objet visuel pour le montrer ou le cacher.

Ici les choses sont simples, selon le profil Xaml utilisé `Visibility` prend deux ou trois valeurs : visible, caché (hidden) ou supprimé de l'arbre visuel (collapsed). Certains profils Xaml ne gèrent pas forcément la nuance entre hidden et collapsed. Mais il est facile de dire que true = visible, et faux = hidden ou collapsed.

Pour revenir à nos boutons radio le cas est plus complexe : il ne suffit pas de convertir 0=faux, 1=vrai, car que fait-on de la valeur 2 ? Et que ferions-nous s'il y avait une valeur 3 voire 4 ou 5 ? (au-delà le principe des boutons radio serait à changer, toujours pour une UX de bonne qualité).

Il faut ainsi convertir une valeur entière en une valeur booléenne mais pour chaque radio bouton en fonction de sa signification "locale".

Le rôle des paramètres des convertisseurs

On oublie parfois tout l'intérêt du paramètre optionnel qu'offrent les convertisseurs de valeur. Ce paramètre est justement là pour personnaliser chaque conversion.

Pour notre exemple il suffira ainsi de passer la valeur qui doit être considérée comme "vraie" pour chaque radio bouton… Le premier aura un paramètre de valeur "`0`", le second "`1`" et le troisième "`2`". Le code du convertisseur n'appliquera donc pas une règle générale permettant de traduire un entier en booléen mais un test purement local permettant de dire si la valeur est égale ou non au paramètre passé…

Vous voyez la nuance ? Nous pouvons maintenant envisager de concevoir un code qui peut traduire l'entier en booléen, il ne s'agit plus que du résultat d'un test par rapport à la valeur du paramètre local de chaque binding, ce qui est parfaitement binaire : la valeur est ou non égale à celle du paramètre. Point. `IsChecked` peut ainsi être basculé à vrai ou faux sans état d'âme ni règle complexe… De plus le convertisseur à écrire devient générique et peut s'utiliser dans tous les cas similaires sans modification de son code…

> *Binding et paramètres : Hélas, et depuis toujours sans qu'aucune modification n'ait bizarrement été faite, le paramètre du convertisseur dans un binding n'est pas une propriété de dépendance… De fait il n'est pas possible de binder le paramètre à une valeur fournie par le VM ou à une simple ressource statique (SL) ou dynamique (WPF). Cela est vraiment dommage, surtout dans le cas des Enum comme nous le verrons plus loin.*

La réversibilité

Les convertisseurs de valeur fonctionnent dans les deux directions, en lecture depuis la propriété de l'objet source vers celle de l'objet visuel, mais aussi dans l'autre sens lorsque la propriété de l'objet visuel change et qu'elle doit impacter la valeur de la propriété liée dans l'objet source.

Très souvent les convertisseurs sont utilisés en sens unique car ils sont liés à des propriétés d'objets visuels que l'utilisateur ne peut pas modifier lui-même, rien ne sert donc d'écrire la conversion réciproque… Par exemple le `BoolToVisibility` évoqué plus haut ira cacher ou montrer un élément visuel à l'écran. L'utilisateur (sauf si cela a été prévu mais c'est une autre histoire) ne peut pas rendre de nouveau visible directement un objet caché. Dès lors rien ne sert de gérer la conversion du visuel vers la source.

En ce qui concerne nos boutons radio les choses sont différentes : ces éléments visuels serviront à afficher la valeur de la source, mais ils seront aussi utilisés pour saisir ou modifier cette dernière.

Le convertisseur qui sera créé devra ainsi prend en compte les deux sens de la conversion.

C'est bien entendu ici aussi en récupérant la valeur du paramètre de conversion qu'il sera possible de transformer le `IsChecked=true` en un entier, ce sera tout simplement la valeur du paramètre qui sera retournée…

Le code XAML

Le code Xaml correspondant à la capture en introduction est le suivant (débarrassé de la présentation et d'autres informations qui chargent les balises sans intérêt pour notre exemple ici) :

```
<RadioButton Content="Lecture seule" GroupName="RIGHTS"
    IsChecked="{Binding CurrentOperateur.NiveauAcces,
    ConverterParameter=0, Converter={StaticResource RadioButtonConverter},
    Mode=TwoWay}" />
<RadioButton Content="Lecture + Ecriture" GroupName="RIGHTS"
   IsChecked="{Binding CurrentOperateur.NiveauAcces,
   ConverterParameter=1, Converter={StaticResource RadioButtonConverter},
   Mode=TwoWay}"/>
<RadioButton Content="Administration" GroupName="RIGHTS"
  IsChecked="{Binding CurrentOperateur.NiveauAcces, ConverterParameter=2,
  Converter={StaticResource RadioButtonConverter}, Mode=TwoWay}"/>
```

Code 46 - Utilisation du convertisseur de valeur avec paramètre

Le groupe de boutons radio est défini de façon classique (`GroupName`), c'est le binding qui est un peu plus sophistiqué que d'accoutumée. On retrouve la propriété qui est liée "`CurrentOperateur.NiveauAcces`", `CurrentOperateur` étant ici une entité remontée via RIA Service et exposée par le ViewModel.

C'est une liaison à double sens (`Mode=TwoWay`) puisqu'elle va agir en lecture et en écriture de la valeur.

Les trois boutons sont liés à la même propriété du même objet. C'est le seul point moins habituel.

Un convertisseur est utilisé, "`RadioButtonConverter`". Il a été déclaré dans les ressources du `UserControl` en cours.

Enfin on note le paramètre "`ConverterParameter`" qui est égal à `0`, `1` ou `2` selon la valeur équivalente dans la base de données que prend la propriété (rappelons que l'entité expose cette valeur comme un entier, tel qu'il est défini dans la base SQL Server sous-jacente).

Le code du convertisseur

C'est lui qui va fournir maintenant le travail "intelligent" : convertir dans un sens et dans l'autre un entier en un booléen. Il va s'appuyer sur la valeur du paramètre passé dans le binding :

```
public class RadioButtonIntConverter : IValueConverter
  {
       public object Convert(object value,
            Type targetType, object parameter,
            System.Globalization.CultureInfo culture)
      {
          return parameter != null &&
               value != null &&
               (value.ToString() == parameter.ToString());
      }
       public object ConvertBack(object value,
                        Type targetType, object parameter,
                        System.Globalization.CultureInfo culture)
      {
          return value == null || parameter == null
                     ?  DependencyProperty.UnsetValue
                     : (bool) value
                          ?  int.Parse(parameter.ToString())
                          : DependencyProperty.UnsetValue;
       }
  }
```

Code 47 - Code du convertisseur de valeur

C'est le code d'un convertisseur tout à fait classique à la seul différence qu'il est adapté à un cas particulier, la conversion *entier<->booléen* en se basant sur la valeur du paramètre de conversion.

On notera l'utilisation de "`DependencyPropery.UnsetValue`" qui permet de retourner "quelque chose" même quand on a rien à retourner et ce sans brouiller le binding. Il s'agit là de l'écriture pour Silverlight, pour WPF on préfèrera utiliser "`Binding.DoNothing`" qui lui indique au binding de rien mettre à jour (cette valeur existe depuis le Framework 3.0 mais pas dans le profil .NET pour Silverlight).

Et avec un Enum ?

Avec un `Enum` le principe reste le même. Toutefois une difficulté supplémentaire viendra contrarier nos plans : le paramètre du convertisseur n'est pas bindable, de fait on peut écrire la valeur en chaîne d'une `Enum` mais guère plus.

Il faudra donc dans le convertisseur récupérer la chaîne et utiliser un `Enum.Parse()` en conjonction avec le type de la valeur qui est passé au convertisseur pour la transformer en une valeur testable.

Sinon le même mécanisme peut être utilisé (il faut bien entendu que la propriété source exposée soit de type `Enum` elle aussi…).

Conclusion

Finalement il s'agit juste d'utiliser un simple convertisseur de valeur extrêmement dépouillé… C'est quasi enfantin non ?

Non, vous avez raison, c'est loin d'être si simple. Quand on sait c'est très facile, mais pour trouver ce genre de solution on se gratte la tête parfois bien longtemps… Et encore faut-il faire l'effort de se gratter la tête plutôt que de se jeter sur son clavier pour écrire un code affreux qui sera bancale, lourd et difficile à maintenir…

Ainsi en va-t-il de la puissance de Xaml, ce langage graphique est tellement ouvert et versatile que tout est possible. La majorité de ce dont on a besoin dans une application est simple à faire. Mais dans certains cas, pourtant assez peu "tordus", il arrive qu'il faille réfléchir un bon moment pour trouver une solution à la fois simple et offrant une UX de bonne qualité.

La seule consolation pour ces cas un peu ardus c'est qu'on peut se féliciter d'avoir choisi de travailler en Xaml et en C#, qui malgré des difficultés ponctuelles comme celle démontrée ici, permettent avec un peu de créativité de s'en sortir et de ne pas rester coincé.

Deux règles pour programmer XAML

Des règles et des bonnes pratiques pour développer des applications XAML il en existe bien plus que deux, faire croire le contraire ne serait pas honnête. Ne serait-ce que par la richesse des EDI utilisés (Visual Studio, Blend…) il faut accumuler de l'expérience et mémoriser de nombreux patterns pour architecturer et designer correctement une application.

Alors, en vrac, voici deux règles qui me passent par la tête et que je voulais vous communiquer :

Programmation par Templates

Sous XAML lâchez les vieilles habitudes. L'une de celles-ci que je rencontre souvent est celle qui consiste à se jeter sur son clavier pour créer un `UserControl` (ou un `Control` par héritage) dès qu'on a besoin d'un nouveau composant qui semble absent de la bibliothèque.

Erreur. Méthode du passé.

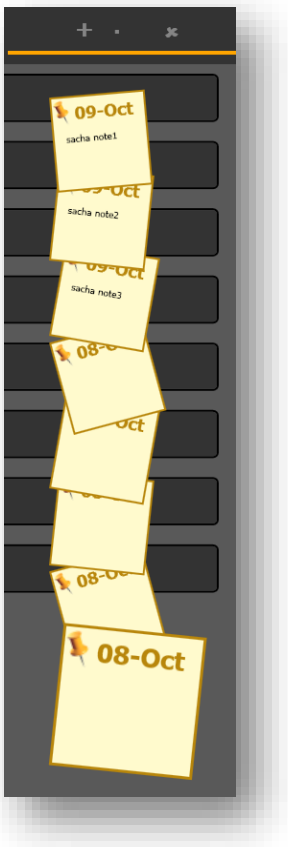
En effet, XAML implique des changements radicaux de mode de pensée et d'habitudes. Sous ces environnements, la majorité des besoins sont couverts par les composants existants, il "suffit" juste d'en modifier le template.

> *Nous sommes passés d'une programmation par héritage à une programmation par modèle (template).*

Deux exemples pour bien cerner ce que j'entends par là.

Sticky Note

Prenons pour premier exemple un cas d'école, déjà bien ancien mais en plein dans notre propos : sticky notes pour WPF (sticky notes listbox). Ce "composant" se présente comme une liste verticale de petites notes de type "post-it" légèrement décalées les unes par rapport aux autres.



Code 48 - Sticky Note, capture

L'effet est sympathique et donne un peu de fantaisie et de vie à ce qui ne serait qu'une liste rectangulaire dans une grille en programmation classique.

Ce magnifique "composant" n'utilise aucun code "classique". Inutile dans les sources de chercher la classe `StickyNotes` qui hériterait de `Control` ou même de chercher dans les fichiers de l'application le `UserControl` créé pour l'occasion.

Rien de tout cela.

Sticky Notes est créé uniquement en jouant sur le templating d'une simple… `Listbox` ! Une poignée de XAML et des templates, c'est tout.

Language button

Il y a un moment dans une application Silverlight je devais implémenter un bouton permettant de choisir depuis la page d'accueil la langue à utiliser (français ou anglais). Forcément on pense à un `ToggleButton` ou quelque chose d'équivalent. On pourrait aussi associer deux `RadioButton` dans une grille.

Mais le plus intéressant consiste à se demander "*quel control existant se rapproche le plus du comportement que je souhaite implémenter*". En y réfléchissant quelques

secondes on s'aperçoit assez vite qu'une `CheckBox` possède deux états stables (oublions ici l'état indéterminé). Ce composant comporte toute la logique et les états visuels permettant de gérer deux états.

Par défaut une `CheckBox` c'est une case à cocher avec un bout de texte devant ou derrière.

Le plus dur consiste à se l'imaginer comme deux drapeaux (français et US) côte à côte, celui qui est sélectionné étant à 110% de sa taille et l'autre à 50%. Par convention arbitraire et chauvinisme inconscient certainement j'ai considéré que `IsChecked = True` était le Français, à False l'anglais. En partant d'une `CheckBox` que j'ai totalement vidée de son contenu, et en ajoutant les deux drapeaux (et quelques autres ingrédients et animations via le VSM) j'ai obtenu un merveilleux "`ToggleButton`" sans jamais "*sous-classer*" le moindre contrôle. Juste en écrivant un template.

Le ViewModel de la page d'accueil offre une propriété `IsFrench` de type booléen qui est simplement bindée à la propriété `IsChecked` du `CheckBox` (en mode `TwoWay`) et l'affaire est jouée !

Règle 1 : De l'héritage au templating

La programmation XAML est une programmation visuelle qui s'effectue par templating. Créer des `UserControl` et encore plus sous-classer des contrôles existants devient quelque chose de rarissime.

Nous sommes passés de l'ère de la programmation objet par héritage à celle de la programmation visuelle par templating. Bien comprendre toute la signification de ce changement est un point primordial et un préliminaire indispensable pour comprendre cette technologie et donc la programmer intelligemment.

Programmation par Properties

Il s'agit du même genre de glissement, une pente douce mais dont la longueur finit par conférer une vitesse tellement grande à celui qui s'y laisse glisser que le décor n'a plus rien à voir avec celui qu'on a tout le temps d'admirer en balade à dos d'âne…

Dans l'exemple précédent on touchait du doigt cette nouvelle approche mais sans la mettre en exergue.

En effet, dans le pattern M-V-VM plutôt que de créer un code incompatible avec le visuel d'un côté pour ensuite créer de l'autre des tripotés de convertisseurs (programmation classique non M-V-VM sous WPF et Silverlight dans une moindre mesure) il semble bien plus simple d'exposer dans les ViewModels des propriétés directement exploitables par l'UI. Si adaptation des données il doit y avoir c'est le ViewModel qui s'en charge (directement si cela est ponctuel, ou via une classe de service si la chose doit être réutilisée ailleurs).

Dans l'exemple précédent du `Checkbox` transformé en `ToggleButton` de langue, aucun événement n'est programmé, aucun gestionnaire n'est écrit pour réagir au clic. Tout se joue dans le ballet automatique de `IsChecked` de la `CheckBox` et de `IsFrench` du ViewModel sous l'égide discrète mais indispensable d'un binding two way…

Quand l'utilisateur clique sur la `CheckBox` (enfin sur le `ToggleButton` avec les deux drapeaux) le composant sous-jacent bascule sa propriété `IsChecked` à vrai ou faux selon le cas. Comme cette dernière est liée à `IsFrench` du ViewModel, une propriété de type booléen aussi pour assurer la compatibilité des comportements, le ViewModel ne reçoit pas un événement mais l'une de ces propriétés (`IsFrench`) se voit modifiée. Ce qui déclenche le `Setter` de cette dernière. Ce dernier s'occupant de modifier le fichier de ressource utilisé pour puiser les chaines de caractères. De là et par une série de notification de changement de propriétés, il avertit en retour la Vue que toutes les propriétés de type texte ont été modifiées. La vue (et ses binding) y réagit en rafraichissant les affichages…

Toute cette mécanique s'est déroulée sans aucun gestionnaire d'événement, sans aucune programmation "classique" (en dehors du ViewModel et de ces propriétés gérant `INotifyPropertyChanged`, ce qui peut être automatisé ou simplifié en utilisant un framework MVVM généralement).

Règle 2 : de l'événementiel au binding

La seconde règle d'or à bien comprendre pour tirer totalement partie de XAML est ainsi d'opter pour **un modèle de développement de type Model-View-ViewModel se basant presque exclusivement sur des couples de propriétés mis en relation via binding.**

On est passé de l'ère du développement dit "événementiel" des premières versions de Windows à ce qu'on pourrait appeler la "programmation par Binding" ou par "properties".

Conclusion

Pour résumer :

Règle 1 - On cherche à templater des contrôles existants sans créer des UserControl ni dériver des contrôles existants.

Règle 2 - On base la dynamique de l'application sur le binding entre propriétés et non plus sur les gestionnaires des événements des contrôles.

Si vous avez déjà fait vôtre ces règles alors vous allez devenir, si ce n'est pas déjà le cas, de bons développeurs Xaml très recherchés !

Si vous n'aviez pas encore vu les choses sous cet angle-là, je serai très heureux si par chance j'ai réussi à tirer le voile qui vous empêchait de les voir ainsi. Vous ne ferez qu'entrer plus vite dans la catégorie précédente !

Style ou Template ?

Durant mes formations je m'aperçois que certaines notions qui, avec l'habitude me semblent "basiques" ont parfois plus de mal à être saisies que d'autres. Ainsi en est-il de la nuance entre **Styles** et **Templates**.

Il me semble donc important de clarifier les choses car si les deux concepts sont proches ils n'en sont pas moins différents et servent des objectifs tout aussi différents.

Les Templates

Commençons par eux. Les Templates permettent de modifier la **forme** d'un composant visuel. C'est grâce à un Template que vous pouvez rendre un bouton rond, un radio bouton rectangulaire.

Le Template s'intéresse à la forme.

Les Styles

Les styles sont des collections de couples propriété / valeur.

C'est dans un style qu'on fixera la couleur de la fonte, sa taille, la couleur du fond, etc, pour qu'un contrôle visuel corresponde à la charte graphique de l'application. Un

Style ne peut pas être utilisé pour rendre un bouton rond, mais il doit être utilisé pour le rendre bleu (ou vert ou ce que vous voulez).

Un style s'intéresse à l'aspect final.

Pourquoi cette confusion entre styles et templates ?

Elle ne naît pas de rien. Les choses peuvent être confuses car si on s'en tenait à ce que je viens de dire les choses seraient certainement plus claires (quoi que… la différence entre "forme" et "aspect final" semble un peu spécieuse il faut l'avouer).

Un Style peut exister sans Template. L'inverse est vrai. Mais un Style peut contenir un Template, l'inverse n'étant pas vrai. En effet, dans un objet le Template est représenté comme une propriété tout à fait ordinaire. Un Style peut donc parfaitement, dans sa liste de couples propriété / valeur, fixer la valeur de la propriété Template…

Pourquoi encapsuler le Template dans un Style ?

Il y a de bonnes raisons de faire de la sorte, regardons de plus près justement la séparation des tâches entre Templates et Styles…

Un Template dans un Style

Vous allez très vite comprendre (enfin je l'espère !). Supposons que nous voulions créer un bouton rond, histoire d'être original (je plaisante…). Comme nous sommes des développeurs (ou des designers) professionnels, nous sommes constamment aiguillonnés par l'envie de faire bien et surtout de faire réutilisable (un bon développeur est un fainéant, ceux qui n'ont pas cette immense qualité sont voués à être des pisseurs de lignes toute leur vie).

Ainsi, bien qu'il faille un bouton rond et bleu pour notre application, nous allons réaliser un bouton "générique" qui pourra être aussi bien bleu que vert ou jaune.

Dans le Template, là on nous allons rendre le bouton rond, on comprend facilement que si je place la couleur du fond à bleu cette dernière sera figée. Je vais donc concevoir un bouton rond sans couleur (ou plutôt qui prépare le terrain à la réception d'une couleur quelconque, ce qui est assez différent en réalité).

Mais on m'a demandé un bouton rond bleu (disons que cela soit la charte graphique de l'application)… Je place alors mon Template dans un style (si ce n'est pas déjà fait) et c'est là que je vais indiquer que le `Background` du bouton est bleu. Et c'est depuis

les ressources de style qu'on puisera le bouton bleu, le Style amenant à la fois la forme (le Template) et l'aspect (bleu). Facile non ?

Oui mais… cela ne marche pas. J'ai beau choisir un `Background` bleu dans le Style, le bouton rond ne change pas d'aspect…

En effet, en créant un bouton rond, j'ai bien été obligé de faire le ménage de tout ce qu'il y avait dans le bouton d'origine. Notamment tous les rectangles qui définissent la forme de l'objet. Parmi eux se trouvaient certainement celui qui était "considéré comme" le fond. Sans lui, la propriété `Background` originale de la classe `Button` ne sait plus à quoi attribuer la nouvelle couleur !

La cassure du templating

En templatant le bouton j'ai bien entendu été obligé de partir de zéro, rien ne pouvant être conservé (à part le `ContentPresenter` mais on ne le garde pas puisqu'on part d'un template vide en général pour ce genre de travail). Le problème serait le même si j'avais modifié un `Template` existant ou avais touché de quelque façon que ce soit le visuel d'un composant via templating. *Il s'agit d'un cas général, et non pas d'une difficulté ponctuelle*.

Seulement malgré la grande souplesse de Xaml il n'a pas de magie non plus… Le code original du bouton savait à quel objet il fallait changer la couleur quand la propriété `Background` était modifiée. Mais maintenant que j'ai supprimé tous les rectangles originaux et que j'ai créé un visuel original utilisant d'autres briques (des ellipses, des images png, des courbes de Béziers, etc) comment diable la propriété `Background` pourrait-elle agir sur mon bouton ? Quelle partie graphique (ou quelles parties graphiques) est (sont) impliquée(s) quand la propriété `Background` est changée ?

C'est ce que j'appelle la "*cassure du template*".

> *Le template est une blessure, une distorsion entre un code original compilé pour un visuel donné et la souplesse de Xaml qui permet de modifier du tout au tout ce visuel…*

En effet, la classe `Button` expose toujours une propriété `Background`, cela est géré par une propriété de dépendance dans le code C# de la classe `Button`. Un code compilé

et immuable. Les changements de couleur de fond étaient correctement gérés par ce code qui savait quel rectangle jouait le rôle de fond.

Mais désormais le code de la classe `Button` se trouve dans l'impossibilité d'effectuer cet appariement et aucun mécanisme automatique ne peut être conçu pour le faire. Graphiquement tout est possible, graphiquement n'importe quoi peut être utilisé pour faire "le fond", notion qui elle-même n'a pas de définition très nette (on parle du fond du rectangle définissant le `UserControl`, le fond d'un élément de ce `UserControl`, de plusieurs éléments ?)

Il faut donc **réparer ce qu'on a cassé** !

Docteur Template Binding

Il nous faut un remède, cela est sûr. Un Template qui laisse des propriétés de l'objet original "en l'air" avec tous les "fils débranchés" n'est pas un Template fini.

C'est là qu'intervient ce bon docteur "Template Binding". Et oui, cela sert à ça le Template Binding : à réparer les cassures du templating ! On peut lui trouver d'autres usages mais celui-ci est en tout cas le plus fréquent.

Je reprends ainsi le `Template` de mon joli bouton rond et je sélectionne l'ellipse que _je_ considère comme étant le "*fond*" de mon objet. Je vais sur sa propriété `Fill` et là je créé un Template Binding avec la propriété `Background` du bouton.

Je viens de réparer la cassure du templating…

Et ce qui est vrai pour la propriété `Background` du bouton l'est aussi pour toutes les autres propriétés que mon nouveau design aura ainsi "débranchées" du code original. Et, bien entendu, le bouton n'est qu'un exemple, cela est valable pour tout contrôle dont on modifie le Template.

Retour au Style

C'est bien joli ces histoire de cassures et de template binding, mais maintenant j'ai un bouton avec un fond noir (ou autre mais pas bleu, sauf avec de la chance et cela ne « compterait pas ») quand je suis en mode templating ! Et quand j'applique mon template à un nouveau bouton c'est pareil ! Au secours !

Pas de panique… Fixer la couleur du `Background` de notre bouton c'est justement le job du `Style`, pas du `Template`. Il faut donc modifier le style entourant le `Template` ou

bien créer un `Style` qui intègre le `Template` et passer en modification du `Style` lui-même.

Là, je vois toutes les propriétés de la classe bouton et je peux choisir le bleu qui a été défini pour les boutons de l'interface de l'application (bleu que je puiserai d'un dictionnaire de ressources définissant les couleurs de la charte graphique de l'application d'ailleurs car, comme je le disais, en bon professionnel je cherche à faire les choses bien jusqu'au bout).

La cassure est définitivement réparée cette fois ci grâce à un niveau hiérarchique supérieur : le Style qui se place au-dessus de notre Template.

Un Template qui a du Style !

Désormais, pour poser un bouton rond et bleu sur une page de notre application il suffira soit d'appliquer le `Style` (qui aura été nommé) à tout bouton existant, soit, plus simplement, sous Blend, de faire un drag'drop du `Style` sur la surface de travail, ce qui créera une instance de bouton avec le Style correspondant en une seule opération.

Bien entendu, si mon Template à du style c'est parce qu'un Style peut contenir un Template !

Conclusion

Un Template gère la forme, le visuel "profond" de l'objet, sa silhouette.

Le Style gère des couples propriété / valeur et permet de fixer l'aspect final d'un contrôle comme une hauteur, une largeur, un fond, une couleur d'avant plan, le nom de sa fonte, la taille de celle-ci, etc. Il faut voir cela comme une "feuille de style".

Mais un Style peut très bien "encapsuler" un `Template`. D'abord parce que `Template` est une propriété comme une autre et qu'un `Style` peut donc affecter une valeur à cette dernière. Ensuite parce qu'il est généralement judicieux d'insérer un `Template` dans un `Style` pour justement réparer les "cassures" du Templating et former un tout même au niveau code.

Le DataTemplateSelector

Les différents profils XAML c'est tout Xaml de père en fils, la seule différence, du point de vue programmation, c'est que certains fistons ont un costume plus petit que le papa (WPF). Et quand on a besoin d'un grand costume, celui de Silverlight ou d'autres profils XAML est parfois un peu juste aux entournures. C'est le cas principalement pour certains scénarios dits "avancés" comme ceux mettant en scène le `DataTemplateSelector`. Comment y remédier ?

DataTemplateSelector

Dans DataTemplateSelector il y a DataTemplate et Selector (même ceux qui ne parlent pas l'anglais l'auront vu j'en suis sûr !). Il s'agit donc de pouvoir sélectionner des `DataTemplates`. La fonctionnalité est bien sûr présente dans WPF et elle a été portée dans le profil XAML de WinRT.

DataTemplate

Un `DataTemplate`, pour mémoire, est une ressource décrivant la mise en page d'un item au sein d'une collection d'items affichés par une `ListBox` ou autre contrôle du même type. La puissance de Xaml se révèle dans ce genre de possibilité d'ailleurs, surtout si, pour le visuel, on s'aide de Blend.

Donc vous avez une `ListBox` (par exemple) et pour définir la façon dont s'affiche chaque item vous créez tout simplement un `DataTemplate`. Sous Blend cela se fait entièrement visuellement. Et qui dit `DataTemplate` dit Binding puisque les éléments affichés par le `DataTemplate` seront issus d'un contexte particulier : l'item en cours d'affichage.

Pourquoi vouloir sélectionner un DataTemplate et dans quel contexte ?

Imaginons une application gérant des *Prospects* et des *Clients*. Les deux classes dérivent ou non d'une classe commune, cela n'a pas d'importance. Ce qui compte c'est qu'elles ont des points communs (comme la propriété `NomDeSociété`) mais aussi des différences (`DateDernièreFacture` pour le Client et `ARevoirLe` pour le Prospect, toujours par exemple).

Supposons maintenant qu'à un moment donné, dans mon application je doive afficher une liste (filtrée et triée par des procédés qui ne nous intéressent pas ici) qui mélange des clients et des prospects.

Si je créée un `DataTemplate` proposant un `TextBlock` bindé à la propriété `NomDeSociété`, tout ira bien car il se trouve que les deux classes proposent cette propriété (encore une fois même si elles ne dérivent pas d'une classe commune, c'est le nom de propriété qui compte pour le binding). J'aurai ainsi une belle liste de noms de sociétés.

Mais visuellement je ne donnerais pas la possibilité à l'utilisateur de voir quels sont les clients et les prospects. Je ne pourrais pas non plus afficher la date de dernière facture pour les clients (car le binding échouera pour les prospects, laissant un libellé suivi d'un "vide"), pas plus que je ne pourrais indiquer la date de prochaine visite pour les prospects (même raison, même effet).

C'est une situation fâcheuse à plus d'un titre.

D'abord sur le plan du cahier des charges : si on me demande d'afficher ces informations complémentaires selon le cas (client ou prospect) il va bien falloir que je respecte le contrat...

Ensuite sur le plan de l'UX l'affichage du seul nom de société n'est vraiment pas très informatif. La distinction prospect / client et les informations supplémentaires qui pourraient être données simplifieraient grandement la tâche à l'utilisateur. Cela rendrait le logiciel plus convivial et plus "lisible".

Bref, voici une situation simple dans laquelle il serait bien agréable de pouvoir disposer de `DataTemplates` personnalisés et qu'au runtime tout cela se débrouille tout seul sans avoir à écrire plein de code.

Aribba el DataTemplateSelector !

Tel un Zorro qui arrive toujours à pic, WPF nous offre un procédé assez simple et élégant, le `DataTemplateSelector` heureusement porté aussi sous WinRT. Je vous laisse consulter MSDN tous seuls (cliquez sur le nom de classe dans la phrase précédente) je ne vais pas traduire la documentation Microsoft vous ne venez pas sur Dot.Blog pour cela...

Mais ¡caramba! Le fiston Silverlight et donc Windows Phone 7 et 8 ne supportent pas cette mécanique ... ¡Qué lástima!

Trêve de lamentations hispanisantes, voyons comment contourner ce problème.

Comment fait WPF ?

Avant de se lancer à l'assaut de son clavier et de dégainer l'artillerie lourde il est toujours préférable dans un tel cas de regarder comment fait WPF. D'abord il y a la documentation qui explique, puis il y a des tutors, et enfin il y a Reflector ou équivalent qui permettent de décompiler le Framework et d'apprendre du code existant.

Je vais vous épargner cette démarche en résumant ici l'essentiel :

Création des DataTemplates

Comme il s'agit de fabriquer des `DataTemplates`, imaginons les deux cités plus haut (client / prospect). Pour l'exemple ils seront rudimentaires :

```
<DataTemplate x:Key="ClientTemplate">
  <StackPanel>
   <TextBlock Text="{Binding NomSociété}"/>
   <TextBlock Text="{Binding DateDernièreFacture}"/>
  </StackPanel>
</DataTemplate>

<DataTemplate x:Key="ProspectTemplate">
  <StackPanel>
   <TextBlock Text="{Binding NomSociété}"/>
   <TextBlock Text="{Binding ARevoirLe}"/>
  </StackPanel>
</DataTemplate>
```

XAML 58- Définition des DataTemplates de l'exemple

Les DataTemplates seront définis dans un dictionnaire de ressources séparés ou localement, peu importe ce n'est pas la question ici.

Ce qui compte c'est que maintenant nous possédons deux `DataTemplates`, chacun ayant une clé différente et, bien entendu, un visuel différent (même si ici la différence n'est pas énorme).

Création du DataTemplateSelector

Ici il faut du code. En effet, la prise de décision, le choix entre l'un ou l'autre des DataTemplates (il pourrait y en avoir 5 ou 50 le procédé n'est pas limité à une alternative) peut dépendre de conditions aussi diverses que variées. Dans notre exemple nous utilisons le type de l'item pour choisir le template, mais dans un autre

contexte les items peuvent être de même type et les templates seraient alors choisis en fonction de conditions, de valeurs précises de certaines propriétés. On peut envisager n'importe quoi.

Dès lors le moyen le plus simple, plutôt que d'inventer un codage pseudo-visuel qui aurait très vite ses limites, WPF laisse le développeur écrire un peu de code pour effectuer la sélection du template. C'est clair, ouvert et simple.

Pour cela on écrit une classe dérivant de `DataTemplateSelector` :

```
public class ClientProspectTemplateSelector : DataTemplateSelector
{
  public DataTemplate ClientTemplate { get; set; }
  public DataTemplate ProspectTemplate { get; set; }

  public override DataTemplate SelectTemplate(object item,
    DependencyObject container)
  {
    return item is Client ? ClientTemplate : ProspectTemplate;
  }
}
```

Code 49 - Le code du DataTemplate Selector

Comme on le voit, rien de sorcier. On créée une classe dérivée de `DataTemplateSelector` dans laquelle on *override* la méthode protégée `SelectTemplate`. Celle-ci propose en paramètre une référence vers l'item concerné et vers le conteneur. Ce dernier ne nous intéresse même pas. Un simple test permet de retourner le `DataTemplate` *ad hoc*, selon que l'item est de type `Client` ou non.

On remarquera bien entendu les deux propriétés que nous avons déclarées, toutes les deux de type `DataTemplate` (une pour le template client, l'autre pour les prospects). Elles stockeront les références vers les templates alternatifs, ceux retournés justement par `SelectTemplate`.

Instancier le sélecteur

Nous disposons des deux `DataTemplates` et d'une classe dérivée de `DataTemplateSelector` qui sait à la fois stocker des références vers ces templates et retourner celui qui convient pour tout item qui lui est présenté.

Il faut maintenant instancier notre sélecteur. Le moyen le plus simple est d'utiliser la faculté de Xaml de créer une instance automatiquement liée à une clé dans une ressource :

```
<local:ClientProspectTemplateSelector
   ClientTemplate="{StaticResource ClientTemplate}"
   ProspectTemplate="{StaticResource ProspectTemplate}"
   x:Key="clientProspectTemplateSelector" />
```

XAML 59 - Création de la ressource locale pour le Selector

Le namespace "`Local`" sera défini dans le `UserControl` en cours (la fenêtre en général) pour pointer vers la déclaration du sélecteur. L'ensemble de la balise sera intégrée aux ressources locales de la fenêtre (mais on peut envisager une déclaration globale pour tout le projet, c'est une question de stratégie qui est hors sujet ici).

Utiliser le sélecteur

Ne reste plus qu'à utiliser notre sélecteur. Pour cela prenons une `ListBox` ou une `ListView` :

```
<Grid>
    <ListView ItemsSource="{Binding ElementName=This, Path=TestCollection}"
              ItemTemplateSelector=
               "{StaticResource clientProspectTemplateSelector}">
    </ListView>
</Grid>
```

XAML 60 - Utilisation du TemplateSelector dans une ListView

On suppose ici que dans l'objet en cours (disons `Window1` de type `Window`, nous sommes sous WPF) nous avons déclaré une propriété publique `TestCollection` qui contient la liste des clients et des prospects.

Ce qui nous intéresse plus particulièrement ce n'est pas l'`ItemsSource`, mais l'`ItemTemplateSelector`. Une propriété disponible sous WPF qui permet justement d'indiquer non pas un `DataTemplate` pour les items mais l'instance d'un sélecteur qui retournera les `DataTemplates` au moment voulu.

WPF montre la voie

Voilà... c'est aussi simple que ça et pourtant c'est un procédé d'une grande souplesse et d'une grande puissance. Il s'agit à la fois de programmation (mais très légère comme nous l'avons vu) et de Design, puisque le Designer doit avoir présent à l'esprit

cette possibilité lorsqu'il "pense" une interface. Charge aux développeurs de lui fournir le code du sélecteur. C'est un peu comme pour les convertisseurs.

Emuler la fonction WPF sous Silverlight et Windows Phone 7/8

Le procédé offert par WPF est si simple et si séduisant qu'on aimerait bien l'émuler à 100%. Hélas les contrôles ne possèdent pas de propriétés `ItemTemplateSelector`. Recréer la classe `DataTemplateSelector`, on le verra plus loin, est un jeu d'enfant et se fait en quelques lignes. En revanche le problème est bien de savoir comment l'utiliser de façon aussi simple que sous WPF...

On trouve plusieurs solutions sur Internet. Beaucoup ont choisi de partir sur la base d'un convertisseur de valeur (interface `IConverter`). Cela fonctionne mais je n'aime pas cette solution qui détourne le sens même des convertisseurs et dont la mise en œuvre m'apparait un peu distordue.

J'ai vu d'autres implémentations qui dans une logique jusqu'au-boutiste utilisaient une propriété attachée pour recréer la propriété `ItemTemplateSelector` simulant à 100% WPF mais au prix de contorsions qui, là aussi, m'apparaissent être des bricolages.

De plus, la plupart de ces solutions, si ce n'est toutes, ne sont pas "blendable". Et c'est pour moi un critère essentiel. Si ça ne passe pas sous Blend, ce n'est pas utilisable.

La seule implémentation que j'ai vue qui émule au plus près WPF sans pour autant empêcher l'utilisation de Blend est celle de Raul Mainardi Neto publiée en 2010 sur The Code Project.

Vous pouvez bien entendu lire l'article directement (en anglais). Cela m'évitera en plus des redites.

Mais pour résumer la démarche :

On créée une classe abstraite `DataTemplateSelector` qui jouera le rôle de son homonyme WPF. Cette classe est très courte :

```
/// <summary>
/// WPF emulation. see WPF class of same name.
/// </summary>
public abstract class DataTemplateSelector : ContentControl
{
    /// <summary>
    /// Selects the template.
    /// </summary>
    /// <param name="item">The item.</param>
    /// <param name="container">The container.</param>
    /// <returns></returns>
    public virtual DataTemplate SelectTemplate(object item,
                                DependencyObject container)
    {
        return null;
    }

    /// <summary>
    /// Called when the value of the
        <see cref="P:System.Windows.Controls.ContentControl.Content"/>
        property changes.
    /// </summary>
    /// <param name="oldContent">The old value of the
        <see cref="P:System.Windows.Controls.ContentControl.Content"/>
        property.</param>
    /// <param name="newContent">The new value of the
        <see cref="P:System.Windows.Controls.ContentControl.Content"/>
        property.</param>
    protected override void OnContentChanged(object oldContent,
                                             object newContent)
    {
        base.OnContentChanged(oldContent, newContent);
        ContentTemplate = SelectTemplate(newContent, this);
    }
}
```

Code 50 - Simulation d'un DataTemplateSelector

C'est une classe abstraite donc l'idée est, comme sous WPF, d'en créer des dérivées. Le principe retenu consiste à faire descendre la classe de ContentControl. La ruse sera de fournir un `DataTemplate` tout à fait "normal" au contrôle hôte, et d'y placer

uniquement une instance d'un dérivé de `DataTemplateSelector` qui lui contiendra les variantes de template.

Mais continuons d'explorer ce petit code pour comprendre comment cela fonctionne.

`DataTemplateSelector` (version Silverlight / WP) se contente de surcharger la méthode `OnContentChanged`, méthode protégée de `ContentControl` qui est appelée à chaque fois que la propriété `Content` est modifiée.

Le code appelle la méthode originale mais en profite pour modifier la valeur de la propriété `ContentTemplate` du `ContentControl`. Car heureusement `ContentControl` possède une propriété fixant le template à appliquer à son contenu. Et ce `ContentTemplate` n'est pas modifié n'importe comment, il l'est en appelant une méthode virtuelle `SelectTemplate` qui recevra en paramètre à la fois le nouveau contenu et la référence vers le conteneur (le `DataTemplateSelector`). Toute l'astuce de cette implémentation est qu'elle semble calquer WPF à 100%, même nom de classe, même noms de méthodes, même stratégie. Mais en réalité elle est très différente.

Sous WPF ce sont ensuite les contrôles "normaux" qui exposent une propriété `ItemTemplateSelector` alors qu'ici notre `DataTemplateSelector` est un descendant de `ContentControl`, un conteneur que nous utiliserons à l'intérieur d'un premier `DataTemplate` (le "vrai" item template du contrôle visé). Tel un parasite, notre `DataTemplateSelector` s'accrochera ainsi à un template existant qu'il trompera en quelque sorte en substituant au cas le cas le véritable `DataTemplate`... Assez malin, tout en restant simple et compatible avec du templating sous Blend.

Généralement, comme je le disais dans l'exemple WPF, on ne se sert que de l'item pour baser le choix du template. Mais avoir une référence sur le conteneur peut certainement servir à couvrir des cas encore plus sophistiquées.

Une fois la classe `DataTemplateSelector` simulée sous Silverlight et Windows Phone 7/8, le reste de la démarche ressemble à celle qui prévaut sous WPF... On écrit des `DataTemplates`, puis on écrit une classe dérivée de `DataTemplateSelector`, classe ayant une propriété par `DataTemplate` et une surcharge de `SelectTemplate` qui effectue le choix en fonction de l'item qui lui est passé.

Ne reste plus qu'à créer l'instance du sélecteur, exactement comme sous WPF et à l'utiliser.

C'est là que la différence existe avec WPF. Comme il n'existe toujours pas de propriété `ItemTemplateSelector` depuis tout à l'heure il va falloir jouer plus fin.

On va alors créer par les voies "normales" un `DataTemplate` d'item pour le contrôle (`ListBox`, `ListView`...). Blend va alors lier ce `DataTemplate` à l'objet afficheur (le `ListView` de l'exemple WPF). Au départ, le template est donc vide (sinon il faut le vider totalement).

C'est là qu'on lui ajoute pour seul contenu une instance de notre descendant de `DataTemplateSelector` dont chacune des propriétés `DataTemplate` (les variantes de templates) sera liée aux ressources statiques correspondantes (chaque `DataTemplate` déclaré séparément).

En faisant de cette façon plutôt qu'en imbriquant les `DataTemplates` dans le descendant de `DataTemplateSelector` (ce qui est fait dans la solution de Raul Mainardi Neto) on garde la possibilité d'intervenir visuellement sur chaque `DataTemplate` sous Blend, sinon ce dernier ne les "voit" pas. Xaml étant tellement souple et permissif, que, forcément, Blend impose certaines limites et contraintes qui lui permettent de comprendre ce que le développeur veut faire et proposer ainsi les bons outils au bon endroit. En optant pour une démarche compatible avec Blend on se garantit de pouvoir utiliser cet outil puissant pour modifier les templates.

Code Exemple

Pour la mise en œuvre sur un exemple simple de l'émulation du `DataTemplateSelector` je vous renvoie à l'article de Raul. C'est plus pratique.

En revanche, je me suis amusé à mettre en pratique la chose sur un exemple un peu plus complexe en Silverlight qui utilise un `TreeView`. L'animal, venant du Toolkit (pour Silverlight), n'utilise pas des `DataTemplates` mais des `HierarchicalDataTemplate` (qui heureusement descendent des premiers).

C'est une solution que j'ai utilisé en situation réelle et que j'ai simplifiée à l'extrême pour l'exemple. Dans le contexte, il s'agit d'une gestion de droits d'utilisation. De façon ultra simplifiée : un client être lié à plusieurs dossiers, et un utilisateur a soit accès tous les dossiers d'un client soit il n'a le droit d'accéder qu'à une sélection de ces dossiers.

Un moyen simple est d'utiliser un `TreeView` affichant la liste des clients (niveau 1). Chaque client possède une case à cocher. Si elle est cochée l'utilisateur voit tous ses

dossiers. Sinon on peut développer le nœud client et on voit tous les dossiers de ce dernier, eux-mêmes précédés d'une case à cocher pour les sélectionner. Lorsque la case à cochée du client est checkée (donc accès tous les dossiers de ce client) les nœuds dossiers deviennent *disabled* et sont tous décochés.

De même, lorsqu'il n'y a qu'une sélection de dossiers, le nombre de dossiers autorisés est affiché à côté du nom du client (ainsi que le nombre total de dossiers).

Enfin, je gère un cas spécial : on décoche un client, ce qui signifie qu'on ne donne accès qu'à une sélection de dossier mais on ne choisit aucun dossier dans liste. L'utilisateur n'a donc plus du tout accès à ce client. Pour matérialiser cette situation (possible), un petit carré rouge apparait alors à côté du nom du client, cela étant considéré comme une erreur de saisie.

C'est un exemple très simplifié d'une petite partie d'une application Silverlight. Ces explications sur le contexte vous aideront à mieux le comprendre mais le plus important c'est le code 

Il comporte quelques astuces notamment dans la gestion des comptages, la façon de désactiver certains éléments dans les DataTemplates en utilisant un convertisseur, etc. Ce n'est pas bien gros et vous devriez finir par vous y retrouver. Le principal étant bien entendu la mise en pratique du `DataTemplateSelector` pour Silverlight et Windows Phone...

Le code de l'exemple Silverlight avec le TreeView : DTSPourSL.zip

Conclusion

Silverlight en version Web ou Windows Phone n'est pas fourni out-of-the-box avec tout ce que sait faire WPF même s'il s'en rapproche beaucoup. Dans ce qu'on appelle les "scénarios avancés" on touche parfois les limites. Mais le plus souvent il suffit d'un peu de ruse et d'un soupçon de code pour rebondir et fournir ainsi une expérience utilisateur de même niveau qu'une application desktop WPF.

Le `DataTemplateSelector` est, une fois qu'on connait son existence, un objet essentiel qu'on regrette de ne pas avoir de base dans certains profils XAML tellement il ouvre de possibilités. Ce billet vous aura, je l'espère, fait découvrir cette feature de WPF qui n'est pas très compliquée à reproduire dans les profils plus restreints.

La Grid cette inconnue si populaire…

Il étonnant de voir à quel point un composant de base tel que le contrôle “Grid” peut à la fois être aussi populaire (son utilisation est quasi obligatoire dans une application XAML) et sembler nébuleux à une majorité de développeurs. Ce billet s’adresse plutôt aux débutants mais je suis certain que les plus confirmés en profiteront…

Demandez autour de vous, en commençant même par votre propre personne… Comment définir des lignes et des colonnes dans une Grid ? Que signifie le mode Canvas de la Grid sous Blend ? Quelle est la différence entre une largeur de colonne “Auto” et une autre en “*” (étoile) ?

Il n’y a pourtant pas milles choses à savoir sur la grille mais bizarrement son paramétrage parait opaque à beaucoup de gens ou, plus exactement, il est très rare de voir quelqu’un se servir intelligemment de toutes les possibilités de la Grid… Tout placer à l’œil ou utiliser les facilités d’un EDI comme Blend n’est pourtant pas la solution miracle à cette méconnaissance. Il arrive qu’on ait besoin de mesure précise, de prévoir une mise en page bien particulière. Dès lors les rudiments de la Grid doivent être “instinctifs”, il ne devrait pas y avoir de question à se poser.

Quelque chose ne vas pas avec la grille j’en conviens. Si elle semble si opaque malgré sa grande simplicité c’est qu’il y a un mauvais choix quelque part…

Ce “mauvais choix” est de même nature que la syntaxe du Binding. C’est à dire que la Grid ne se paramètre pas selon des propriétés claires et bien définies, mais par le biais d’une mini-syntaxe locale qui lui est propre. Tout comme le Binding (mais heureusement en moins compliqué que ce dernier !).

Si ce choix se comprend, XAML a ainsi introduit des langages dans son langage. Des choses mystérieuses qui s’écrivent comme des chaines de caractères, non contrôlées à la compilation et utilisant un logique et une syntaxe propre uniquement valable dans ces cas précis et qu’aucune aide de type IntelliSense ne vient vraiment seconder.

C’est pour moi une erreur, une perte de cohérence grave dans un environnement déjà complexe à maitriser. Il en résulte que le paramétrage de la Grid est presque aussi peu connu que les méandres du Binding. Mais d'un autre côté XAML est bien plus rigoureux et "prévisible" que HTML+CSS qui sont un enfer.

Comment vouloir créer des mises en pages sophistiquées sur un savoir dont les bases reposent sur des sables mouvants ? Comment apprendre à conduire une voiture alors qu’on hésite sur la façon de la démarrer ?

Alors reprenons les fondamentaux !

(NB: les exemples qui suivent sont très simples et sont tous réalisés en utilisant Kaxaml, un utilitaire gratuit qui permet de taper directement du XAML et de voir le résultat pour s'exercer, téléchargez-le pour reproduire les exemples et vous entrainer !)

La Grid

C'est avant tout "the" conteneur XAML par excellence. Celui qui apparait par défaut dans toute nouvelle page comme élément racine.

C'est un conteneur rectangulaire. Jusque-là rien ne le différencie d'un `Canvas` ou d'un `StackPanel`.

Mais la `Grid` est pourtant bien particulière : elle permet de définir des tableaux un peu dans le même esprit que les tables HTML.

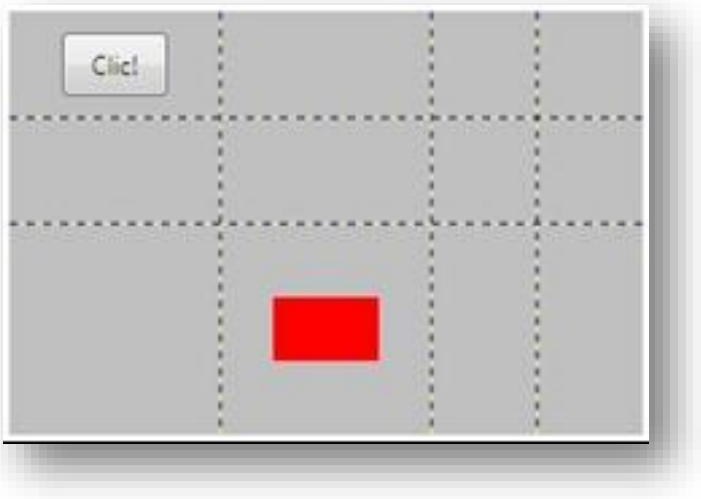
Lignes et colonnes sont définies par des balises, comme pour une table HTML. Sans aucune définition précise une grille se compose en réalité d'une seule cellule de coordonnées (0,0).

Le placement des contrôles enfants dans les cellules s'effectue par le biais des commandes d'alignement de ces derniers (propriétés *attachées* par la grille à ses enfants) concernant les deux axes (vertical et horizontal). Une fois ces alignements déterminés, on peut jouer sur le positionnement d'un élément en ajustant ses marges (`margin`). A la différence du `Canvas` qui ne propose qu'un placement `Top` / `Left`, la grille propose plus de nuance dans les moyens de positionner les éléments dans les cellules. Bien entendu la grille offre aussi à ses enfants des propriétés permettant de décider dans quelle cellule ils doivent être placés (`Grid.Row` et `Grid.Column`). Il existe même des commandes `RowSpan` et `ColumnSpan` pour permettre à un contenu de s'étendre verticalement ou horizontalement sur plusieurs cellules.

La Grid joue aussi un rôle important sur le clipping et le redimensionnement des éléments présents dans les cellules.

Tout cela est déjà finalement assez complexe et non n'avons pas abordé la définition des colonnes et des lignes !

Placement simple



XAML 61 - Grid, placement simple – Capture écran

L'exemple ci-dessus définit une grille comprenant 4 colonnes et 3 lignes. Dans la cellule (0,0) nous avons placé un bouton et dans la cellule (1,2) un rectangle rouge.

La définition d'une telle grille se décompose en deux parties : d'une part la définition des lignes et colonnes, d'autre part les éléments enfants.

```
<Grid Width="300" Height="200" ShowGridLines="True" Background="Silver">
       <Grid.RowDefinitions>
           <RowDefinition Height="50"></RowDefinition>
           <RowDefinition Height="50"></RowDefinition>
           <RowDefinition Height="*"></RowDefinition>
       </Grid.RowDefinitions>
       <Grid.ColumnDefinitions>
           <ColumnDefinition Width="100"></ColumnDefinition>
           <ColumnDefinition Width="100"></ColumnDefinition>
           <ColumnDefinition Width="*"></ColumnDefinition>
           <ColumnDefinition Width="50"></ColumnDefinition>
       </Grid.ColumnDefinitions>
       <Button x:Name="btnTest" Width="50" Height="30" Content="Clic!"
                                       Grid.Row="0"
                                       Grid.Column="0">
       </Button>
       <Rectangle x:Name="rectTest" Width="50" Height="30" Fill="Red"
                                    Grid.Row="2"
                                    Grid.Column="1">
       </Rectangle>
   </Grid>
```

XAML 62 - Code Xaml du placement simple dans une Grid

La section <Grid.`RowDefinitions`> laisse comprendre qu'elle définit une collection de `RowDefinition`, c'est à dire de définitions de lignes. La section `Grid.ColumnDefinitions` fait de même pour la collection des définitions de colonnes.

Une grille se découpe ainsi comme une gros gâteau rectangulaire : par des coupes parallèles aux bords et donc orthogonales entre elles.

Mais l'un des autres problèmes que ce mode de définition pose est que le raisonnement semble être celui des "piquets et des intervalles". On a l'impression de poser des piquets alors qu'en réalité on définit des intervalles... Si je plante trois piquets dans le sol, je ne définis uniquement que 2 intervalles. Or, ici dans la `Grid`, ce qui nous intéresse ce sont les intervalles (les cellules) et ce sont bien ces intervalles qui sont définis, non des piquets malgré leur représentation visuelle sous cette forme...

Ainsi, on pourrait croire que si je définis 3 "piquets", au lieu de définir 2 intervalles, je devrais en définir 4 (car les bords comptent - entre 3+2 piquets il existe 4 intervalles). En fait, en définissant 4 `ColumnDefinition` je définis bien 4 intervalles, donc 4 cellules.

De même pour les lignes, en définissant 3 `RowDefinition`, je définis trois lignes et non deux ou quatre.

> *Bien que ressemblant à une définition de "piquets", la définition des lignes et des colonnes d'une `Grid` correspond à celle des intervalles donc au nombre exact de lignes et de colonnes désirées....*

Regardez à nouveau la capture écran plus haut. J'ai activé l'affichage des "piquets" (`ShowGrildLines=True`). On voit bien 3 piquets verticaux. Comptez... oui il y a 4 cellules sur l'horizontale. Mais regardez le code XAML, il existe bien 4 lignes de `ColumnDefinition`. Des intervalles matérialisés comme des piquets, cela peut créer la confusion.

Donc on se rappelle : nombre de cellules sur un axe = nombre de définitions sur cette axe, tout simplement.

C'est à dire ici : nombre de cellules sur l'axe horizontal = 4 `ColumnDefinition` = 4 colonnes. De même nombre de cellules sur l'axe verticale = 3 `RowDefinition` = 3 lignes.

Ce qui est vraiment trompeur c'est qu'autant Visual Studio que Blend donnent l'impression de placer les "piquets" et de déplacer ces derniers... Mais en Xaml la notion de "piquet" n'existe tout simplement pas ! Si on est habitué à des logiciels comme Expression Design, Photoshop, Illustrator, PaintShop..., tous ces logiciels de dessin permettent généralement de placer des lignes horizontales ou verticales qui peuvent être "magnétiques" ou non. Ici c'est bel et bien une logique de "piquets" et "brochettes". Ces lignes (repères) existent vraiment et possèdent des coordonnées précises. Quand on manipule une `Grid` Xaml, les EDI mettent à notre disposition un procédé visuel qui ressemble à celui des logiciels de dessin, mais ce n'est qu'une astuce visuelle, les piquets et les brochettes n'existent pas, ils sont virtuels et découlent de la définition des intervalles !

Ainsi, lorsqu'on déplace un "piquet" dans l'EDI on ne fait que changer la valeur de l'intervalle (et de l'intervalle voisin!) et non la "position du piquet", position qui n'existe pas, pas plus que le piquet lui-même... Pour en supprimer un il suffit de cliquer sur son marqueur et de taper sur la touche de suppression. Supprimer un piquet fusionne deux lignes ou deux colonnes... parfois on met longtemps à le comprendre !

Bien entendu le raisonnement est rigoureusement le même sur les deux axes. Pour les horizontales on ne parlerait plus de piquets mais de “brochettes” par exemple. Les piquets qui sont verticaux matérialisent les colonnes qui s’étalent sur l’horizontale, les brochettes qui sont horizontales matérialisent les lignes qui découpent la grille verticalement...

Définir les lignes et les colonnes

Dans le code publié à la section précédente on voit clairement les deux blocs définissant les lignes et les colonnes enchâssés dans leurs balises respectives RowDefinitions et ColumnDefinitions.

Prenons les lignes (`Row`). Chaque définition est constituée de la façon suivante :

```
<balise taille=”xx”/>
```

“`balise`” vaut “`RowDefinition`” pour la définition d’une ligne et “`ColumnDefinition`” pour celle d’une colonne.

”`taille`” permet de définit la hauteur de la ligne (`Height`), elle est suivie du symbole égal puis, entre guillemets, d’une valeur. Nous verrons cette dernière plus loin.

Lorsqu’il s’agit d’une colonne, ce n’est plus “`Height`” qui est défini mais en toute logique “`Width`”. Le principe reste le même.

Comme tout langage de type XML, les balises XAML peuvent être fermées immédiatement (mon exemple ci-avant) ou bien être fermées par une balise fermante complète (exemple du code publié plus haut), cela ne change rien à la signification du code.

Les Valeurs des Dimensions

Là les choses peuvent varier et l’effet final sera très différent. C’est même ici que se joue la vraie subtilité de la `Grid`.

Si vous reprenez l’exemple publié plus haut, vous trouverez des définitions de lignes avec des valeurs entières (50 pour les 2 premiers intervalles et “étoile” pour le dernier). Les colonnes utilisent un mode équivalent : valeurs entières (100 et 50) et une valeur “étoile” mais pas en dernière position.

Les valeurs entières permettent de définir un marqueur de façon fixe, au pixel près.

Ainsi ‘`Width=”100”`’ définit une largeur de 100 pixels très exactement, tout comme ‘`Height=”50”`’ définit une hauteur de 50 pixels.

Cette façon de définir des colonnes et des lignes est particulièrement pratique lorsqu'on doit mettre en page des parties d'écran selon un positionnement rigoureux et immuable.

Ce n'est hélas que rarement le cas tant les résolutions et ratios d'écran sur lesquelles peuvent tourner les applications sont aujourd'hui d'une diversité hallucinante... Il est donc souvent préférable de concevoir des mises en pages souples dites "dynamiques". Pour le développeur qui vient du monde du Desktop, il s'agit souvent d'un nouveau réflexe à prendre. Ceux qui viennent du développement Web sont rompus à cette problématique et à la gymnastique qu'elle impose.

L'étoile

Utilisée par deux fois dans l'exemple, l'étoile définit une colonne (ou une ligne) de largeur (ou hauteur) variable. Mais pas variable n'importe comment... L'étoile utilisée seule indique d'utiliser toute la place restante dans l'espace de la `Grid`.

Comme dans l'exemple nous avons défini une largeur fixe pour la grille (balise `<Grid Width="300"...>`) on peut calculer à coup sûr la taille de la colonne notée étoile : 300 – 100 – 100 – 50 = 50. Dans ce cas précis (une grille ayant une largeur fixe), l'intérêt de l'étoile est purement académique, nous aurions pu mettre 50 directement... Mais si la Grid est elle-même placée dans un autre conteneur et qu'elle n'a pas une taille fixe, la colonne définie avec une étoile aura une taille qu'il n'est pas possible, *a priori*, de connaitre.

En tout cas, ainsi définie, la grille possède une colonne et une ligne dont la taille variera automatiquement si les dimensions de la grille sont changées. Si on agrandit en hauteur la grille, c'est la dernière ligne (hauteur = étoile) qui s'agrandira, les autres conserveront leur taille (puisque fixée eu pixels). De même si on élargit la grille, c'est l'avant dernière colonne qui profitera du nouvel espace puisque c'est elle qui est marquée par l'étoile.

Ce mode de dimensionnement est particulièrement souple mais impose de bien "calculer son coup", donc d'avoir un sketching assez précis de l'écran (ou du contrôle) final avant de se lancer à l'aveuglette !

Proportionnalité

Intervalle fixe, intervalle variable, il est aussi possible de définir des intervalles par pourcentage.

Dans ce cas la valeur est suivie d'une étoile. Cette dernière conserve donc le sens de "proportionnalité" qu'on lui a vu précédemment mais ici le développeur peut fixer la proportion.

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="2*"></ColumnDefinition>
    <ColumnDefinition Width="2*"></ColumnDefinition>
    <ColumnDefinition Width="5*"></ColumnDefinition>
     <ColumnDefinition Width="1*"></ColumnDefinition>
</Grid.ColumnDefinitions>
```

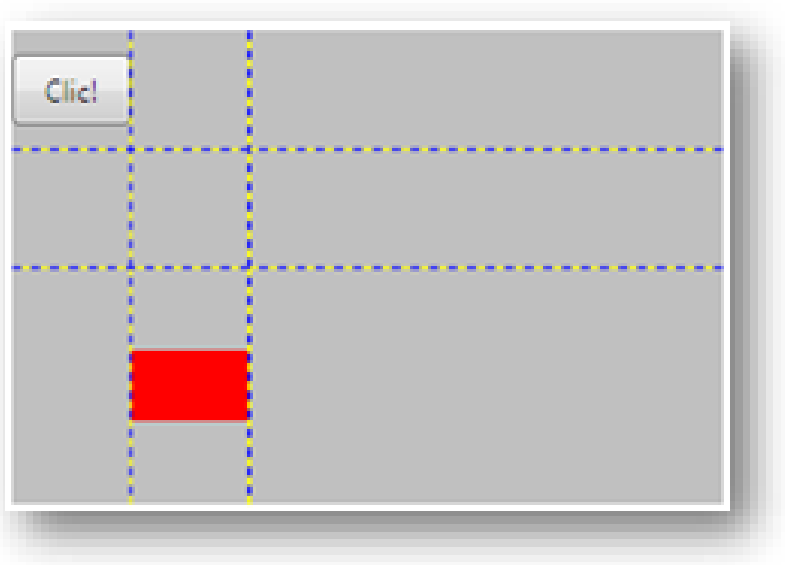
XAML 63 - Définition des colonnes d'une Grid

La somme des proportions doit être égale à 100% peu importe les nombres. Dans l'exemple ci-dessus la somme fait 10. Remplacez les valeurs par leur dixième (0,2; 0,2; 0;5 et 0,1) et vous obtiendrez exactement le même résultat !

Dans le mode proportionnel toutes les lignes (ou colonnes) ainsi définies se comportent comme la colonne "étoile" vu précédemment, sauf que les tailles relatives sont conservées (en fonction des valeurs de proportion saisies). Donc si la taille de la grille change, les cellules changeront de taille en conservant l'aspect général des intervalles.

Taille automatique

A tous les modes déjà vus s'ajoute le mode automatique. Un intervalle défini en automatique se comportera de la façon suivante : toutes les cellules de ligne (ou de la colonne) prendront la taille du plus grand objet contenu dans cette ligne (ou cette colonne).



XAML 64 - Taille automatique de colonne d'une Grid - capture écran

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="auto"></ColumnDefinition>
     <ColumnDefinition Width="auto"></ColumnDefinition>
     <ColumnDefinition Width="auto"></ColumnDefinition>
     <ColumnDefinition Width="auto"></ColumnDefinition>
</Grid.ColumnDefinitions>
```

XAML 65 - Colonnes de Grid en taille automatique

Comparez l'image ci-dessus avec l'image se trouvant plus haut. C'est un peu le jeu des 7 différences

La première chose que l'on constate est que chaque colonne a une largeur identique à l'objet qu'elle contient (colonne 0 : le bouton, colonne 1 : le rectangle rouge).

Mais on peut aussi voir que derrière ces éléments, et bien que nous ayons défini 4 colonnes, il ne s'en trouve qu'une seule de visible qui prend "tout le reste" de l'espace. La colonne 2 (la troisième donc) existe bel et bien, mais s'adaptant à son contenu (qui est "rien"), elle prend une taille qui ne vaut.. rien, donc zéro. Il en va de même pour la dernière colonne. Mais comme la grille est un espace rectangulaire qui ne peut pas contenir de trou, l'espace semble être rempli (c'est la couleur grise choisie pour le background qui donne cet effet).

Egalité

Variable, automatique, proportionnel, un intervalle peut se définir de plusieurs façons. Mais il est aussi bien pratique parfois d'obtenir uniquement un quadrillage régulier.

Comme toujours en XAML il y a plus d'un chemin pour arriver au résultat. On pourrait définir toutes les colonnes (ou lignes) avec l'étoile :

```
<ColumnDefinition Width="*"></ColumnDefinition>
<ColumnDefinition Width="*"></ColumnDefinition>
<ColumnDefinition Width="*"></ColumnDefinition>
<ColumnDefinition Width="*"></ColumnDefinition>
```

XAML 66 - Colonnes de Grid en mode "étoile"

On obtiendrait bien 4 colonnes de même dimension.

On pourrait aussi décomposer les proportions :

```
<ColumnDefinition Width="25*"></ColumnDefinition>
<ColumnDefinition Width="25*"></ColumnDefinition>
<ColumnDefinition Width="25*"></ColumnDefinition>
<ColumnDefinition Width="25*"></ColumnDefinition>
```

XAML 67 - Colonnes de Grid en mode proportionnel

4 fois 25 font 100, il y a bien 4 colonnes définies à 25%, donc 4 colonnes identiques.

Mais il y a plus simple :

```
<Grid.ColumnDefinitions>
   <ColumnDefinition />
   <ColumnDefinition />
   <ColumnDefinition />
   <ColumnDefinition />
</Grid.ColumnDefinitions>
```

XAML 68 - Colonnes de Grid automatiquement proportionnelles

Limiter les intervalles

Nous avons vu comment la Grid nous laisse jouer sur les tailles de ses cellules. La liberté est d'autant plus grande qu'il est possible de mixer les modes entre eux (avec une colonne fixe suivie d'une colonne étoile; avoir 3 colonnes proportionnelles et une 4ème automatique, etc...).

Mais cela ne s'arrête pas là. La Grid permet aussi de définir des tailles minimum et maximum pour chaque définition de ligne ou de colonne grâce aux propriétés MaxWidth et MawHeight.

XAML 69 - Effet visuel de MaxWidth et MawHeight dans les colonnes d'une Grid

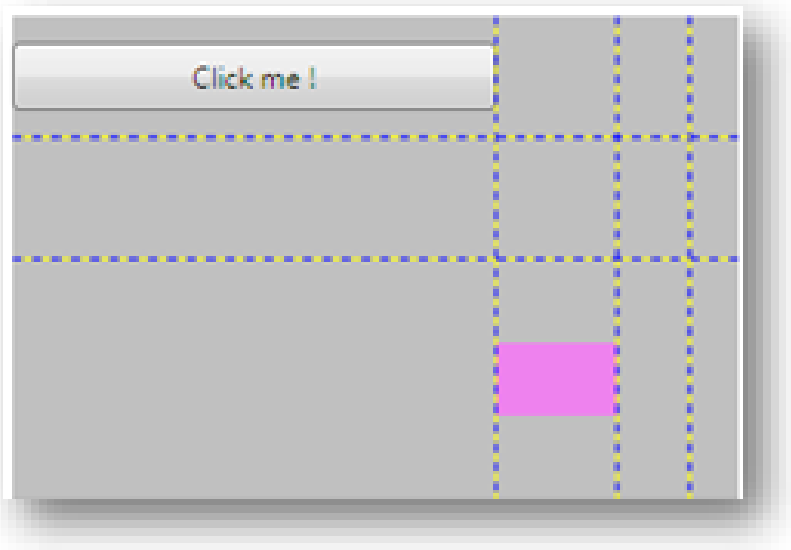
```
<Grid Width="300" Height="200" ShowGridLines="True" Background="Silver">
   <Grid.RowDefinitions>
      <RowDefinition Height="50"/>
      <RowDefinition Height="50"/>
      <RowDefinition Height="*"/>
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition MinWidth="30" MaxWidth="150"/>
      <ColumnDefinition Width="Auto" MinWidth="30" MaxWidth="150"/>
      <ColumnDefinition Width="Auto" MinWidth="30" MaxWidth="150"/>
      <ColumnDefinition Width="Auto" MinWidth="30" MaxWidth="150"/>
    </Grid.ColumnDefinitions>

 <Button Width="200" Height="28" Content="Click me !"
                                 Grid.Row="0" Grid.Column="0"/>
 <Rectangle Width="50" Height="30" Fill="Violet"
                                 Grid.Row="2" Grid.Column="1"/>
</Grid>
```

XAML 70 - MaxWidth et MinWidth dans les colonnes d'une Grid

Ici nous avons placé un bouton de 200 pixels de large dans une colonne dont la taille est limitée à l'étendue 30-150 pixels. De fait le bouton est coupé. La Grid sait donc effectuer un *clipping automatique* sur le contenu des cellules. Astuce à se rappeler lorsqu'on sait que le clipping n'est pas supporté de façon égale par tous les conteneurs (du moins automatiquement).

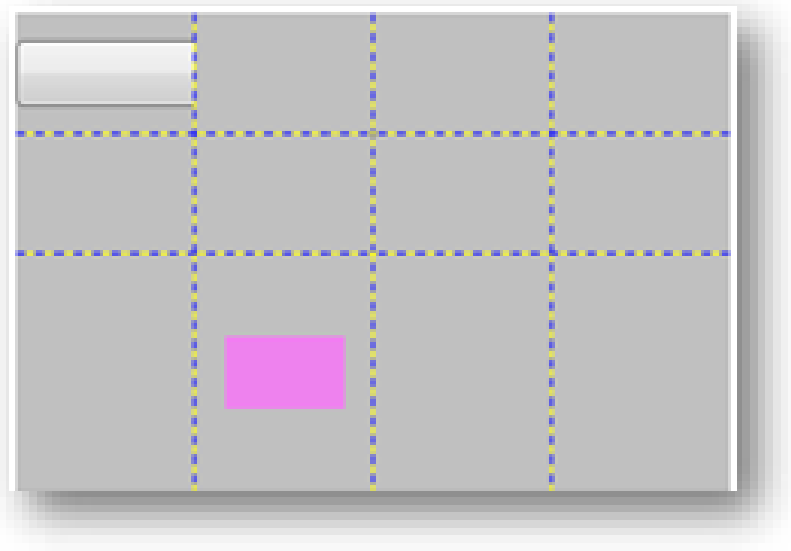
Dans le code exemple ci-dessus l'effet est rendu parce que toutes les colonnes sont automatiques sauf la première. Toutes les indications de taille sont très sensibles à leur "environnement". Par exemple si nous passons la première colonne en mode auto (en conservant toutes les autres définitions) nous obtiendrons cela :



XAML 71 - Mélanger les définitions de colonnes dans une Grid

C'est comme si l'indication de taille maxi de la colonne 0 n'était plus prise en compte...

De même, Si nous supprimons l'indication "auto" de toutes les colonnes, encore une fois sans rien changer d'autre, nous obtiendrons :



XAML 72 - Importante du mode "auto" dans les définitions de colonnes de Grid

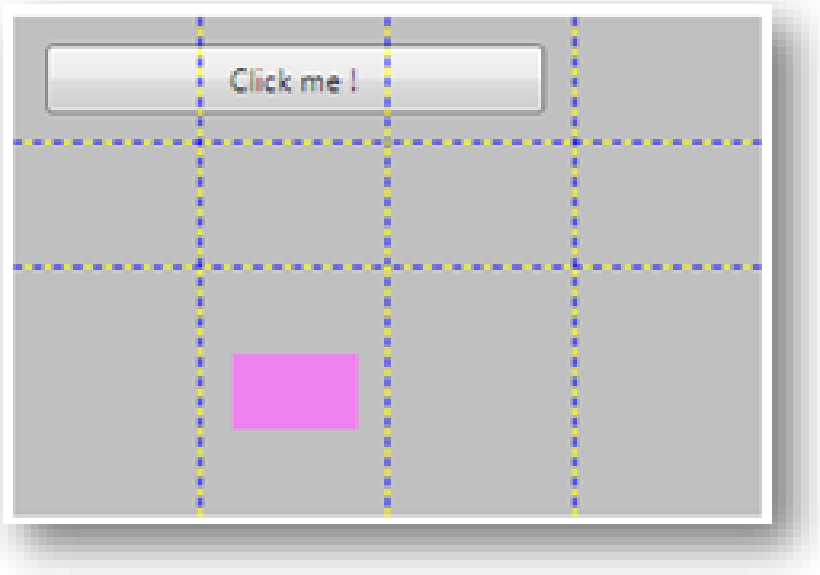
Toutes les colonnes ont la même taille...

Tout cela est assez délicat à bien maitriser (c'est à dire intégrer intelligemment l'ensemble de ces possibilités dans un design bien pensé). Pas si simplette que ça la Grid finalement...

Placement hors cellules

La `Grid` permet de définir des colonnes et des lignes format des cellules dans lesquelles on peut placer divers objets. Il n'est pas nécessaire d'insérer des grilles dans les cellules si on désire placer plusieurs objets dans celles-ci, mais qu'il s'agisse de `Grid`, de `Canvas` ou autre `StackPanel` cela rend le positionnement plus facile dans le cadre de mises en page dynamiques.

La Grid permet de définir assez librement le placement des objets dans les cellules avec de nombreuses options d'alignement (gauche, droit, haut, bas, stretch, center) et de marges. Elle autorise aussi l'étalement d'un objet sur plusieurs colonnes ou lignes :



XAML 73 - Placement hors cellules

```
<Button Width="200" Height="28" Content="Click me !"
   Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3"></Button>
```

XAML 74 - Placement hors cellules via un ColumnSpan

En utilisant `Grid.ColumSpan = "3"` dans la définition du bouton j'ai indiqué que celui-ci devait s'étendre sur les 3 colonnes commençant à la colonne 0 (puisque le bouton est placé en `Grid.Column=0`).

FAQ

Peut-on avoir des lignes non pointillées ?

En réalité les lignes pointillées affichées par `ShowGridLines` n'ont pas vocation a être affichées dans une application terminée. Elles ne servent que d'aide au placement en mode conception. Pour ajouter une bordure à une cellule il faut y placer un `rectangle` ou un `Border`, de même si on souhaite dessiner les contours d'une `Grid` il faut la placer dans un `Border` par exemple. On peut aussi dessiner de simples lignes en utilisant un `Path`.

Qu'est-ce que le mode layout ou Canvas de la Grid ?

Il s'agit d'une astuce de Blend. Quand la grille est en mode `Layout`, le déplacement d'un "piquet", en forçant le changement de taille des cellules qu'il définit, force à un repositionnement des éléments éventuellement contenus dans les cellules concernées. En mode `Layout` les objets vont changer de taille car c'est leur `Margin` qui est conservée.

En mode `Canvas`, Blend ne cherche plus à préserver les marges mais au contraire le positionnement et la taille des objets. Tout naturellement ce sont alors les marges des objets qui sont modifiées...

Peut-on placer un objet à cheval sur deux cellules ?

Nous avons vu que le `RowSpan` ou le `ColumnSpan` permettaient de placer un objet en le faisant s'étaler sur plusieurs lignes ou colonnes, c'est une approche. Quand celle-ci ne convient pas et qu'on veut pouvoir placer un objet n'importe où dans la grille sans se soucier des cellules une des solutions est tout bêtement de poser l'objet sur le conteneur parent de la `Grid` et de le positionner exactement où on le souhaite (avec un z-order supérieur à celui de la grille bien entendu).

Grid ou Canvas ?

Les mises en page se font avec des grilles et non des `Canvas` car seule la grille gère les proportions, le clipping des cellules, etc, c'est à dire de nombreuses options très utiles lorsqu'il s'agit de faire de la mise en page dynamique (ou non d'ailleurs). Le `Canvas` est un conteneur "bête" n'acceptant qu'un positionnement Top/Left, sans clipping. Bête dans le sens de simple, mais très utile lorsqu'il s'agit de positionner des dessins au pixel près en étant sûr que rien ne viendra modifier ce placement.

Toutefois une grille avec une seule cellule peut être aussi utilisée (en bénéficiant d'options supplémentaires).

Conclusion

Il y aurait encore beaucoup de choses à dire sur le contrôle `Grid`. Versatile, parfois déroutant, il reste le contrôle de base utilisé de façon privilégiée pour toute mise en page.

Ce billet s'adressant avant tout aux débutants, j'espère que les lecteurs confirmés y auront malgré tout trouvé matière à réflexion...

Mise en page dynamique et transitions

A chaque génération Blend rend la mise en page de plus en plus fluide, vivante et ce avec le moins de code possible (et si possible sans code du tout). Le double but : rendre les applications XAML encore plus attractives pour les utilisateurs et rendre Blend utilisable à 100% par un Designer sans mettre les mains dans le code.

Et, en effet, à chaque nouvelle génération les choses se sont améliorées approchant toujours de plus près ces deux buts. Blend ne s'est pas arrêté à la version 4 sortie avec Silverlight 4. Avec la fin de la série « Expression » il s'est vu intégré à Visual Studio – version 2012 puis 2013 – devenant à chaque fois plus puissant et plus polyvalent avec la prise en charge notamment des interfaces WinRT en HTML. Depuis la version 4 Blend a encore beaucoup évolué pour arriver à une maturité impressionnante, en total accord avec XAML de Silverlight 5 et de WPF 4.5 ou le XAML de WinRT. Si on peut espérer raisonnablement voir l'outil se bonifier encore plus avec les prochaines releases on peut penser avoir atteint un palier et un sentiment de "complétude" nous touche quand on regarde l'édifice. En tant que fan de Blend je me réjouis que ce produit ait pu survivre à tous bouleversements et que d'une voie de garage (Silverlight) il ait su s'imposer comme compagnon indissociable de Visual Studio pour toutes les plateformes Microsoft, de Windows Phone 8 à WinRT en passant par ce curieux mode de programmation en « faux » HTML de WinRT (faux puisqu'il ne tourne que sous Windows RT et non pas sur le Web).

Grâce à cet outil extraordinaire il est possible pour le Designer ou l'informaticien un peu doué de faire des mises en pages efficaces et séduisantes.

La mise en page dynamique

Je ne vais pas vous parler de toutes les nouveautés de Blend, j'y reviendrai certainement dans de prochains billets, mais d'un aspect essentiel pour les applications modernes : supporter une **mise en page fluctuante**.

- Fluctuante en fonction des *dimensions physique du support* (écran du PC, notebook, smartphone, tablettes…) ;
- Fluctuante en fonction de la densité des affichages qui passent aujourd'hui de 96 DPI d'un bon écran PC à près de 400 DPI sur certains smartphones ou tablettes ;
- Fluctuante en fonction de la *cinématique même de l'application* (changement de page, ouverture et fermeture de panneaux…) ; Et enfin
- Fluctuante en fonction de la *dynamique des données* (apparition de nouvelles données, modification, suppression de données…).

On pourrait même ajouter les *fluctuations liées à l'utilisateur* (préférences, droits d'accès…) qui peuvent aussi impacter la mise en page d'une application.

Bien entendu les animations sont présentes depuis les débuts de XAML, mais elles ont toujours souffert et souffrent certainement encore un peu, des animations Flash bas de gamme dont le Web fut gavé pendant un certain temps. La proximité de Flash et Silverlight a certainement rendu cette confusion encore plus grande.

Heureusement les mœurs évoluent, et quels que soient les travers qu'on peut reprocher à Apple, il faut leur rendre grâce d'avoir inventé l'iPhone qui a mis dans les mains de millions d'utilisateurs un outil informatique où le look compte autant que les fonctionnalités. Il est vrai qu'Apple l'avait déjà fait avec le Mac sans qui Windows n'existerait certainement pas. Mais c'est une guerre perdue qui date d'il y a 20 ans… Une piqûre de rappel était nécessaire pour espérer faire sursauter le monstre de Redmond ! Quand on voit à quel point Microsoft a laissé passer sa chance on peut toutefois rager, le monstre dormait trop bien sur ses deux oreilles et il s'est réveillé un peu tard… Car Windows Mobile existait bien avant l'iPhone, et Windows aussi. Mais personne n'avait pensé marier la puissance de l'un avec le look de l'autre, Ballmer se gaussant de l'iPhone en prédisant que personne n'en voudrait (vous rappelez-vous de cette interview qui laisse aujourd'hui encore plus songeur ?). Pourtant Microsoft

avait aussi inventé le concept génial de l'Origami (2006), une tablette avant l'heure, restée au stade de concept qu'Apple à transformer en succès avec l'iPad. De sacrés copieurs Apple tout de même, mais quel talent qu'on jalouse forcément quand on voit les technologies merveilleuses inventées par Microsoft qui ne se vendent pas ou mal (Zune, Surface, Windows Phone, Windows 8.x …). C'est un cas d'école vraiment intéressant, le copieur talentueux vs le l'inventeur génial qui ne sait pas vendre ses créations…

Quoi qu'il en soit et pour l'homme de la rue qui n'a pas suivi la comète Origami, c'est Apple qui a fait du look & feel des applications une composante essentielle de l'informatique moderne. Dommage pour Microsoft qui avait le jeu en main et des années d'avance sur tout ça.

Et cette avancée-là, c'est celle qui fait qu'une application n'est plus égale à « algorithmes + structures de données » comme le prônait le père de Pascal Niklaus Wirth dans les années 76[8], mais qu'une application moderne est égale aujourd'hui à « *fonctionnalités + Expérience Utilisateur de qualité* ».

C'est réellement un nouveau paradigme, et je le dis avec d'autant plus de plaisir que cela doit bien être la première fois depuis 20 ans qu'on prononce ce mot à bon escient !

Dans ce nouveau cadre, la mise en page d'une application n'est plus « accessoire », elle devient un élément indissociable de l'application qu'elle contribue à créer tout autant que le code qui la compose.

Du fait des fluctuations évoquées plus haut, les mises en pages, en dehors d'être esthétiques, se doivent techniquement d'être fluides et dynamiques donc.

Donner vie au cycle de vie de l'information

Car les mises en page dynamiques intégrant des animations confèrent avant tout **une meilleure visibilité au cycle de vie de l'information**. Quand dans une liste un item disparaît ou apparaît, cela est instantané en raison même des capacités incroyables de nos ordinateurs modernes. Si la liste est l'élément d'interface principal et que l'utilisateur à les yeux rivés dessus il verra certainement quelle ligne vient de disparaitre ou d'apparaitre (mais pas toujours). Mais si l'utilisateur n'est pas concentré

[8] « Algorithms + Data Structures = Programs » livre publié en 1976 par Niklaus Wirth inventeur des langages Pascal – 1968/69 - et Modula (1976).

à ce moment précis sur l'interface ou bien si la liste en question n'est qu'un des éléments du visuel de l'application, il ne s'apercevra de rien du tout le plus souvent !

Cette vitesse, cette instantanéité, nous l'avons tous recherchée pendant des années : chaque développeur consciencieux essayant de rendre son application plus "réactive", plus "rapide" que celle du concurrent, luttant contre la lenteur de PC n'ayant pas la puissance dont il rêve… De l'autre côté des armées d'ingénieurs travaillaient à rendre ce même PC toujours plus rapide… Mais ces temps sont révolus, les machines d'aujourd'hui ont plus de puissance et de mémoire qu'il n'en faut pour traiter la compatibilité de cent PME à la fois. Même les cartes graphiques modernes de simple obligation pour piloter un écran sont devenues des monstres de puissance servant à bâtir des supercalculateurs ! Du coup, il reste de la puissance disponible pour s'attacher à un autre aspect que la pure fonctionnalité : le visuel et "l'expérience utilisateur" la fameuse UX des américains (User eXperience).

Ainsi, pour notre liste d'items, *l'instantanéité de l'arrivée ou de la sortie d'une nouvelle ligne n'est pas forcément un avantage*. Le fait d'arriver ou de disparaitre est **aussi une information** pour une ligne, son contenu n'est pas la seule information existante qui doit être véhiculée à l'utilisateur… *Son cycle de vie est une information tout aussi importante*.

Le Design, un gadget ?

Ceux qui pensent (et il y en a encore) que le nouveau paradigme des applications "designées" n'est qu'un gadget, un artifice de vente pour les commerciaux ou de la simple "frime" n'ont rien compris et passent à côté de l'essentiel.

Le fait pour une information de naître ou de mourir, de passer d'un état à un autre (et *pas seulement d'être dans tel ou tel état*) est **aussi** une information **essentielle** pour l'utilisateur et *comme toute information elle doit être mise en valeur* (selon son degré d'importance fonctionnelle) !

Le Design d'une application est une étape désormais incontournable. Non seulement parce qu'il rend l'application plus "attractive", mais surtout parce qu'il permet de **mettre l'information en valeur au lieu de mettre la technique en avant**. Le temps des techniciens, des tripoteurs de bits est morte, celle des designers arrive…

Et XAML ?

Peut-être certains penseront qu'une fois encore en éternel bavard je me perds dans des digressions au lieu d'aller à l'essentiel : la technique.

Je ne leur dirai pas de passer leur chemin, au contraire ! Je prêche aussi (et surtout ?) pour que ceux-là m'entendent ! Et il convient donc de ne pas les chasser ni de leur jeter la pierre, ou alors juste une petite … Car la "technique" nous sommes en plein dedans ! Ce qui peut échapper à certains c'est que **la "technique" a changé** ! Et que ce qui était hors de son cadre il y a dix ans fait aujourd'hui partie intégrante de ces bases.

> **Le passéisme n'est finalement pas tant le fait de rester accroché aux valeurs d'un passé idéalisé que d'être aveugle aux changements et aux évolutions du temps présent…**

Un peu d'archéologie !

La préhistoire (an 2007)

Dès sa naissance, Blend offre la possibilité de créer des animations pour Silverlight et WPF sur la base des StoryBoards, en manipulant des KeyFrames. C'est la base de tout ce qui suivra. Les nouvelles fonctionnalités auront pour base des Storyboards écrits par le développeur, d'autres vont construire ces StoryBoards, et d'autres utiliseront les deux approches.

L'histoire ancienne (an 2008)

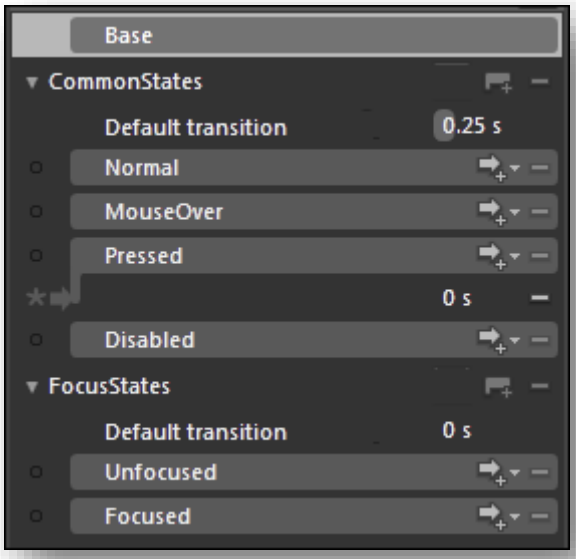
Dans Blend 2 SP1, les équipes Blend et Silverlight introduisent le Panneau des États "States Panel" qui permet de créer et de gérer les *VisualStates* et les *VisualStateGroups* (Silverlight 2 et WPF 3.5 avec le Toolkit).

> *C'est ici qu'a été introduite la **notion d'état** définie comme un moyen de communication entre le code et le visuel et comme une simplification énorme dans la description et l'utilisation des différents états visuels.*

Le code peut décider en fonction de ses entrées de passer dans un état ou un autre. L'état signale ainsi de façon simple et avec un nom explicite une situation qui elle peut découler de conditions ou calculs complexes. Sous cet angle un état est un niveau d'abstraction utilisé par le code pour avertir "l'extérieur" sur ce qu'il vient de

faire, ce qu'il s'apprête à faire, sur ce qui est possible de faire ou sur ce qui est impossible à faire. Les états sont généralement à double sens, c'est à dire qu'il est possible, depuis l'extérieur du code, de faire changer ses états internes en fixant un nouvel état global.

Du côté de l'interface, les états sont des "constats" d'une situation qui peuvent être pris en compte pour modifier l'aspect de l'affichage. Aucun code n'est nécessaire, tout se fait grâce à un intermédiaire : le panneau des états derrière lequel se cache le VSM (*Visual State Manager*).



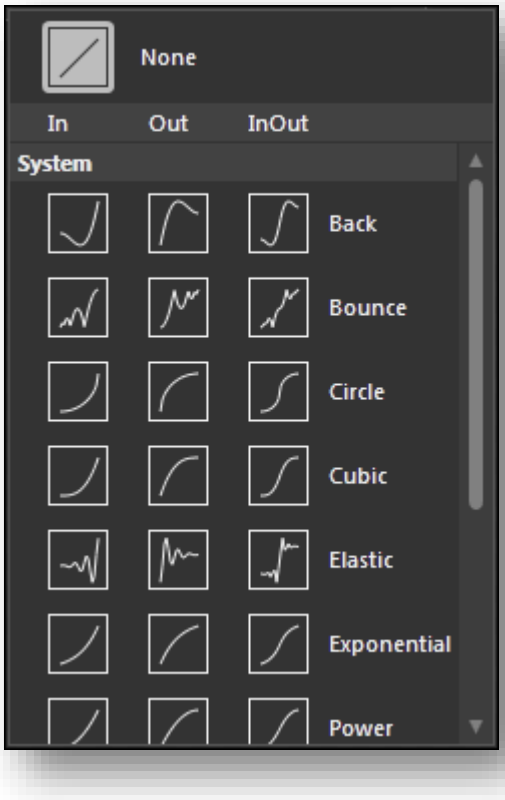
XAML 75 - Visual State Manager dans Blend

Le VSM permet même de fixer un temps global ou ponctuel pour gérer automatiquement des transitions entre les différents états visuels.

Le VSM a toutefois ses limites. Par exemple il ne peut calculer que des interpolations linéaires entre deux valeurs numériques. Il ne marche pas vraiment pour les propriétés prenant des valeurs discrètes (énumérations par exemple) ou pour des données qu'il ne peut connaître à la conception. Cela peut toutefois être contourné car il est toujours possible de rendre continue une valeur discrète (en effectuant une gamme de valeurs numériques en général), les valeurs « intermédiaires » n'ont pas forcément de sens, mais pour faire varier une couleur ou la position d'un élément visuel cela peut fonctionner.

Histoire récente (an 2009)

L'équipe de Blend a beaucoup travaillé sur ces limitations et dans la version 3, elle a ajouté de nombreuses améliorations. La première concerne les *EasingFunctions* qui permettent d'ajouter un peu de "crédibilité" visuelle aux animations. Effet d'amortissement, ralentissement progressif, rebonds élastiques, tout cela s'ajoute simplement à toute animation par le biais d'un menu présentant de façon graphique les diverses fonctions disponibles (en entrée d'animation, en sortie ou bien les deux). Les EasingFunctions sont applicables à une `KeyFrame` donnée ou une animation et un easing par défaut peut être défini globalement pour toutes les animations découlant d'un même changement d'état (transition).



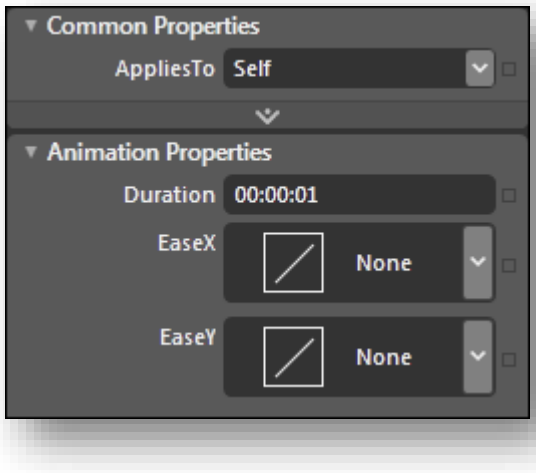
XAML 76 - Easin et Easout sous Blend

La seconde idée introduite dans la V 3 est celle de **comportement** (les *behaviors*). L'un des tous premiers est justement le `GotoStateAction` permettant de programmer les changements d'état du code sans avoir à utiliser de code-behind (C# ou VB). De fait l'interface de passive (faisant le constat d'un changement d'état du code) devient active en autorisant la modification de l'état du code.

La troisième idée consiste en l'ajout du `FluidMoveBehavior`. Ici c'est tout un ensemble de nouveaux scenarii qui sont concernés. Par exemple les éléments d'un `StackPanel` sont amenés à changer de position suite à un redimensionnement, l'ajout ou la suppression d'un contrôle dans leur collection. Aucun procédé simple n'existait pour gérer ce type de situation. Cela était possible, mais à la condition de le faire par code. Un code souvent complexe, très spécialisé et peu souple (sauf à entrer dans un développement démesuré).

Le `FluidMoveBehavior` vient ici combler une lacune en automatisant de façon simple et déclarative des transitions animées lorsque les éléments enfants d'un conteneur changent de taille, de position, naissent ou meurent. Le tout contrôlé par les `EasingFunctions` pour plus de réalisme et d'attractivité visuelle.

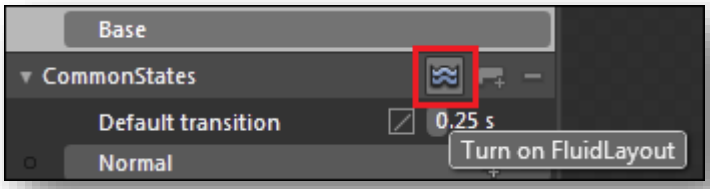
Bien plus que d'étendre les possibilités de Blend, c'est bien un pas de géant en plus qui est fait dans la direction d'une programmation visuelle réalisable par un Designer sans l'aide de code.



XAML 77 - Réglage d'une fonction de ease sous Blend

Mais il restait encore quelques scenarii inaccessibles au système d'animation de Blend malgré ces évolutions essentielles. Par exemple il était impossible d'animer un changement entre `Visible` et `Collapsed`… Dès qu'on souhaitait cacher un élément ou le rendre visible il fallait jouer sur son opacité. Or cela n'a pas le même effet : un élément `Collapsed` est "retiré" visuellement : sa place devient libre et d'autres contrôles peuvent remplir l'espace vacant.

Après certainement pas mal de recherches, l'équipe de Blend créa la quatrième bonne idée introduite dans Blend 3 : le `FluidLayout`.



XAML 78 - Accès au FluidLayout sous Blend

Il suffit de cliquer sur ce petit bouton dans le panneau du VSM et tous les changements de mise en page dans le `VisualStateGroup` considéré seront animés (changements d'état) même lorsque cela paraît impossible, comme pour une propriété `Visibility`.

Techniquement le `FluidLayout` prend un cliché de la mise en page avant la transition et un autre après (ce qui est non visible heureusement, ce sont des opérations internes) et il créé un *morphing* entre les deux clichés… Cela permet d'animer et de rendre "fluide" des transitions qui, autrement, étaient brutales.

L'histoire présente (an 2013)

Avec Blend 4 en 2010 l'idée principale était au sujet des animations d'aller encore plus loin sur la voie tracée par les améliorations judicieuses de la V3.

LayoutStates

Par exemple, prenons le cas d'une liste et de ses items. Comment rendre l'arrivée ou la sortie d'un item plus attractive qu'un simple changement instantané (voir ce que je disais bien plus haut à ce sujet) ?

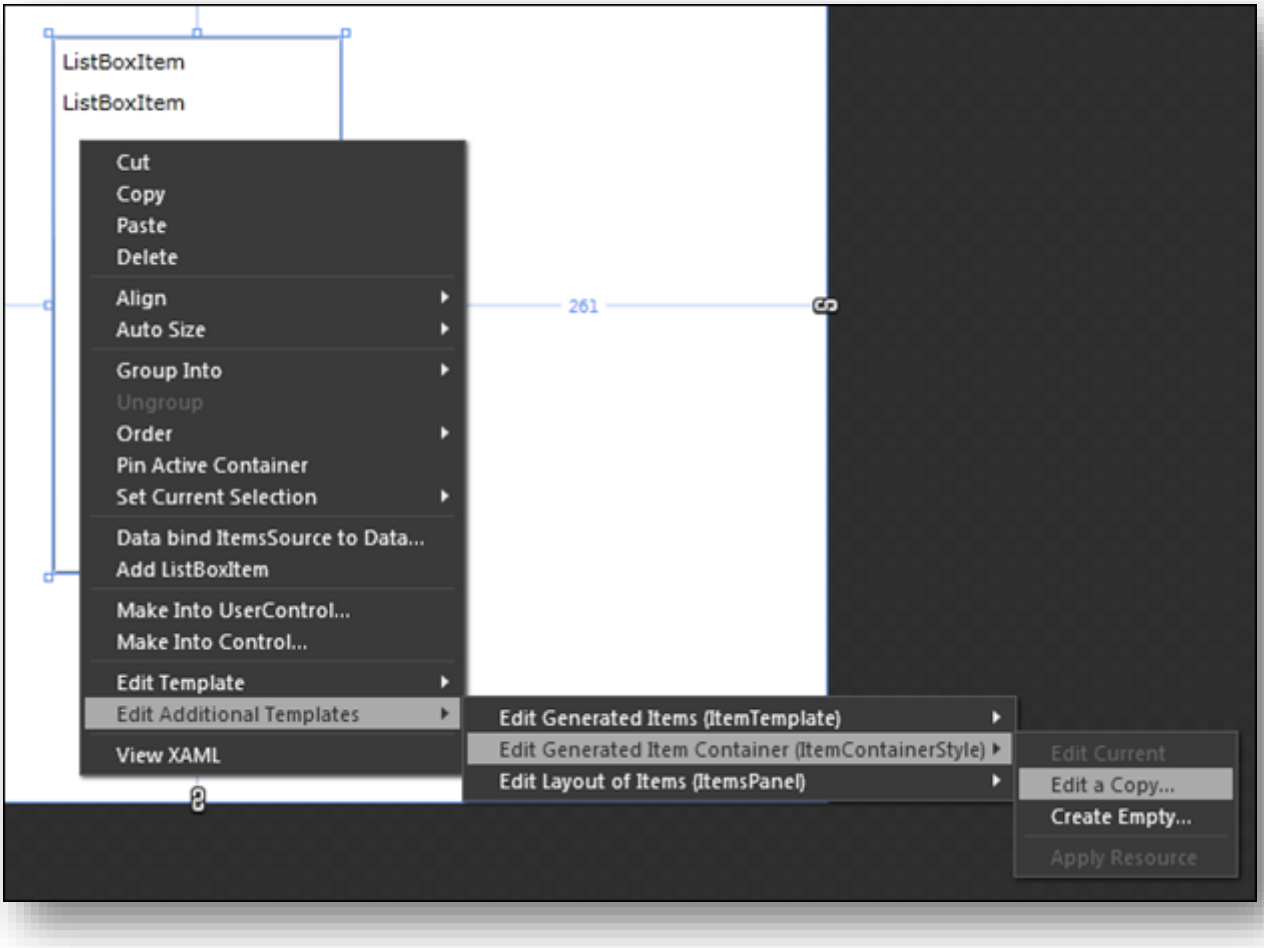
La première solution, grâce aux nouveautés de la V3, consiste à utiliser un `FluidMoveBehavior`. Ce n'est déjà pas si mal. L'entrée d'un nouvel item se fera doucement : les items présents vont, grâce au `FluidMovebehavior`, se pousser gentiment pour lui faire de la place. Lors de la sortie d'un élément les items présents viendront tout aussi gentiment combler le trou laissé par le partant.

Mais au-delà de cette parade (déjà efficace) il n'était pas forcément facile de réellement contrôler l'élément entrant ou sortant. Les développeurs les plus malins pouvaient, en bricolant plusieurs événements, arriver à contrôler, au moins partiellement, l'entrée d'un item. Mais il fallait être sacrément rusé (et au top niveau de l'utilisation de Xaml et Blend) pour trouver une solution à la situation inverse (la

sortie d'un item). Et encore cela au prix d'un incroyable déploiement d'efforts et de codes alambiqué.

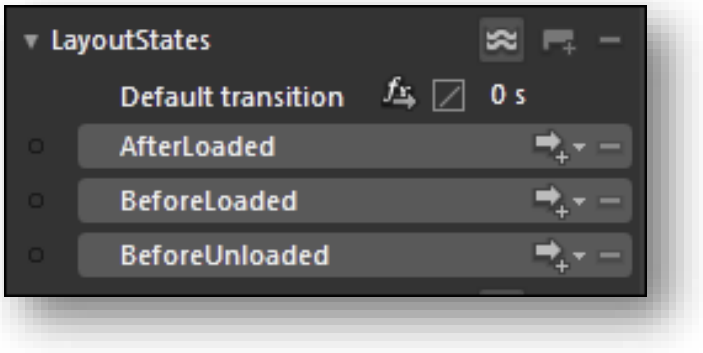
Pour pallier ces cas assez courant, l'équipe de Blend a travaillé avec l'équipe XAML. La solution s'appelle les "`LayoutStates`". Ils sont bien cachés et nul doute que vous ne les découvrirez pas tout seul juste en regardant Blend…

Pour les trouver il faut aller les chercher là où ils se cachent, c'est à dire là où la solution a été implémentée en éditant le `ItemContainerStyle` du contrôle (`ListBox` par exemple).



XAML 79 - Accéder aux LayoutStates

Pour l'instant rien de spécial… mais si vous regardez attentivement le panneau du VSM vous découvrirez 3 nouveaux états :



XAML 80 - Les LayoutStates sous Blend, réglages des animations

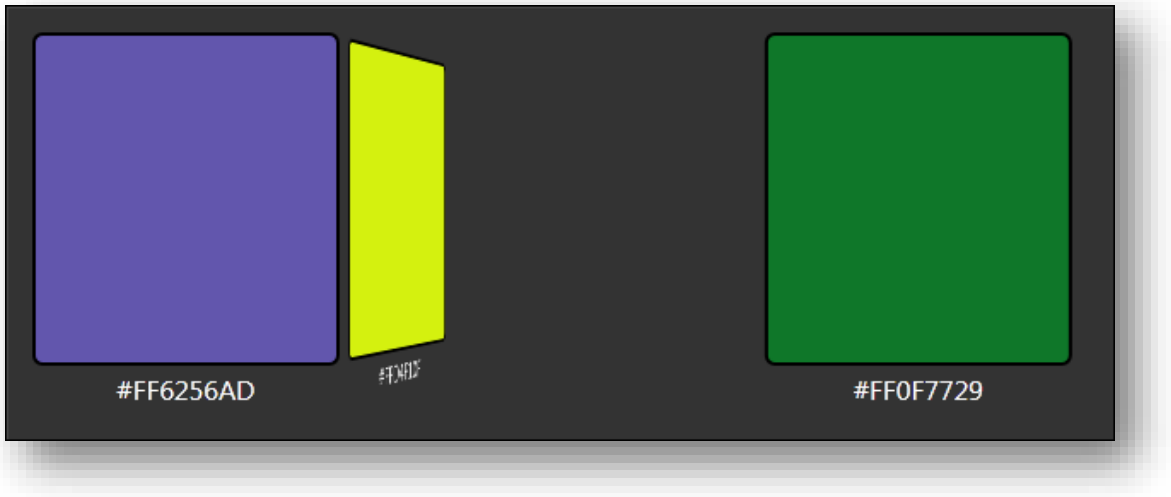
Grâce à ces nouveaux états vous pouvez contrôler ce à quoi ressemble un item juste après s'être chargé, juste avant d'être chargé ou juste avant qu'il ne soit déchargé (retiré de la liste).

XAML créera automatiquement les animations voulues lorsque ces transitions auront lieu, c'est à dire quand des items seront ajoutés ou retiré de la liste.

La fonction est puissante mais bien cachée. Plus vicieux, elle ne fonctionnera que si vous ajoutez un `FluidMoveBehavior` au template de l'`ItemsPanel` en n'oubliant pas de fixer sa propriété `AppliesTo` à la valeur `Children`. Sinon les autres items (en dehors de celui concerné par sa propre entrée ou sortie) ne seront pas animés. Les `LayoutStates` s'entendent donc comme un complément au `FluidMoveBehavior` pour créer un effet visuel complet.

Pour compliquer un tout petit plus le tableau, il faut signaler que si l'`ItemsPanel` est de type `VirtualizingStackPanel` la propriété `VirtualizinStackPanel.VirtualizationMode` de la `Listbox` devra être fixée à la valeur "`Standard`".

Ce n'est pas très simple tout cela, mais en le pratiquant on comprend mieux. Et puis il faut se rappeler qu'on est là dans le raffinement, dans des comportements visuels complexes qui étaient, avant ces améliorations, réellement très difficiles à coder.



XAML 81 - Effet des LayoutStates - capture écran

Ci-dessus, un exemple de `LayoutStates` utilisés pour permettre aux items entrants d'arriver dans une sorte de basculement 3D.

TransitionEffects

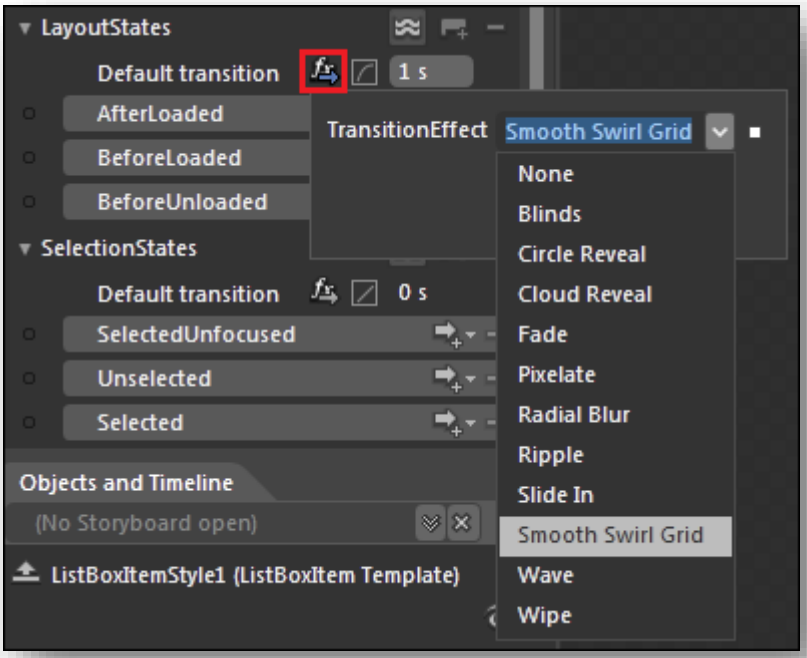
La V3 de Blend ajoutait le `FluidLayout` qui permet d'animer des changements d'état qui n'étaient pas "animables", comme par exemple les propriétés discrètes de type `Visibility`. La solution retenue utilise un morphing entre l'image de départ et celle d'arrivée.

Dans Blend 4 l'équipe de développement a voulu aller un cran plus loin dans cet esprit. Ainsi apparaissent les `TransitionEffects`.

Si vous pensez aux logiciels d'édition vidéo, vous connaissez sûrement les effets de transition qu'on peut ajouter entre deux séquences ou deux images (pour faire un slideshow à partir d'un album photo par exemple). Dans ces logiciels les transitions sont des calculs basés sur les pixels constituants les images des deux séquences entre lesquelles prend place la transition. Avec les `TransitionEffects` de Blend 4, on obtient dans le même esprit un calcul de transition basé sur les pixels pour assurer le passage d'un état à un autre.

Prosaïquement, un `TransitionEffect` sous Blend 4 est un `PixelShader` (comme l'ombre portée, le floutage…) qui a une propriété `Progress` animable.

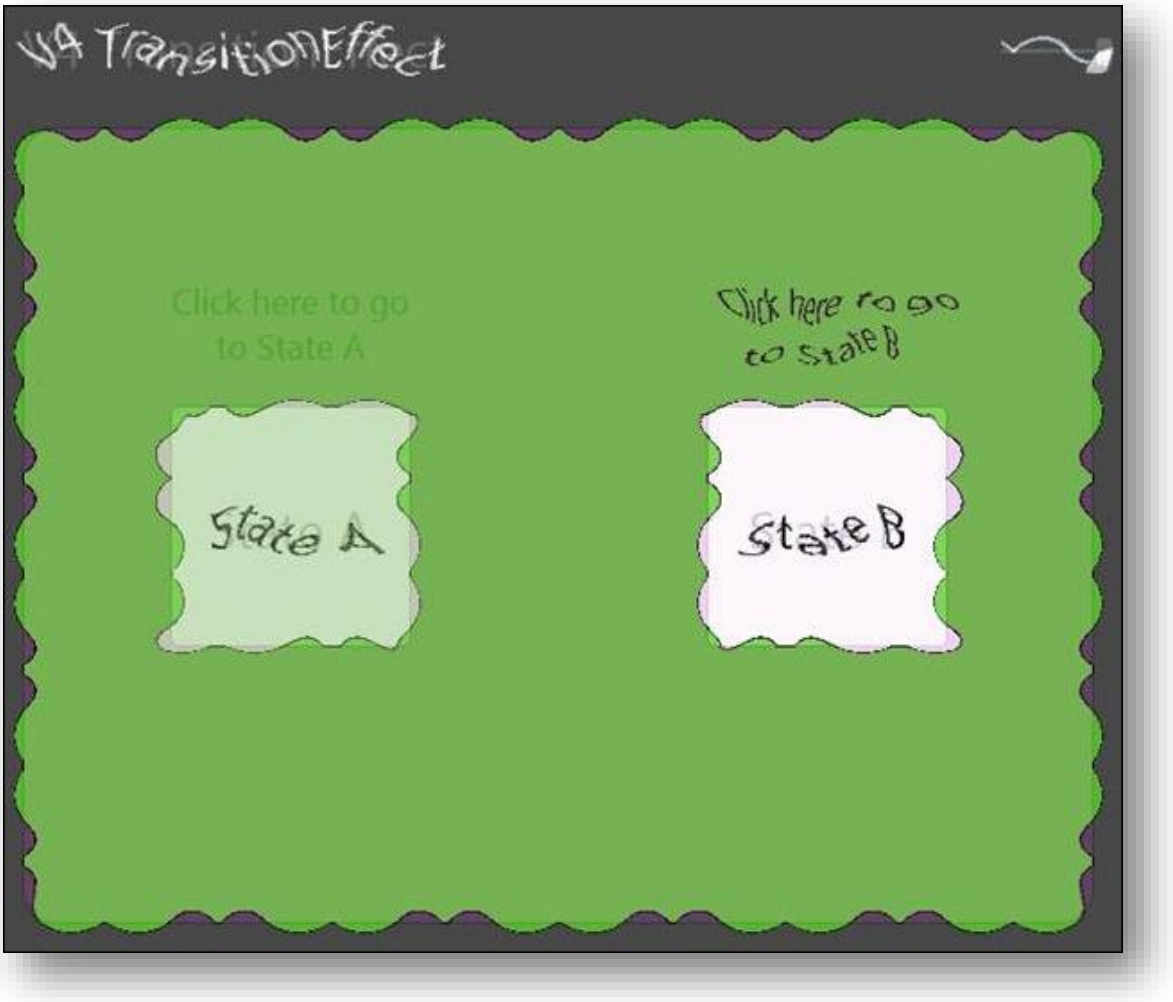
Blend 4 est livré avec plusieurs effets de transitions mais comme pour les PixelShaders, si vous savez coder en HLSL vous pouvez écrire les vôtre et les ajouter à votre palette !



XAML 82 - Pixel Shaders sous Blend

Ci-dessous on voit comment la transition par défaut du groupe "`LayoutStates`" d'un contrôle est fixé à "`Smooth Swirl Grid`" ("grille de tourbillons lisses'"), sur une durée de 1 seconde avec un easing cubique… Quand on pense à la complexité de ce qui est accompli par ces quelques clics on mesure à quel point Blend est un outil essentiel et pourquoi je me tue à répéter que Visual Studio n'est pas l'outil adapté au développement de la partie visuelle d'applications XAML malgré ses nombreuses améliorations.

L'image ci-dessous capture l'effet réglé plus haut en pleine transition (on devine le morphing entre les images de départ et d'arrivée et on voit nettement l'effet de "smooth swirl grid") :



XAML 83 - Catpure d'une transition par Pixel Shader

FluidMoveTagSetBehavior

Ici on entre dans le jardin secret de la team Blend, son fantasme fou, en passe d'être réalisé, de pouvoir faire les animations les plus extravagantes sans aucun code !

Selon l'équipe elle-même, cela faisait 5 ans qu'il cherchait une solution à un problème tout bête : ils s'étaient aperçus, perspicaces qu'ils sont, que très souvent des éléments sont animés d'un endroit à l'autre d'une application mais que ces éléments ne sont pas seulement des petits dessins ou des contrôles posés sur le artboard, puisqu'ils peuvent aussi être générés depuis des données.

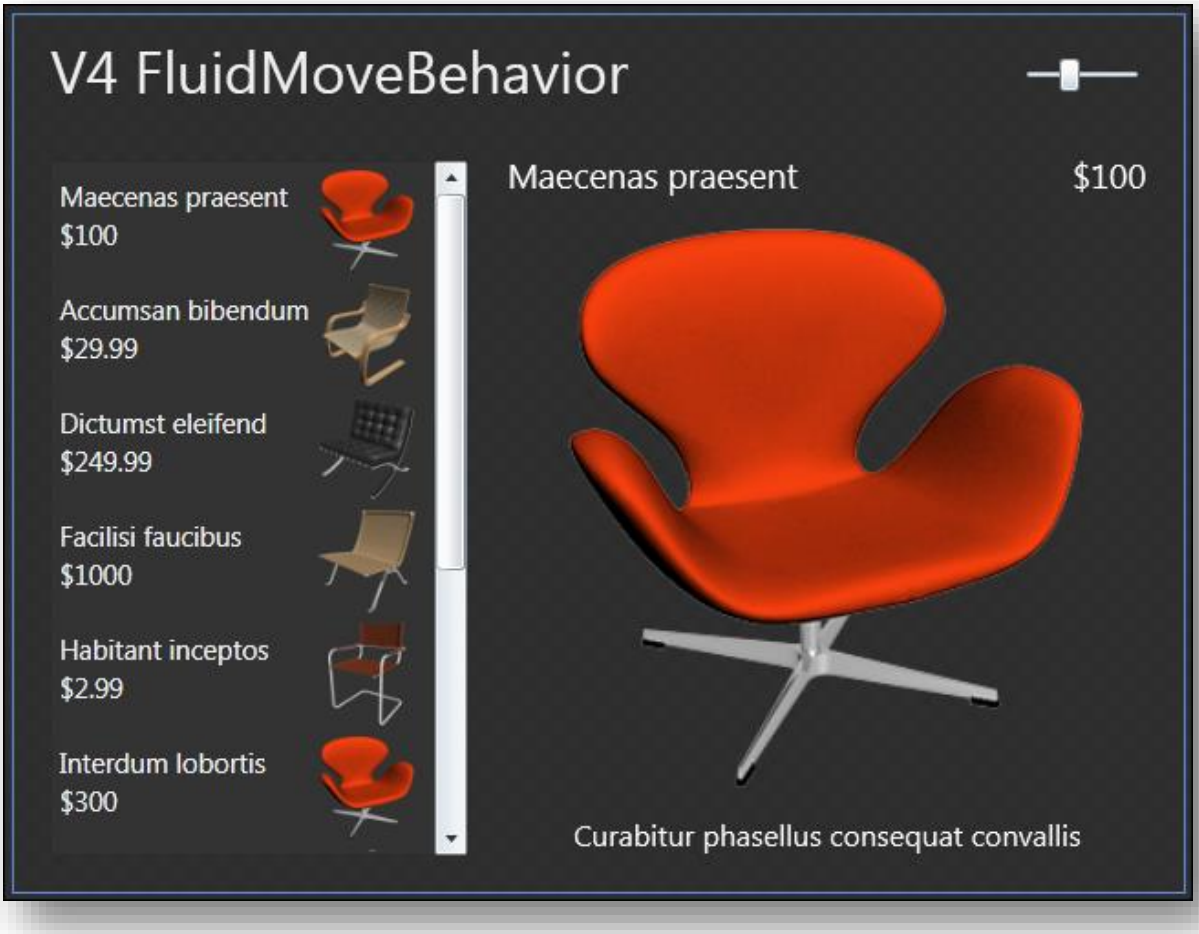
Seulement voilà, dans le paradigme MVVM qui s'impose sous XAML, le Model (les données) ne doit absolument rien savoir à propos du visuel (View). Et cela peut devenir très sportif d'essayer de programmer ce type particulier d'animations sans casser le modèle MVVM et sans truffer le ViewModel de code qui n'a normalement rien à y faire.

La solution retenue pour Blend 4 consiste à faire en sorte que ce soit le visuel qui, d'une façon ou d'une autre, en sache plus sur les données qu'il traite. Ce qui est parfaitement licite en MVVM (le visuel doit même forcément connaître les données puisque sa raison d'être est de présenter convenablement ces dernières…).

C'est en rendant le `FluidMoveBehavior` plus souple que la solution est venue : en lui permettant d'associer des positions non plus seulement avec des éléments visuels mais aussi avec des données.

Dit comme ça, c'est assez flou, j'en conviens, toutefois c'est très facile à utiliser, une fois qu'on a compris la façon de procéder.

Voici l'image d'une application exemple fournie par Microsoft et illustrant ce nouvel effet visuel :



XAML 84 - Capture FluidMoveBehavior

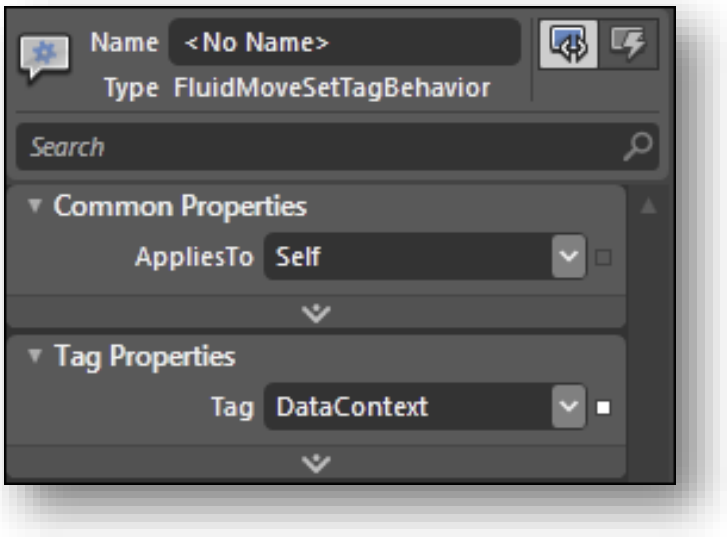
Ici nous avons, à gauche, une `ListBox` contenant des données avec un `ItemTemplate` personnalisé montrant le nom et le prix de l'article ainsi qu'une miniature de celui-ci.

A droite nous avons un panneau de détail affichant l'article sélectionné en grand, avec des explications en plus etc (on pourrait supposer un site marchand, l'ajout au panier…).

Le but du jeu est de faire apparaitre l'article qui se trouve dans la partie détail par une animation depuis sa miniature, comme si cette dernière grossissait tout en venant se placer dans la partie de droite.

La `ListBox` est alimentée par des données "vivantes", le système d'animation doit donc s'adapter à cette situation particulière, c'est tout le challenge de cette nouveauté.

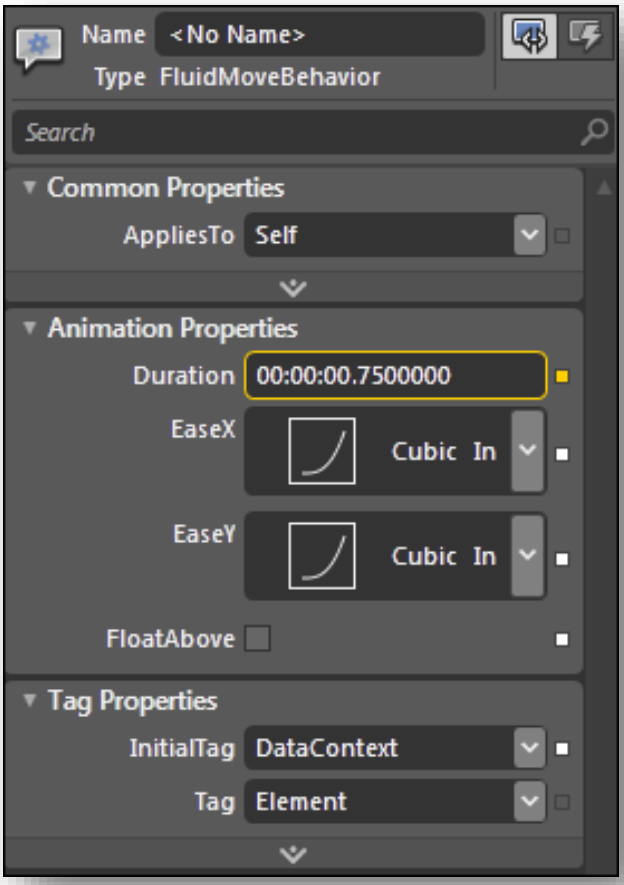
Et comme je le disais plus haut, en réalité c'est très simple à réaliser. Dans un premier temps il faut localiser l'élément à animer dans l'`ItemTemplate` de la `ListBox` (la miniature) puis de lui ajouter un `FluidMoveTagSetBehavior`. Ce behavior va avoir pour rôle d'enregistrer la miniature dans le moteur de `FluidMove`. Placer le behavior se fait comme suit :



XAML 85 - Réglage du FluidMoveBehavior

La propriété `Tag` du behavior indique "`DataContext`", ce qui signifie que l'item sera marqué en fonction de son `DataContext` qui est le modèle se cachant derrière le visuel. Ce n'est donc pas le contrôle image lui-même qui est tagué mais la donnée qui se cache derrière.

Maintenant il faut prendre l'image se trouvant dans la partie détail de l'écran (la grande image de droite) et lui ajouter un `FluidMoveBehavior` :

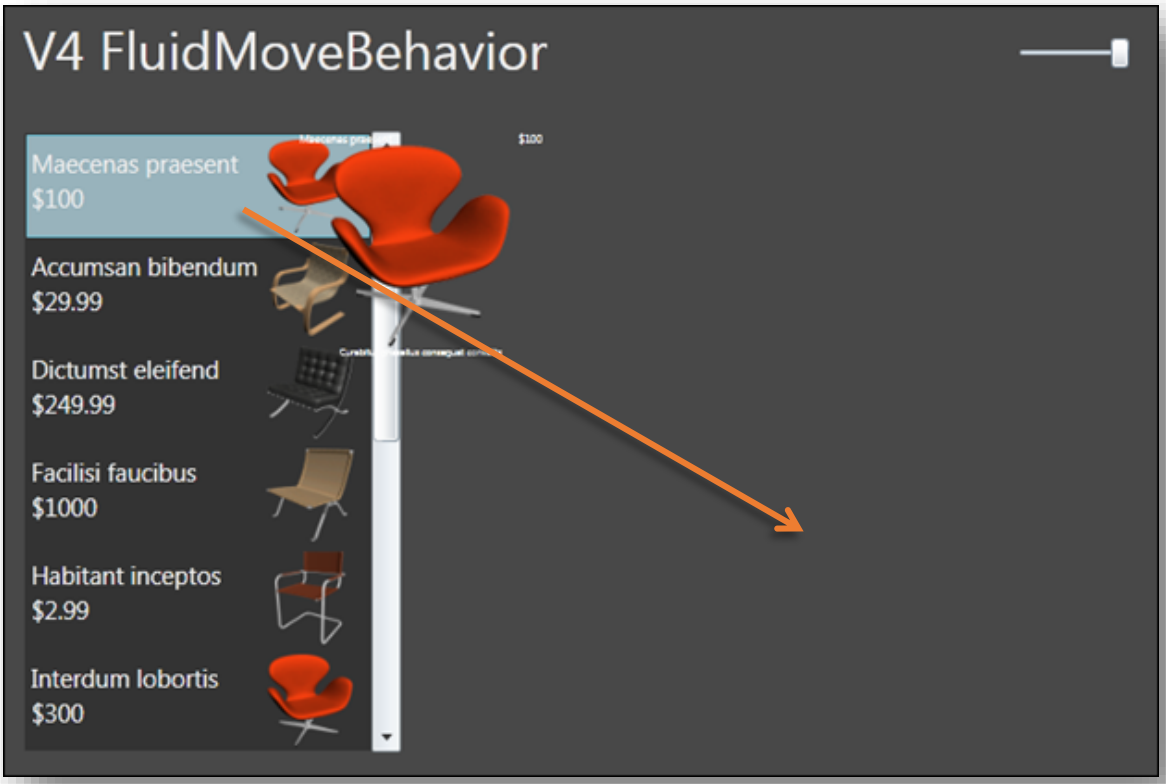


XAML 86 - Les animations du FluidMoveBehavior

C'est là qu'entre en jeu la partie "`Tag Properties`" de ce behavior… `InitialTag` pointe aussi le `DataContext` (qui est le même pour la liste et la fiche détail). Cela signifie que lorsque l'item du détail apparaitra il devra prendre en considération comme point de départ la position enregistrée de son `DataContext`. La connexion est ici effectuée entre l'image détail et la miniature de la `ListBox`. En réalité la liaison s'opère, comme on le voit, entre Le `FluidMoveBehavior` de la grande image (détail) et le `FluidMoveTagSetBehavior` qui décore la miniature dans le data template des items de la Listbox.

Et le tour est joué !

L'image ci-dessous montre comment la chaise rouge semble partir depuis la position de l'item sélectionné dans la `ListBox` pour rejoindre sa place à droite de l'écran :



XAML 87 - Effet d'un FluidMoveBehavior

On s'en doute un peu, avec une poignée de propriétés c'est une animation d'une grande complexité qui est réalisée, et le travail de développement derrière ces effets visuels est à la hauteur de cette sophistication. Automatiser de façon générique ce type d'animation réclame beaucoup de code, mais surtout beaucoup de savoir-faire. On veut bien croire que l'idée a pu nécessiter 5 ans avant de prendre forme.

Bien entendu on ne voit ici qu'un simple exemple. Le procédé est utilisable de mille façons, comme par exemple pour donner l'impression qu'un élément passe d'une `ListBox` à une autre.

Plus fou ?

Si, ça existe. Dans Blend 4 aussi. Un truc complètement délirant. Ça s'appelle la `PathListBox`.

C'est tellement fou qu'il m'est bien impossible de résumer ses possibilités sans doubler la taille de ce post déjà bien long… La `PathListBox` est une `ListBox`, certes, mais en fait c'est autre chose. Elle possède bien des `items`, mais elle sait les animer, en rond, en carré, en suivant un tracé vectoriel quelconque, elle peut servir à animer

un objet le long d'un chemin (plus rien à voir avec une `listbox` donc), ou bien à présenter les lettres d'un mot en arrondi, en vague, en n'importe quoi (loin d'une `listbox` encore). Bref c'est un contrôle délirant. J'essayerai d'en parler prochainement, quand je serai sûr d'avoir tout compris de ses ruses (elles sont nombreuses) et si j'arrive à condenser ça de façon intelligible. Il existait un tutor assez complet sur le site ".toolbox" qui depuis la « purge » anti-Silverlight de Sinofsky a disparu. Dommage. Mais en fouillant bien, on peut retrouver un autre tutor de même type sur ce le site .toolbox « new look ».

Conclusion

La mise en page dynamique et les transitions sont des piliers visuels des applications XAML. Savoir donner de l'importance aux états de l'application, à l'apparition ou la disparition d'éléments, aux changements de forme, de position, de couleurs de blocs visuels ou d'éléments isolés, tout cela s'appelle le Design. Loin d'être juste un embellissement graphique, le Design fait partie du cycle de développement d'un logiciel moderne dans le sens où il s'attache à trier les informations essentielles, à donner un sens aux états, une lisibilité aux données produites par le code.

Il n'y a pas d'un côté le travail sérieux que serait le codage et de l'autre une tâche accessoire, supplémentaire et un peu futile que serait le Design. Je croise encore trop souvent des développeurs qui sont dans cet état d'esprit. C'est peut-être d'ailleurs pour ça qu'ils ne sont que développeurs dans la hiérarchie, un manque de vision, la peur d'être détrôner, que sais-je… Mais encore une fois ne les blâmons pas. Tous ne pourront être sauvés mais en expliquant et en expliquant encore je crois que certains finiront par comprendre.

Quant à ceux qui ont déjà compris, j'espère que ce petit tour d'horizon de la puissance de Blend leur donnera envie de tester au plus vite ces effets pour les mettre en pratique, et surtout, au service des données que leurs applications produisent !

Les Pixel shaders (effets bitmap)

Les Pixel shaders sont des effets bitmap qui peuvent être appliqués comme des « filtres PhotoShop » par-dessus les objets graphiques vectoriels de XAML. Ils sont supportés par WPF, Silverlight et WinRT.

Leur utilisation permet d'obtenir des effets visuels saisissants, mais bien entendu cela se fait au prix de calculs qui peuvent peser sur la fluidité de l'application. Certains environnements XAML proposent des modes d'accélération GPU. WPF le fait quasiment tout seul, Silverlight offre quelques possibilités mais ne s'appliquant pas aux pixel shaders, Windows Phone ou WinRT offrant des approches différentes (en raison du Windows Runtime assez différent dans sa construction du framework .NET et aux différences dans l'accès à DirectX). On prendra ainsi toujours soin de vérifier l'impact des pixel shaders sur la fluidité de ses applications en fonction de la cible XAML choisie.

Les Pixel Shaders

En raison certainement de leur impact sur le CPU / GPU, les pixel shaders ne sont pas forcément mis en avant, ni très nombreux de base. Par exemple lorsqu'ils ont été introduits dans Silverlight 3 on ne pouvait compter que sur peu d'effets : principalement le blur (flou) et le drop shadow (ombre portée). Mais on trouve sur CodePlex un très beau projet offrant toute une série de nouveaux effets comme le swirl ou l'emboss (WPF Effects Library), bibliothèque fonctionnant à la fois avec WPF et Silverlight et dont on pourra s'inspirer, au moins pour le code HLSL pour d'autres environnements. Grâce au SDK Directx et au langage HLSL il possible de développer ses propres effets utilisant les possibilités graphiques de DirectX.

Le langage en lui-même n'est pas bien compliqué un fois qu'on en a compris l'esprit. Mais on peut très bien se limiter aux effets existants, bien utilisés ils peuvent déjà apporter beaucoup visuellement.

L'exemple ci-dessous illustre mon propos : la fenêtre principale est décorée d'une magnifique photo d'éclair (personnelle, donc pas de problème de copyright) sur laquelle un flou peut être appliqué grâce à un slider se trouvant dans une petite fenêtre semi transparente sur laquelle est appliqué un drop-shadow.

Bien entendu, en cliquant sur le titre de ce chapitre du livre vous aurez accès au billet original dans lequel se trouve l'exemple Silverlight directement utilisable ce qui est plus plaisant que les captures écran proposées ici.

Figure 92 - capture écran - Appliquer un flou en temps réel

Figure 93 - capture écran - Flou appliqué, palette déplacée

Vous remarquerez que la petite fenêtre en question peut être déplacée par drag-drop grâce à une astuce de XAML, les Behaviors (comportements). Les Behaviors ont été présentés plus haut dans ce livre je n'en dirais donc pas plus ici.

Le slider a été quant à lui templaté "à l'arrache" mais il a été templaté tout de même, créant ainsi un look personnalisé pour cet exemple.

Autre capacité de XAML qui est ici démontrée : le Databinding entre éléments d'interface, appelé *Element Binding*, ainsi le texte indiquant la quantité de flou appliquée est directement lié à la propriété `Value` du Slider. Pour formater cette valeur un convertisseur a été ajouté.

Bref, une petite application vite fait pour montrer les Pixel Shaders, mais pas seulement...

Le mieux étant maintenant de jouer avec et de télécharger le projet (VS2008 + SL3 toolkit ou Blend 3 minimum) : PixelShader.zip (168,29 kb)

Composer des effets pixel shaders

Lorsqu'on joue un peu avec les pixel shaders on tombe assez vite sur une limite qui semble infranchissable : les objets n'acceptent qu'un seul effet, et placer un nouvel effet ne s'ajoute pas au premier mais le remplace. Damned ! Bien entendu il existe une solution...

L'astuce est simple alors je n'en ferai pas des tonnes : le principe consiste à imbriquer l'objet dans des containeurs et à appliquer chaque effet sur chaque containeur...

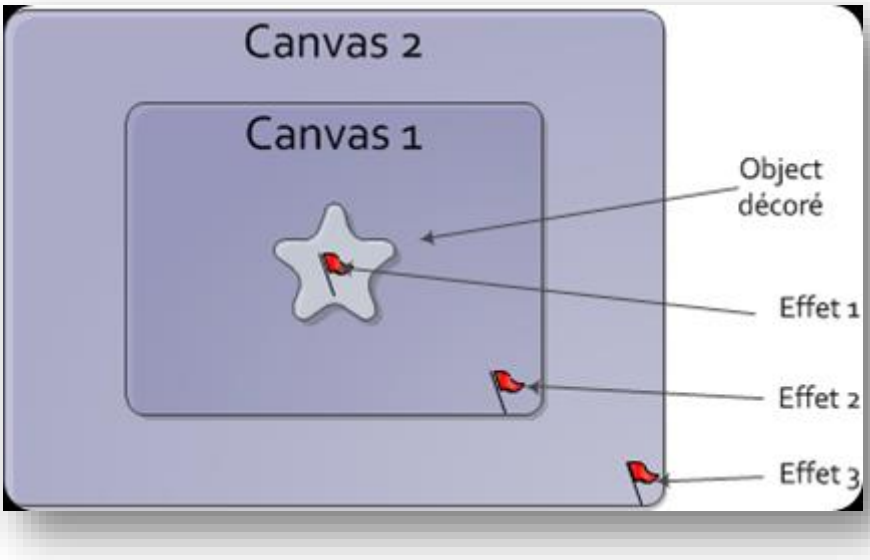


Figure 94 - Composition des effets visuels

L'objet décoré (l'étoile centrale) est agrémentée de l'effet 1, puis on englobe l'étoile dans un canvas sur lequel on applique l'effet 2, puis on englobe le canvas dans un autre canvas sur lequel on applique l'effet 3, etc, etc …

L'ordre dans lequel les effets sont appliqués peut avoir un impact sur le visuel mais une fois les "n" niveaux de "poupées russes" créés, il suffit juste de changer les effets appliqués à chacun pour tester la combinaison qui donne le meilleur résultat.

Conclusion

C'est simple, facile à faire (sous Blend on sélectionne un objet et on demande « grouper dans un canvas », sous XAML on entoure tout simplement d'une balise `Canvas`), et ça permet d'appliquer autant d'effets que nécessaire.

N'ayez pas la main trop lourde, les pixel shaders consomment du calcul, et vérifiez toujours l'impact en jetant un œil à l'outil de performance de Windows ou en testant sur des devices réelles et non des émulateurs.

Largeur de DataTemplate et Listbox

Une petite astuce rapide en passant…

Lorsque vous créez un `DataTemplate` qui sera utilisé comme `ItemTemplate` d'une `ListBox` vous avez certainement rencontré ce très désagréable problème : vos items sont bien affichés, cadrés à gauche, mais leur largeur est variable, elle s'adapte au

contenu. Si votre template est en fond blanc sur une listbox en fond blanc aussi, on n'y verra que du blanc… Mais si votre template possède un fond ou un border par exemple, la listbox aura un aspect affreux avec des items en escalier, cadrés à gauche, mais avec une partie droite irrégulière…

En général on arrive à un résultat qui ressemble à cela :

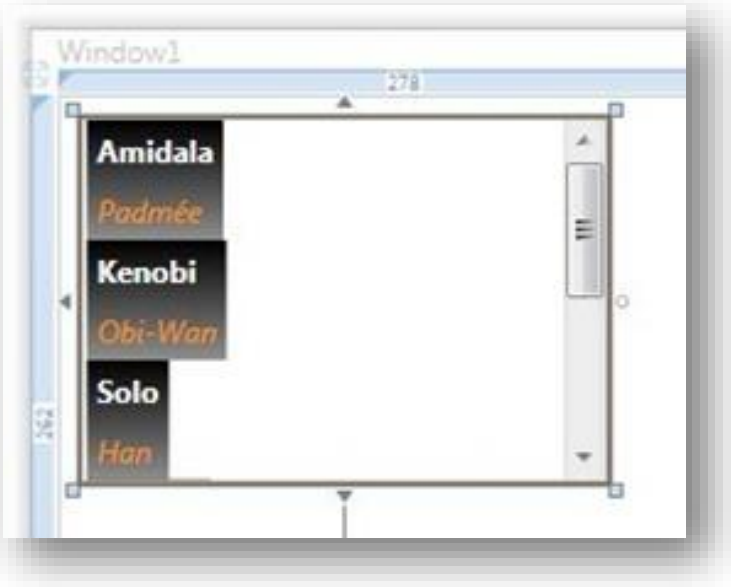


Figure 95 - Liste avec items templatés non cadrés

C'est franchement laid… Alors qu'on voudrait obtenir ça :

Figure 96 - Liste avec items templatés et cadrés

Ce qui est déjà beaucoup plus agréable…

Le problème ne vient pas du `DataTemplate`, arrêtez de le martyriser, le pauvre n'y est pour rien !

J'ai vu des développeurs essayer de faire des Bindings relatifs de type `FindAncestor` pour remonter sur la listbox et se binder à son `ActualWidth` ou d'autres combines de ce genre qui ne donnent pas même entière satisfaction. Le problème est ailleurs… Laissez le `DataTemplace` et son contenu en mode "auto", c'est ce qui est le mieux.

Pour résoudre cet épineux problème il faut comprendre que le `DataTemplate` n'est qu'un modèle et que lorsqu'il est instancié pour chaque item il est placé dans un conteneur généré pour l'occasion. C'est ce conteneur qui pose problème car il a un cadrage à gauche et non `Stretch` par défaut.

Comment l'atteindre et le modifier ? La méthode lourde consiste à modifier le Style de la listbox pour atteindre la propriété `ItemContainerStyle` et modifier ce dernier (un style aussi) afin de mettre `HorizontalContentAlignment` à `Stretch`.

Si vous devez créer un style pour votre `ListBox`, alors c'est le meilleur moyen. Sinon il existe une version courte : ajouter uniquement dans la listbox une modification directe de la propriété en question :

```
<ListBox ItemsSource="{Binding}"  Name="listBox1"
   ItemTemplate="{DynamicResource PersonneTemplate}"
   HorizontalAlignment="Left">
     <ListBox.ItemContainerStyle>
       <Style TargetType="ListBoxItem">
        <Setter Property="HorizontalContentAlignment"
                Value="Stretch"></Setter>
       </Style>
     </ListBox.ItemContainerStyle>
</ListBox>
```

XAML 88 - Correction du cadrage des items templatés

La partie en gras (`ItemContainerStyle`) est celle qu'il faut ajouter à la définition de la listbox qui supportera un `ItemTemplate` personnalisé. Et vous voilà avec une belle listbox dont les items sont bien alignés et qui s'étalent sur toute la largeur de la listbox. Le tout sans bricolage, sans toucher au template, et en 4 lignes de Xaml.

Masques d'opacité (OpacityMask)

Les masques d'opacité sont des outils très puissants que l'on retrouve dans de nombreux logiciels gérant de l'image (ou même de la vidéo). XAML offre un support de cette fonction qui peut rendre bien des services…

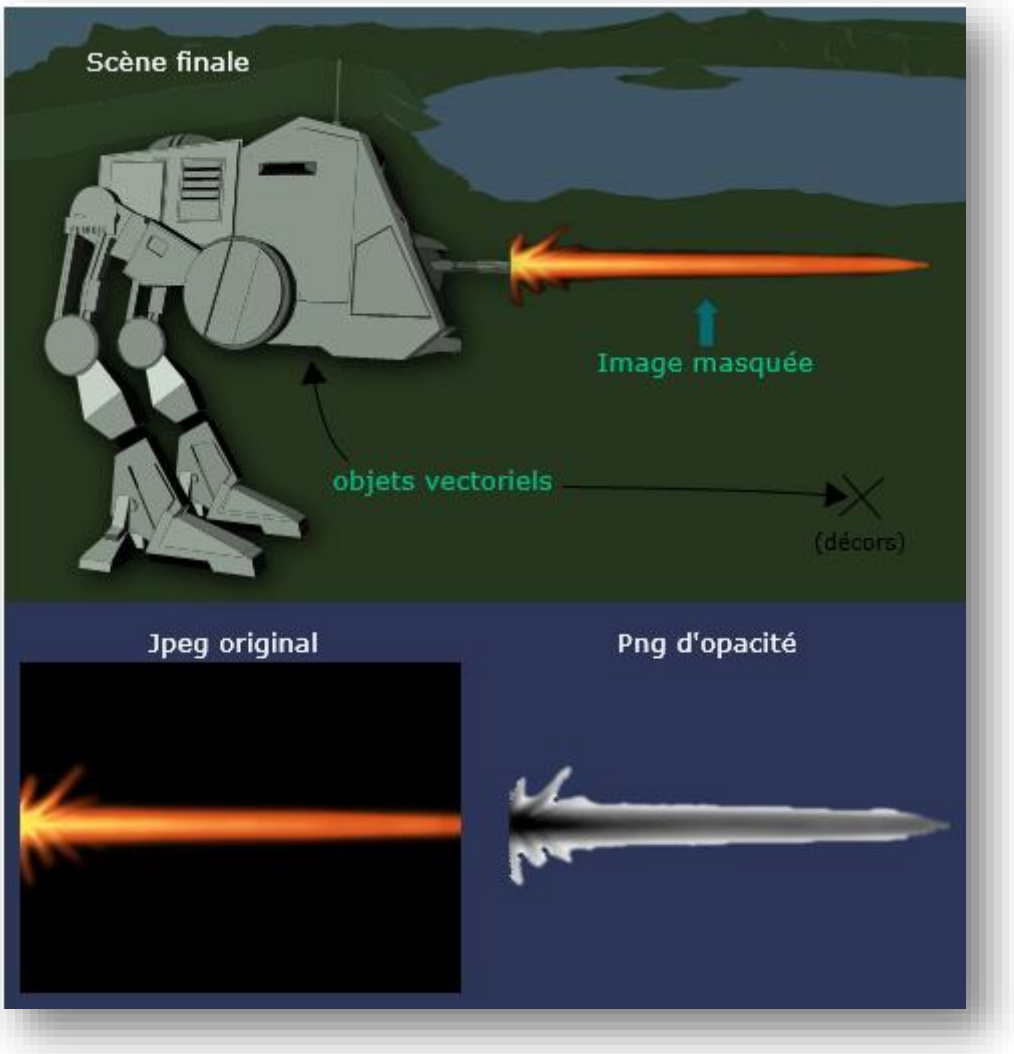


Figure 97 - capture de scène de jeu fictif utilisant les masques d'opacité

Une scène de jeu

Imaginons la scène de jeu ci-dessus. Grâce à Swift 3D j'ai créé un fichier Xaml de deux objets 3D, le célèbre ATP de Star Wars et un décor de fond (un lac et de la verdure). Ces fichiers Xaml (un canvas généré automatiquement par Swift) ont été placés dans

des `UserControl` séparés permettant de les manipuler facilement ensuite. Mais là n'est pas vraiment la question. Ces images étant déjà en Xaml, on s'aperçoit que les transparences sont gérées "de fait". Par exemple on voit le décor entre les "pattes" de l'ATP. Mais pour le tir laser j'ai récupéré un Jpeg… et là les choses se compliquent.

Créer des transparences

Comment, à partir d'un Jpeg (en bas à gauche dans l'exemple) obtenir le tir laser de la scène finale (qui se découpe parfaitement sur le décor) ?

Deux approches : la première consiste à utiliser un programme de dessin pour créer un PNG ayant les bonnes transparences. C'est certainement la meilleure méthode pour créer un sprite par exemple. Mais d'une part c'est un travail d'infographiste (hors sujet ici) et cela ne répondrait pas à tous les besoins que la seconde solution permet de couvrir.

Soit on créé un masque d'opacité qui sera utilisé dans la scène finale.

La première méthode ne nous intéresse pas trop ici, laissons les infographistes à leur métier. C'est la seconde voie qui est la plus intéressante car elle ouvre la porte à de nombreuses astuces de présentation.

Pas seulement pour les jeux !

Bien entendu dès qu'on parle d'images, de transparences, de décors, etc, on pense "jeu". C'est bien entendu un domaine où le travail de l'image est une obligation incontournable. Mais sous XAML créer des interfaces graphiques de qualité passe aussi par la manipulation d'images et de vecteurs !

Fondre une image dans une autre, créer une bordure un peu originale, fabriquer un titrage comme une trouée sur une image de fond, etc, tous ces effets réclament de gérer… des masques d'opacité. Et même un logiciel de gestion peut tirer parti de toutes ces possibilités graphiques, grâce à Xaml.

OpacityMask

Le masque d'opacité est une propriété qu'on retrouve notamment dans les `UserControl`. Il s'agit de fournir un dessin (image PGN ou vectorielle) dont l'opacité (le canal Alpha) sera utilisé pour gérer la transparence de l'objet auquel il sera

accroché. Le dessin en lui-même n'a guère d'importance, c'est l'opacité de la couleur qui compte (son canal Alpha donc).

Dans notre exemple je suis parti d'un Jpeg d'une flamme sur fond noir. J'ai transformé le Jpeg en PNG parce que j'aime mieux ce format dans un tel cas, mais ce n'était pas nécessaire. Ensuite, en utilisant un logiciel de dessin, j'ai "bricolé" l'image originale pour obtenir la seconde (en bas à droite dans l'exemple) qui elle, par force (pour gérer la transparence), est un PNG. Le but du jeu étant de retrouver la silhouette de la flamme (opaque) et de rendre tout le reste transparent.

Une fois cette opération terminée les deux images sont importées dans le projet.

On place l'image originale puis on applique la seconde image comme masque d'opacité, c'est aussi simple que ça…

Sous Blend on clique sur la propriété `OpacityMask` de l'image puis on utilise l'onglet "`Tile Brush`", puis on sélectionne l'image gérant le masque comme s'il s'agissait d'une brosse à appliquer. C'est tout.

En Xaml cela donne :

```
1: <Image Source="images/flare008.png" >
2:     <Image.OpacityMask>
3:         <ImageBrush ImageSource="images/flare008b.png"/>
4:     </Image.OpacityMask>
5: </Image>
```

Figure 98 - Application d'un masque d'opacité

Quelques astuces

L'utilisation de Swift 3D est une "grosse" astuce ! Il s'agit d'un logiciel très complet permettant de créer des objets et des animations 3D de façon "abordable". Les logiciels de type Maya, LightWave (ou Blender si on préfère le gratuit), sont des monstres à maîtriser et il ne faut faire que ça tous les jours pour les comprendre. Swift est certes assez limité mais il est assez puissant pour faire des choses utilisables. Il possède deux énormes avantages sur tous les autres logiciels de 3D : d'une part il est assez facile à comprendre (et à se souvenir quand on ne fait pas que ça tous les jours) et d'autre part il sait générer des `Canvas` XAML, voire des animations 3D (en trichant sur les opacités de plusieurs images fixes superposées).

Grâce à Swift 3D il est possible de créer des éléments d'interface avec un look 3D et de les intégrer à ses applications.

L'autre astuce consiste à utiliser des bibliothèques d'images faites pour les softs de 3D. Chaque image est généralement, et au minimum, accompagnée de son *Bump* (une technique permettant de créer du relief sur des textures). La création du masque d'opacité est généralement plus rapide si on part d'un Bump qui a déjà été traité et simplifié que de l'image originale en couleur…

The sky is the limit !

La technique est simple, savoir bien l'utiliser et quand l'utiliser pour élargir son champ de vision et créer des interfaces encore plus sexy est une autre question … La seule limite est votre imagination !

Contrôles et zone cliquable, astuce…

Une petite astuce toute simple (une fois que l'on connaît la réponse).

Un contrôle très basique pour commencer

Prenons un contrôle, un `UserControl` carré avec un joli cercle à l'intérieur. Pour que tout cela soit facile à comprendre et à tester, disons que le `UserControl` fait 200x200 pixels, et que l'ellipse à l'intérieur remplie toute la zone (c'est un cercle donc). Cette ellipse aura un `Stroke` noir par exemple et une épaisseur de trait de `12`. Le `LayoutRoot` sera un `Canvas`.

Donc en gros nous avons dessiné un cercle noir sur fond transparent.

"Hou la ! Mais c'est du super haut niveau tout ça !" … On ne s'inquiète pas, ça va très vite se compliquer …

Première petite amélioration : nous allons mettre un curseur personnalisé à notre contrôle. Dans les propriétés du `UserControl`, propriété `Cursor`, on choisit `Hand`, (la main). Et on exécute (F5).

Ca se complique…

Passer la souris au-dessus de votre magnifique contrôle : la main s'affiche bien, mais uniquement lorsque votre souris survole le `Stroke` du cercle ! Rien à l'intérieur.

Qu'est-ce qu'il se passe ?

Tout ce qui n'a pas une brosse n'existe pas pour la souris. Notre contrôle et le `Canvas` sous-jacent n'ont pour l'instant aucune couleur de fond (`Background`). L'`Ellipse` n'a pas de `Fill` non plus. Tous ces espaces "sans rien" ne sont pas détectés. Bug ou feature, après tout ça peut être l'un ou l'autre selon le point de vue...

Mais comment régler le problème ?

En fait qu'importe qu'il s'agisse d'un problème ou d'un choix délibéré des concepteurs, nous devons remplir ces "riens" si nous voulons que le curseur de souris (et la zone cliquable du contrôle) apparaisse sur toute la surface.

Mais il est hors de question d'aller placer une brosse, donc une couleur, pour faire ce travail ! On pourrait bien entendu mettre un `Background` blanc au `Canvas`, cela règlerait le problème. Mais si nous changeons le fond de notre page, nous allons obtenir un cercle dans un carré blanc sur un fond différent. Une horreur, ce n'est pas le dessin que nous avons fait. Quant à s'enliser dans l'erreur en faisant en sorte de trouver une astuce pour que le fond de notre `UserControl` soit synchronisé avec celui de son conteneur parent... je préfère ne pas en parler !

Une "fausse" brosse

Il faut remplir notre `Ellipse` pour que le curseur soit visible et que même son espace intérieur soit cliquable. Mais pas avec n'importe quoi. XAML nous fournit une brosse un peu spéciale, disons une "non brosse", ou une "fausse" brosse puisque visuellement elle ne se voit pas. Il s'agit de la brosse "`Transparent`".

Dès lors, il suffit d'ajouter dans le code XAML de notre ellipse `Fill="Transparent"` et de relancer l'application. Miracle ! Le curseur en forme de main est visible même à l'intérieur de l'ellipse. Et si nous changeons la couleur de fond de la page, comme cette brosse est transparente, on n'y verra que du feu ...

Sous Blend vous pouvez agir directement dans le code XAML ou bien simplement cliquer sur le carré de binding de la propriété `Fill` et choisir "*custom expression*". Tapez "`Transparent`" (sans les guillemets) dans le champ de saisie et l'affaire est jouée.

Comme quoi, même un `UserControl` ne contenant qu'un cercle peut déjà soulever des problèmes pas si simples à résoudre. Ça force à l'humilité !

Xaml Dynamique

Xaml n'est pas compilé, ce qui signifie que le runtime l'interprète à l'exécution. Cet interpréteur est donc présent aussi bien quand on exécute une application WPF que Windows Phone ou WinRT. Cela peut permettre des choses intéressantes comme la création de code Xaml dynamiquement au runtime…

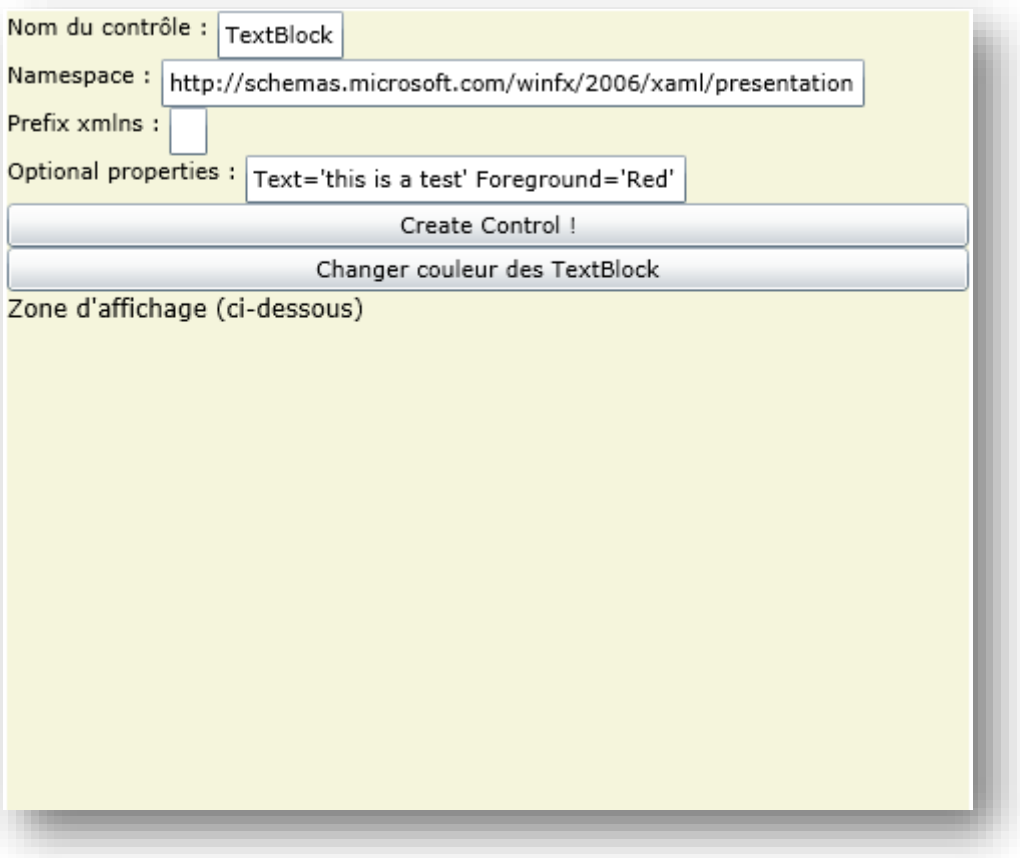


Figure 99 - capture du programme de test XAML Dynamique

Cette possibilité n'est que peu souvent évoquée alors que dans certains cas elle autorise de dynamiser tout ou partie d'une interface. Rien n'interdit en effet de créer une mise en page complète par ce procédé, VSM et animations comprises. La génération de Vues (partielles ou totales) en fonction d'une description paramétrée par l'utilisateur fait partie des cas concrets où de la génération dynamique de code Xaml peut grandement simplifier la mise en œuvre du code.

Le potentiel est grand, il suffit juste de savoir que cela existe car techniquement cela est finalement très simple. L'essentiel tient dans `XamlReader.Load()` qui sait rendre un objet à partir d'un code Xaml.

Pour sortir des sentiers battus, voyons comment on peut utiliser cette stratégie pour créer dynamiquement un composant à partir de son nom de classe non pas en utilisant la réflexion sous C#, juste en générant du code Xaml…

L'exemple live ci-dessous permet de créer un contrôle (pré-saisi par défaut pour un `TextBlock`) qui sera ajouté dans un `StackPanel` dans la seconde moitié verticale de l'affichage (l'exemple est en Silverlight pour être proposé en live sur Dot.Blog). Créez plusieurs `TextBlock` de cette façon en modifiant le texte ou d'autres paramètres. Puis créez-en un auquel vous ajouterez dans les paramètres: `Name='Test'`. Enfin cliquez sur le bouton "*changer couleur des TextBlock*".

Tous les `TextBlock` créés passeront en `Foreground` noir, et celui qui a pour non '`Test`' verra sa taille de fonte augmentée de 4.

Le code de création du contrôle est le suivant :

```
namespace DynamicXaml
{
 public static class DynamicXamlHelper
 {
   public static UIElement CreateControlFromName(string controlName,
                               string controlNamespace, string xmlnsPrefix,
                               string properties)
   {
     var sb = new StringBuilder();
     sb.Append("<" + controlName + " xmlns");
     if (xmlnsPrefix.Length > 0) sb.Append(":" + xmlnsPrefix);
     sb.Append("=\"" + controlNamespace + "\" " + properties + "/>");
     try
      { return (UIElement)XamlReader.Load(sb.ToString()); }
     catch
     { return null; }
    }
 }
}
```

Code 51 - Créer des éléments XAML par leur nom de classe

Les paramètres sont ceux que vous pouvez taper dans les `TextBox` de l'exemple live ci-dessus:

- le nom de la classe du contrôle, par exemple "`TextBlock`"

- le namespace du contrôle,
  ex: http://schemas.microsoft.com/winfx/2006/xaml/presentation
- le préfixe XML, vide dans l'exemple par défaut
- une zone libre de paramètres qui seront ajoutés à la balise de création.

Pour le reste, le plus simple est de jouer avec le code source de l'exemple : DynamicXaml.zip (7,08 kb)

Blend et les ressources de Design

Si vous créez des applications de type MEF ou bien dont certaines ressources sont chargées dynamiquement par d'autres procédés, vous savez à quel point cela pouvait être pénible en Design sous Blend qui, par force, ne trouvait pas les styles ou templates (puisque chargés au runtime). Il existait des ruses plus ou moins contraignantes, mais rien de bien pratique il faut l'avouer. *A partir de Blend 4 cela n'est plus qu'un souvenir* !

Design-Time Resource Dictionary

Quand vous ouvrez un projet dans Blend et que le document en cours possède des références qui ne peuvent pas être résolues, et si votre solution contient au moins un fichier ResourceDictionary, le dialogue "*Add Design-time Resource Dictionary*" sera ouvert :

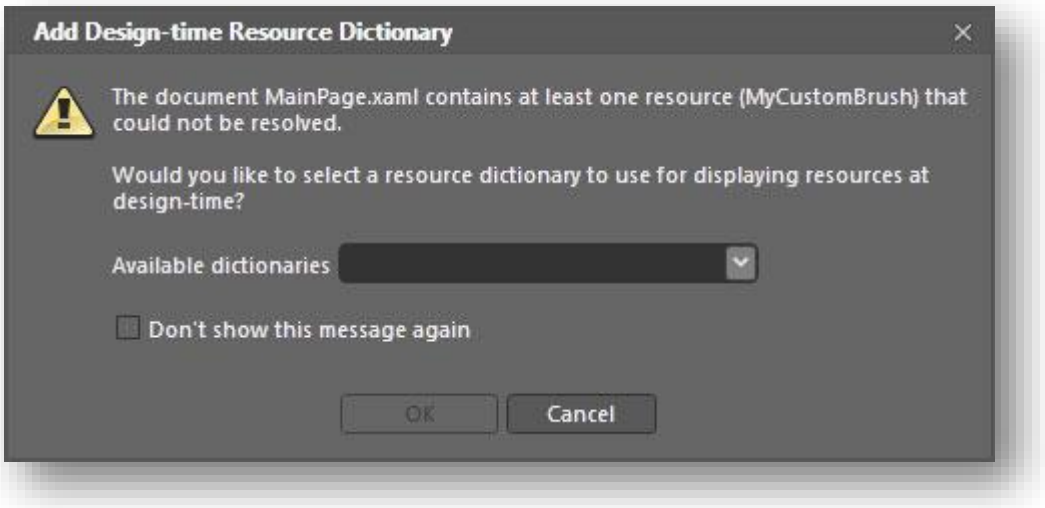


Figure 100 - Ajout d'un dictionnaire de ressources sous Blend

Une fois que la référence vers un dictionnaire répertorié dans le projet est sélectionnée un nouveau fichier est ajouté par Blend sous les Propriétés du projet : "`DesignTimeResources.xaml`". Il s'agit en fait un peu d'un centralisateur de dictionnaires comme `App.xaml` mais uniquement dédié au design donc sans aucune influence sur le logiciel compilé.

En revanche, des ressources chargées dynamiquement peuvent être ainsi "forcées" pour le Design. Fini les écrans qui ne correspondent pas aux styles prévus ! Le tout sans "bricolage".

En intervenant manuellement sur `DesignTimeResources.xaml` on peut bien entendu ajouter d'autres ressources ou en supprimer (dans ce dernier cas le dialogue ci-dessus reviendra s'afficher dès que vous ouvrirez un document pour lequel Blend ne trouve pas certaines ressources référencées).

Simple, facile, encore une astuce Blend pour rendre le travail du Designer plus plaisante, plus rapide et plus intelligible…

Hard ou Soft ?

Êtes-vous plutôt Hard ou Soft ? Ok, c'est trop personnel comme question. Mais si on la pose à une `ListBox` immédiatement on se sent plus à l'aise pour répondre…

La ListBox Hard et la ListBox Smooth

Ce n'est pas une fable de La Fontaine, mais une affaire de visuel sous XAML.

Pour la listbox d'ailleurs on parlera plutôt de hard et smooth. Une ListBox "soft", ça ferait trop montres molles de Dali ! Pour être d'ailleurs plus pointilleux, on opposerait Smooth à Rugged, Scratchy ou ici plutôt à Rough qu'à Hard. Mais heureusement pour certains on n'est pas là pour faire un cours d'anglais :-)

Regardez attentivement les deux ListBox de l'exemple live ci-dessous… Vous ne voyez vraiment pas de différence ? Normal, il n'y en a pas, c'était un piège :-) En fait si, bien sûr, il y a une différence, mais elle ne saute pas aux yeux, il faut manipuler les ascenseurs pour s'en rendre compte. Allez-y…

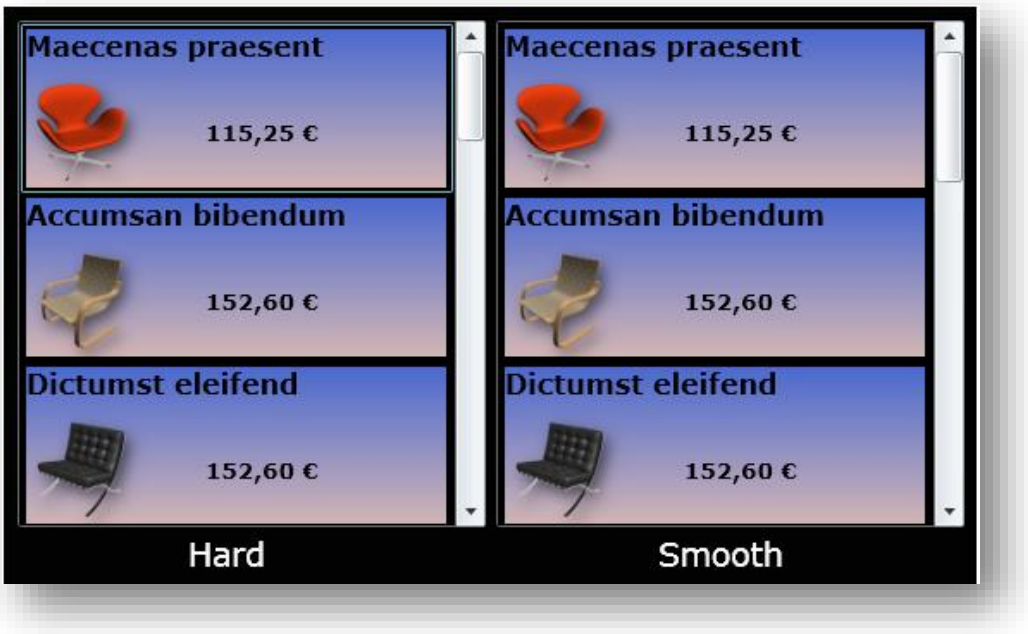


Figure 101 - capture - listes "hard" et "smooth" position de départ

Si on bouge les ascenseurs :

Figure 102 - capture - déplacement "smooth" de la liste de droite

Smooth Scrolling vs Rough Scrolling

Ce coup-ci vous avez vu ? La ListBox de gauche déplace ses items par "bloc". Lorsqu'on utilise une `ListBox` dans un contexte classique avec une seule ligne de texte pas trop grosse notre œil supporte la "saccade". Mais dès qu'on template un peu et que les items sont assez gros comme ici l'effet visuel n'est vraiment pas terrible.

La Listbox de droite en revanche scroll en douceur. On que le premier item commence à disparaitre partiellement (alors qu'à gauche on a déjà fait le saut vers l'item suivant).

To be optimized or not to be ?

A cela une bonne raison : la `ListBox` est optimisée et utilise par défaut un conteneur spécial, un `VirtualizingStackPanel`. Ce type de `StackPanel` très particulier virtualise l'affichage des items, il ne prend en compte que ceux qui sont affichés et se règle uniquement sur ceux-là. Il ne peut pas connaitre à l'avance la taille des prochains items à venir (qu'ils viennent du haut ou du bas) : puisqu'ils ne sont pas affichés, il ne les connait pas. Grâce à cette optimisation la `ListBox` peut présenter des centaines, des milliers d'items sans jamais consommer plus de mémoire que ne le réclame le nombre d'items visibles.

Hélas, toute optimisation a un coût ! Ici il est visuel.

La souplesse de Xaml, l'intelligence de Blend et la ruse du développeur

Mais Xaml est tellement souple qu'il suffit simplement de modifier la nature du conteneur de la `ListBox` pour améliorer les choses.

La ListBox "hard", celle de gauche dans notre exemple plus haut, a été conçue "tel quel", rien n'a été modifié en dehors de l'`ItemTemplate` qui met en forme les trois propriétés de chaque enregistrement (une base de données, une source XML… ici j'ai utilisé la génération de données aléatoires intégrée à Blend plus quelques modifications des nombres ainsi que l'ajout d'un convertisseur pour obtenir un format monétaire).

Pour créer la ListBox “Smooth”, celle de droite donc, j’ai d’abord effectué un simple copier / coller. J’ai ainsi récupéré tout le visuel, l’`ItemTemplate`, le binding à la source de données, etc.

C’est maintenant que les choses deviennent super difficiles, soyez attentifs ça va aller très vite : Clic droit sur la `ListBox`, puis “modifier des modèles associés” / “Modifier disposition des éléments (`ItemsPanel`)” / “Créer un élément vide”. Ce qui ouvre le dialogue suivant :

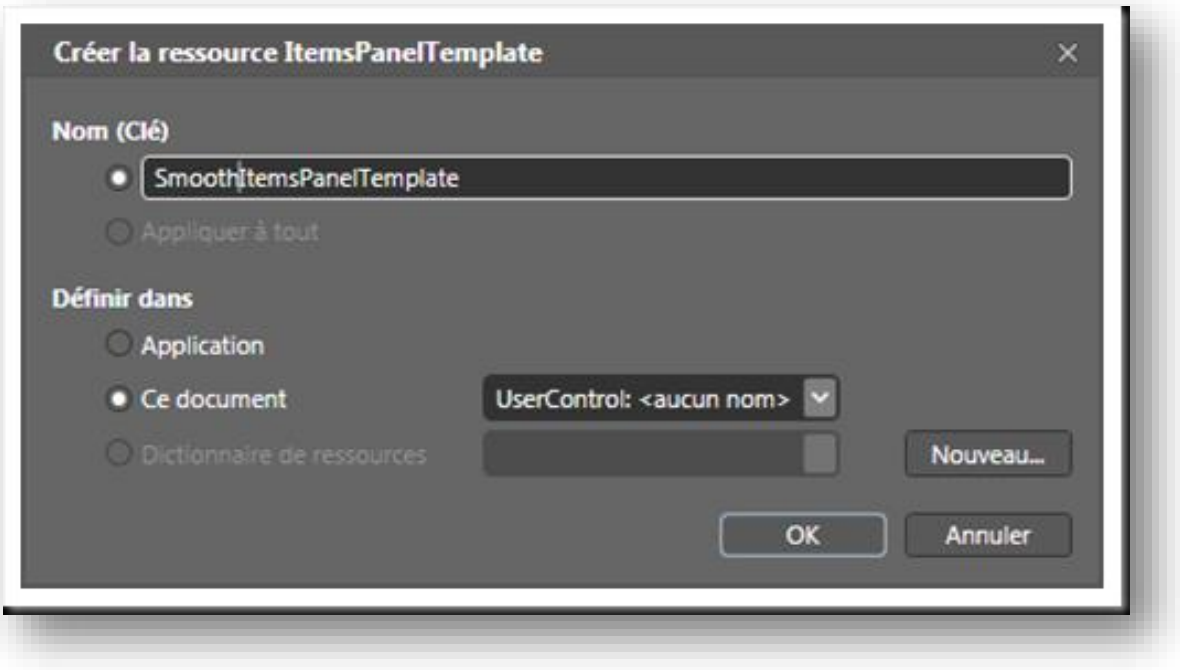


Figure 103 - Créer un ItemsPanelTemplate

Automatiquement l’affichage de Blend bascule en mode de modification de template, et regardez bien la capture ci-dessous :

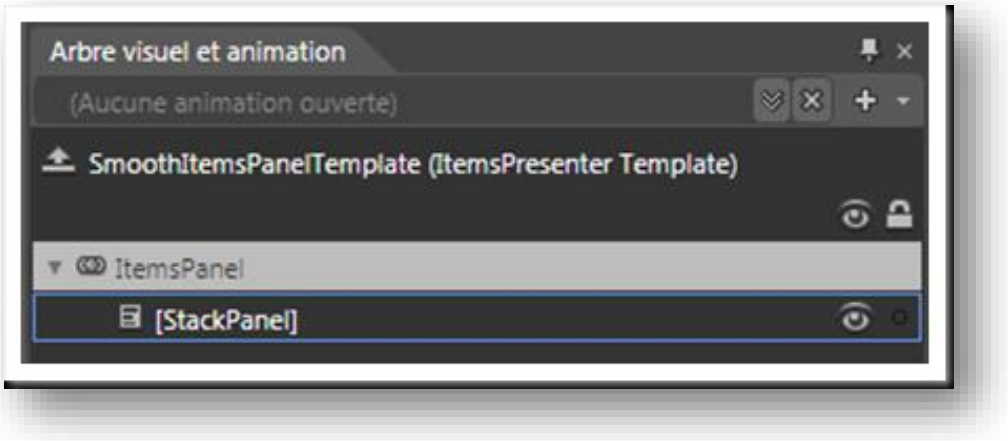


Figure 104 - Mode modification de Template sous Blend

`ItemsPanel` est déjà constitué d'un `StackPanel`, nous n'avons rien à faire puisque c'est justement ce que nous voulions : remplacer le `StackPanel` virtualisé par un `StackPanel` de base. Fini. Ca va très vite, je vous avais prévenu (vous pouvez relire plus lentement pour voir tous les détails de l'action) !

Pourquoi ce changement automatique ? Blend lit-il dans mes pensées ?

Blend est très fort, mais pas à ce point-là… La raison est autre : le `StackPanel` virtualisé n'est pas un composant disponible. Comme nous entrons en modification du template et que Blend ne peut pas nous afficher un composant auquel nous n'avons normalement pas accès, il le remplace par le plus proche, ici un `StackPanel` "normal". C'est tout de même assez malin de sa part.

Du coup, rien de plus à faire, on sort du mode templating et F5 pour lancer l'application et au lieu d'une `ListBox` hard, nous avons maintenant une superbe `ListBox` ultra smooth…

La prochaine étape serait, par exemple pour application tactile Windows Phone 7, d'ajouter de l'inertie pour que les mouvements ressemblent plus à une "glissade". Le premier pas est fait puisque nous disposons d'un `StackPanel` contenant en réalité tous les items à afficher. Mais nous verrons cela une autre fois.

Un monde presque parfait

En effet, en supprimant l'optimisation de la `ListBox` (son `StackPanel` virtualisé) nous avons beaucoup gagné côté visuel, mais nous avons perdu tout autant en termes de gestion mémoire. Désormais, si nous affichons 200 items dans la ListBox il faut savoir qu'un `StackPanel` classique sera créé ainsi que les 200 items à l'intérieur. L'affichage

n'est plus qu'une sorte de clipping. Si le `DataTemplate` ou l'`ItemTemplate` est un peu gros (ou utilise des effets comme le DropShadow que j'ai placé sur l'image, image elle-même consommatrice de mémoire) on risque fort d'avoir de désagréables surprises niveau consommation mémoire !

L'astuce qui rend la `ListBox` "smooth" est donc à réserver à des cas parfaitement identifiés où l'on sait par avance et de façon sûre que le nombre d'items à afficher sera toujours assez limité ou que le template sera assez léger.

Mais rappelez-vous qu'une liste ou une grille proposant plus d'une centaine d'items est souvent la preuve d'une analyse bancale ou d'une réalisation paresseuse, car aucun utilisateur au monde ne lit plus de deux pages. Ou bien il le fait parce qu'il n'a pas le choix mais il maudit le développeur sur 1000 générations !

Comme je ne vous souhaite pas un tel destin funeste, surtout pour votre innocente descendance (encore vous, vous l'avez bien cherché !), pensez à mettre la pédale douce sur les items que vous chargez dans les listes et les grilles qui ne sont plus virtualisées ! C'est une erreur que je vois tellement souvent que ça devient une fixette. Je vous le redirais donc certainement dans un autre billet, histoire que cela rentre bien dans toutes les têtes, même les plus … hard !

Un peu d'aléatoire dans les mouvements

Lorsqu'on créée des animations il faut savoir ajouter une pincée d'aléatoire pour rendre les mouvements plus naturels, plus réalistes, plus organiques.

Seulement voilà, l'esprit humain est encore moins doué pour inventer des séries aléatoires que ne le sont les ordinateurs...

Il faut donc s'aider. Pour cela on trouve sur le Web quelques sites intéressants.

La tremblante du bouton

Du "bouton", pas "mouton" hein... Rien à voir avec Creutzfeldt-Jakob ou les mad cows. Nul besoin d'infecter une pauvre bête innocente avec un prion malformé, nous voulons juste faire trembloter un bouton. L'effet peut être amusant.

C'est un exemple typique où il faut créer l'illusion d'un mouvement désordonné, donc aléatoire.

Le principe de base consiste à créer une time line sur un état, par exemple le `MouseOver`, de placer des `keyframes` espacées régulièrement et d'ajouter une translation sur `X` et `Y` sur l'ensemble de l'objet à chaque `keyframe`.

Pour un mouvement rapide suffisamment désordonné il faut entre 5 et 8 `keyframes`. Le problème n'est pas tant ici comment on fabrique cette animation sous Blend ou directement en Xaml mais comment on règle les translations pour que l'effet soit visuellement attractif.

Si vous essayez d'inventer six séries de couples de translation, je peux vous assurer que votre bouton aura un mouvement qui ne sera pas aléatoire, et ça se verra. Tentez le coup. Bien entendu, avec un peu d'esprit mathématique on peut y arriver. Mais pourquoi s'embêter ?

Utiliser un générateur de nombre aléatoire

Plutôt que de tenter de rendre votre pensée purement aléatoire, ce qui n'est pas possible si vous êtes sain d'esprit, il est bien plus efficace d'utiliser un générateur de nombres aléatoires.

Pour le bouton qui tremble il nous faut environ 6 couples d'entiers relatifs dont la valeur varie entre disons `-4` et `+4` (ici tout dépend de la taille du bouton et de l'intensité du tremblement qu'on désire obtenir).

En se rendant par exemple ici : http://www.random.org/integer-sets/ nous pouvons générer des ensembles d'entiers. Demander 6 couples de 2 entiers compris dans les valeurs fixées est un jeu d'enfant.

Ne reste plus qu'à copier ses valeurs dans la transformation (translation) des 6 `keyframes` et l'affaire est jouée...

On obtient ainsi rapidement et du premier coup un mouvement "naturel" en ce sens qu'il semble effectivement aléatoire, ce que le cerveau perçoit tout à fait. Il peut arriver qu'on soit obligé de générer une ou deux séquences différentes de plus et essayer les valeurs. L'aléatoire sur si peu de valeurs peut parfois être surprenant de régularité...

Pour rendre l'effet plus agréable la timeline est en autoreverse et en durée infinie (si on suppose une animation sur le `MouseOver` cela est indispensable pour que l'effet dure tout le temps où la souris est au-dessus du contrôle, peu importe la durée exacte de ce "stationnement" de la souris).

D'autres utilisations

Faire trembler un bouton, un hyperlien ou tout autre objet est une utilisation possible. Il existe bien entendu une infinité de situations où les nombres aléatoires peuvent rendre des services inestimables. Je parle ici de leur utilisation dans des animations, on peut bien sur utiliser la technique dans du code pur et dur.

Par exemple initialiser la position d'un objet, créer une légère variation dans une couleur, dans un déplacement, etc.

Quand on utilise des nombres aléatoires par code, on peut utiliser la classe `Random` (bien qu'il ne s'agisse que d'un générateur pseudo-aléatoire). Mais quand il s'agit de créer des timelines le mieux est donc d'utiliser des générateurs existants.

D'autres liens

J'ai évoqué http://www.random.org/ plus haut. Ce site est très complet et propose de nombreux générateurs et outils comme un randomiseur de liste, un générateur de bitmaps, etc. Le site offre un véritable générateur aléatoire basé sur le bruit atmosphérique. Les séries proposées sont réellement imprédictibles.

Le site http://stattrek.com/ propose une autre approche et d'autres outils. Sa vocation est plutôt de vous apprendre les statistiques. C'est assez bien fait et permet de recadrer certaines notions. Il y a bien entendu des outils comme un générateur de nombres aléatoires, un outil de calcul des permutations et combinaisons (très pratique quand on essaye d'estimer les combinaisons possibles dans un automate par exemple), et plein d'autres choses. En engliche bien entendu.

Lorsqu'il s'agit d'utiliser des valeurs stockées d'avance on peut aussi utiliser les digits du nombre Pi : http://www.piday.org/million.php. Ce site propose 1 million de décimales. Pi n'est pas aléatoire en lui-même, il est le résultat exact d'une opération connue (je vous renvoie à vos cours de trigonométrie...), mais jusqu'à aujourd'hui il n'a pas été possible de détecter des cycles ou des patterns dans ses décimales qui elles, forment donc une série aléatoire. Dans le même esprit on peut trouver sur http://apod.nasa.gov/htmltest/gifcity/e.2mil 2 millions de décimales du nombre "e" (base des logarithmes naturels).

Ensuite on trouve plein de choses sur Internet, il suffit de chercher avec Bing ou Google, je ne vais pas vous gaver de liens que vous pouvez trouver vous-mêmes.

Conclusion

“Pensez-y !” me semble une bonne conclusion. C’est en tout cas ce que veut réellement dire ce billet : penser à utiliser des générateurs aléatoires. Ils peuvent ajouter une note de réalisme à vos programmes, les rendre plus organiques, plus agréables. Tout est affaire de créativité bien entendu, mais encore faut-il penser à l’existence de ces outils. N’hésitez pas non plus à utiliser la classe `Random` pour générer par code des comportements plus « organiques ».

Lequel est le plus foncé : Gray ou DarkGray ?

Histoire de se détendre un peu, voici une question intéressante car la réponse n'est pas forcément celle que vous pensez !

La taille du cul des vaches

Rappelez-vous, il y a de de cela longtemps circulait sur Internet une histoire (vraie ou fausse peu importe) qui démontrait que le diamètre des fusées de la Nasa dépendait, par une suite invraisemblable de relations de cause à effet, de la taille du cul des vaches au temps des romains. On passait par la taille des charriots dont la largeur dépendait justement de celle du fessier de ces honorables bovins, ce qui imposait une distance entre les roues, elles-mêmes créant des traces au sol qui firent que les premières voitures à cheval, pour emprunter les routes aux ornières profondes tracées par les charrues se devaient de respecter la même distance inter-roue, etc. On en arrivait ainsi au diamètre des fusées qui, pour passer sur les trains, qui devaient passer dans les tunnels, dont la taille dépendait etc, etc... Au final un objet ultra technologique et on ne plus moderne se retrouvait à respecter une taille qui dérivait de celle du cul des vaches des romains...

Bonus pour cette version revisitée, une analyse de cette histoire ici : http://www.straightdope.com/columns/read/2538/was-standard-railroad-gauge-48-determined-by-roman-chariot-ruts

Une histoire plus vraie qu'on le pense

Bien entendu je n'ai jamais pu savoir à l'époque s'il s'agissait d'un hoax, d'une vérité romancée ou bien d'une vérité historique. L’analyse proposée dans le lien ci-dessus confirme à mi-mots que cette histoire est probable mais pas forcément vraie sur

toute la ligne. Mais l'histoire que je vais vous raconter aujourd'hui me fait penser, bien des années plus tard, que cette histoire était probablement vraie...

Je tire librement inspiration d'un billet publié en 2006 par Tim Sneath, ceux qui désirent lire ce billet original (en anglais) n'ont qu'à cliquer ici.

Les couleurs sous XAML

24 bit RGB

En Xaml il existe plusieurs façons d'exprimer une couleur. Une des options est d'utiliser la notation 24 bits hexadécimale de type RGB (Red/Green/Blue, rouge/vert/bleu) :

```
<Rectangle Fill="#B2AAFF" Stroke="#10F2DA" ... />
```

XAML 89 - Définition d'une couleur 24 bit RGB

32 bit ARGB

Comme XAML sait gérer la transparence via le canal dit Alpha, on peut spécifier une couleur par un code hexadécimal 32 bits dit A-RGB (Alpha / RGB) :

```
<Line Stroke="#7F006666" ... />
```

XAML 90 - Définition d'une couleur 32 bit ARGB

Ce qui donnera une ligne en Cyan à 50% d'opacité.

scRGB

Plus subtile et bien moins connu (et encore moins utilisée) est la notation scRGB qui permet d'exprimer une couleur sous la forme de 4 nombres décimaux de type Single ce qui autorise la représentation d'un gamut ultra large. A quoi cela peut-il servir d'utiliser un système de notation des couleurs qui dépasse de loin les limites du RGB qui va de #000000 à #FFFFFF ? Créer un noir plus noir que le noir ou un blanc plus blanc que blanc, à part si on est-on un publiciste en mal d'argument pour une nouvelle lessive, cela semble idiot.

Et pourtant cela ne l'est pas. Il n'y a qu'en matière de lessive que "plus blanc que blanc" est une ânerie (dont Coluche se délecta dans un sketch célèbre). Lorsqu'on parle infographie cela peut avoir un sens très concret : conserver les couleurs originales si celles-ci doivent subir de nombreux traitements, comme par exemple une correction du contraste ou de la Hue (teinte dans le système HSL). En effet, à force de calcul, d'arrondis et d'approximations, votre filtre de correction peut finir par

créer des à-plats horribles. En utilisant le système scRGB, le code (qu'il soit C# ou XAML) pourra conserver le maximum d'information sur chaque composante couleur.

Un exemple de notation scRGB :

```
<Rectangle Stroke="sc#1, 0.6046357, 0.223508522, 0.182969958"
      Fill="sc#1, 0.7785599, 1, 0"
     RadiusX="25" RadiusY="25" Width="250" Height="80"
      StrokeThickness="5" Margin="60" />
```

XAML 91 - Notation d'une couleur en scRGB

(Cela donne un rectangle jaune à bords arrondis bistre).

Les couleurs nommées

Enfin, il est possible d'utiliser des noms pour définir des couleurs.

WPF divergent sur ce point avec d'autres moutures de XAML comme Silverlight car économie de code oblige pour le Framework réduit de Silverlight, ce dernier ne contient pas toutes les définitions de couleur de son aîné WPF. Mais le principe reste rigoureusement le même (je vous joins d'ailleurs en fin d'article un code qui définit toutes les couleurs WPF utilisables sous Silverlight ou Windows Phone).

On peut ainsi utiliser des noms tels que : `Green`, `SteelBlue` ou `MediumVioletRed`. Ce qui donne en XAML :

```
<Rectangle Stroke="yellow" Fill="red" Width ="50" Height="50" />
```

XAML 92 - couleurs nommées

Et le cul des vaches ?

C'est là que l'affaire devient croustillante... Attendez la suite pour juger !

Par souci de compatibilité avec HTML et SVG, Microsoft a repris la liste des couleurs définies dans ces standards. Une bonne idée, on a souvent accusé Microsoft de ne pas respecter les standards, ce que j'ai toujours trouvé idiot puisque justement l'innovation vient de ce qui est différent et non de l'uniformisation. Et bien justement, quand les développeurs de chez Microsoft se plient à cette servile obligation cela donne des choses bien curieuses (à l'insu de leur plein gré comme disait l'idiot pédaleur).

En effet, la liste des couleurs HTML est pleine de bizarreries et d'idiosyncrasies qu'il eut été préférable de corriger. Expurgée de ces anomalies la liste des couleurs HTML

aurait pu faire une belle liste pour XAML, mais voilà, compatibilité oblige (en fait rien ne l'obligeait, sauf les habitudes), on se retrouve dans l'un des environnements de développement le plus moderne à trainer des bêtises d'un autre âge ! La fameuse taille du cul des vaches au temps des romains influençant le diamètre des fusées de la Nasa...

Par exemple, le spectre couvert par les couleurs HTML ne respecte pas même une répartition à peu près homogène, ce qui fait qu'on dispose de très nombreuses teintes dans les rouges ou les oranges alors que les verts sont très mal couverts.

Autre exemple, les couleurs ont parfois des noms ésotériques qui montrent à quel point les auteurs originaux n'avaient aucune volonté de rendre l'ensemble compréhensible. En dehors d'un américain pure souche, qui peut en Italie, en France ou en Slovénie s'imaginer de quel bleu il s'agit lorsque HTML nous indique un "DodgerBlue" ? La charte couleur des Tshirts d'une équipe d'un sport totalement inconnu (le baseball) chez nous n'évoque rien (les dodgers sont en effet une équipe de baseball de Los Angeles, très connus certes, mais uniquement des américains et des rares amateurs étrangers).

Des origines encore plus lointaines

Mais si cela s'arrêtait là le rapprochement avec la petite histoire sur l'arrière train des vaches pourrait sembler un peu capilotractée. En fait nous sommes exactement dans le même cas. Car les choses ne s'arrêtent pas à HTML. Ces couleurs remontent en réalité aux premières implémentations sous UNIX du système X-Window ! HTML définit 16 couleurs qui sont directement mappées sur les 16 couleurs de la palette EGA. Mais plus loin encore, les premiers browsers comme Mosaic (1992) supportaient aussi les couleurs nommées de X11 ! Malheureusement certaines couleurs HTML avaient des homonymes X11 qui, bien entendu, ne donnait pas exactement la même teinte... par exemple ce vert HTML donnait ce vert sous X11.

Un gris plus foncé que le gris foncé

Et c'est ainsi que de tout ce mélange le `Gray` HTML fut défini par `#808080` alors que le `DarkGray` est défini par `#A9A9A9`, un gris plus clair que le gris...

On en revient à la question posée dans le titre de ce billet. Et vous voyez que la réponse est loin d'être celle qui semble s'imposer en toute logique !

XAML réutilisent cette liste de couleurs qui ne remonte pas aux temps des romains, mais pour l'informatique, l'époque de X11 c'est même pire : de la préhistoire !

Du coup, il semble bien plus intelligent d'utiliser les codes couleurs RGB, ARGB ou scRGB que les couleurs nommées si on ne veut pas obtenir des résultats étranges...

Incroyable non ?

On pourrait tirer milles conclusions de cette petite histoire. Je vous laisse y réfléchir.

On peut aussi juste en rire, c'est bon pour la santé ☺

(pour ceux qui ont lu jusque-là, le cadeau annoncé : ColorHelper.cs (16,31 kb))

Quels outils et quelle chaîne graphique XAML ?

WPF, Silverlight, WinRT, Windows Phone, 2D, 3D, le monde des nouvelles interfaces réclame désormais de fortes compétences en infographie. S'il semble évident que la présence d'un graphiste soit devenue indispensable en phase de conception des interfaces, savoir quels outils utiliser est en soi déjà un parcours du combattant !

Pour e-naxos j'ai formé une infographiste afin d'offrir un service de design pour les applications XAML. Avant de former, il faut savoir de quoi on parle, c'est toute la difficulté pour un bon formateur : avoir la connaissance "livresque" mais surtout la connaissance pratique !

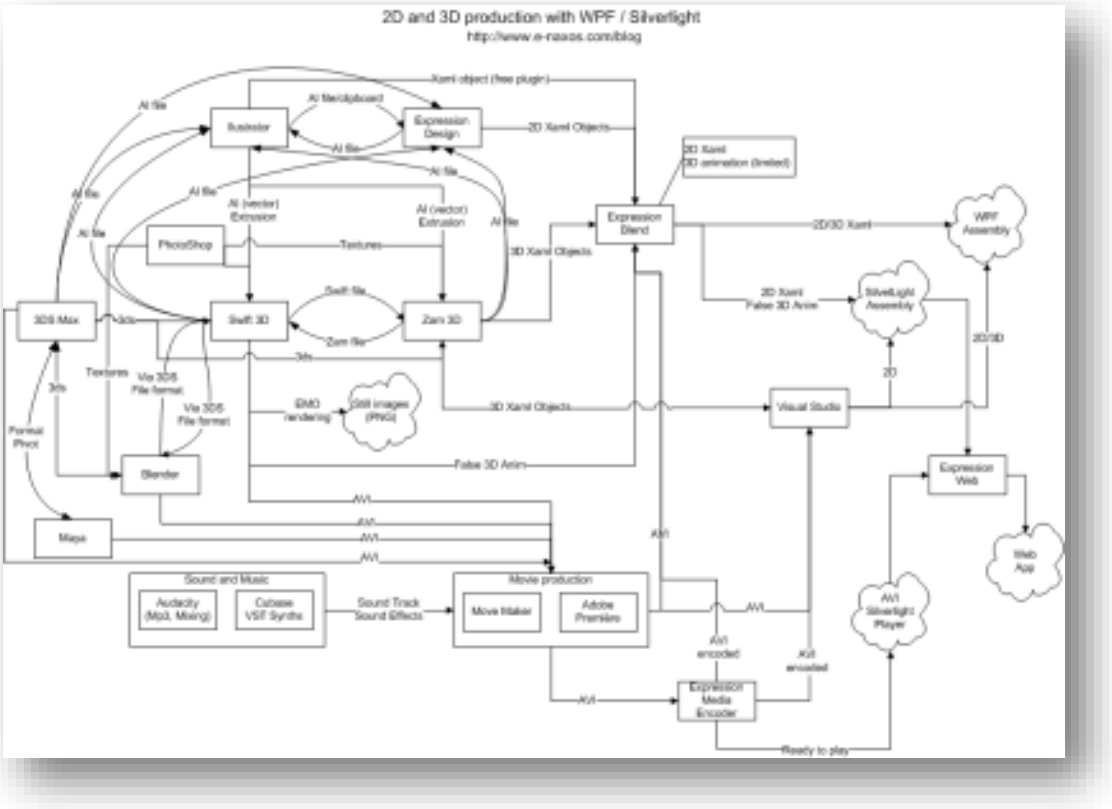
Cela fait donc de longs mois que je m'entraîne. Je n'ai pas forcément beaucoup évolué niveau dessin, je ne serai jamais Dali ni Da Vinci, je m'y suis fait depuis mes premiers essais en maternelle, et même si les courbes de Bézier et autres splines n'ont plus beaucoup de secret, l'investigation du monde 3D n'a pas été une promenade de santé !

Bref, pour produire une application "graphisée" c'est beaucoup plus simple à dire qu'à faire, je peux vous l'assurer ! Gérer un ou plusieurs graphistes, leur faire comprendre ce qu'est une application, la différence entre une `Listbox` et un `StackPanel` n'est pas forcément non plus la chose la plus simple. Et pourtant, c'est essentiel. Depuis des années j'engrange donc une expérience à la fois nouvelle et excitante, car ce qui se profile est l'avènement de nouveaux logiciels conçus autrement. Et j'aime le changement.

Je ne vais pas ici vous détailler l'ensemble du processus de création d'une application XAML, mais vous proposer un petit tour dans les outils qui peuvent (et doivent) être utilisés pour produire une logiciel « designé ».

Tour d'horizon des outils

Un dessin valant mieux qu'un long discours, regardons d'abord ce "petit" diagramme que j'ai fait rapidement sur la base de mon expérience :



XAML 93 - Schéma de la chaîne graphique XAML

Pour voir l'image en 100% téléchargez-la : 2D3DProduction.png (43,20 kb)

J'ai testé tous les chemins dessinés et tous les softs présentés et même bien plus... Ce qui reste sur ce diagramme sont les principaux chemins que peut prendre un objet graphique et les principaux softs qu'il faut connaître et maîtriser pour produire une solution professionnelle. Certaines personnes se limiteront à des portions du diagramme, ce qui reste possible, mais seule la connaissance de la totalité des logiciels et chemins de celui-ci permet de "tout" faire.

On notera que depuis que j'ai écrit ce billet des petites choses ont changé mais finalement très peu. La disparition de la suite Expression laisse tout de même encore la version Pro de Encoder ou de Blend qui est fourni avec Visual Studio. De même la disparition de Expression Design n'est pas totale et la version 4 est toujours téléchargeable gratuitement. D'autres logiciels sont devenus plus matures entre temps aussi. Par exemple InkScape est un excellent concurrent de Illustrator, en Open Source et donc gratuit. Les dernières versions savent même exporter du XAML !

Le diagramme mériterait donc quelques ajustements pour refléter ces changements, mais cela ne remet pas en cause la vision qu'il propose des « chemins » de production graphique avec XAML.

Partons de la fin

A l'arrivée nous désirons obtenir une application XAML (WPF, WinRT…). Sous XAML nous avons besoin de composants templatés, d'image PNG ou Jpeg et d'objets visuels en 2D ou 3D. La 3D XAML n'existe que sous WPF, pour les autres profils de XAML la 3D se programme autrement (ce qui est dommage).

Chaque élément graphique impose certains chemins incontournables:

Par exemple, Silverlight gère très bien les objets 2D créé par Expression Design, mais les effets d'Expression Design ne sont pas gérés, comme le Drop Shadow. Moralité, un objet 2D créé en XAML avec Design ne peut être utilisé tel quel sous Silverlight, il faudra le transformer en PNG. Sous WPF il est possible d'ajouter des effets comme le Drop Shadow et il sera donc possible d'exploiter directement l'objet 2D Xaml. Objet XAML ou PNG on voit tout de suite que les chemins de production diffèrent légèrement et que les possibilités pour le développeur seront très différentes…

Pour produire des objets 3D il y a un goulot d'étranglement : aucun outil Microsoft ne produit d'objets 3D. Le seul logiciel qui permet de faire cela est Zam 3D d'Electric Rain. Il est plutôt agréable à utiliser et pas très cher. Il n'a pas évolué depuis sa version 1.0 mais rend des services appréciables. Seulement voilà, parfois vous avez envie d'utiliser un objet 3D un peu compliqué ou mis en scène. Et chez Electric Rain ils sont malins, ou un peu mesquin au choix, et Zam 3D ne possède pas de moteur de rendu ! Il est juste capable de sortir du code Xaml. Et si vous voulez créer un PNG d'un objet 3D ? Pas de solution sauf à acquérir en plus Swift 3D du même éditeur qui dispose d'un moteur de rendu (EMO) qui ne peut pas concurrencer les gros moteurs de ray tracing comme on en trouve sur 3DS Max ou Blender, mais on peut produire

des images de belle qualité malgré tout. Ce produit existe aujourd'hui en version 6 et est plutôt agréable à utiliser. On peut s'amuser à faire de la 3D sans entrer six mois en formation pour manipuler les monstres comme Maya ou Blender.

On le voit le chemin se complique... J'ai un objet 3D, pour une application WPF je vais le réutiliser directement, pour une application Silverlight je vais le "prendre en photo" en faisant un PNG, du coup j'ai besoin de Xam 3D pour créer l'objet 3D Xaml et de Swift 3D pour réimporter mon projet Xam 3D et utiliser le moteur de rendu pour créer le PNG ! Vous allez me dire autant acheter tout de suite Swift 3D ! Et bien non... Ce dernier a bien un mode d'export en Xaml mais c'est plutôt pour les animations comme en Flash, par exemple une animation de 20 images créera 20 fois l'objet Xaml sur 20 couches différentes dont l'opacité est à 0% et il y aura 20 timelimes générées chacune s'activant à la suite de la précédente pour passer l'opacité de l'image numéro à 100%. C'est un peu le principe d'animation des petits livres qu'on doit feuilleter à toute vitesse pour avoir l'impression d'un dessin animé.

Bien entendu en bout de course il faut avoir Blend, c'est obligatoire. Et puis ensuite viennent Visual Studio ou Expression Web.

Produire des modèles 2D avec Expression Design c'est sympa, le soft est agréable. Mais il n'a pas le niveau d'un Illustrator ou d'InkScape. Si on veut créer une image 2D un peu "léchée" il faut donc posséder l'un ou l'autre de ces logiciels et savoir s'en servir. On exporte en XAML directement grâce à un petit plugin Illustrator gratuit 'ou directement depuis InkScape), ou bien on importe le fichier AI dans Expression Design (le copier-coller entre Illustrator et Expression Design fonctionne aussi et c'est très pratique).

Pour produire de la 3D c'est un peu pareil... Swift 3D ou Xaml 3D sont bien sympathiques, mais le mesh editor est un peu limité, et dès qu'on entre dans l'animation on voit bien que ces logiciels sont parfaits pour produire des banners de pub à la Flash mais pas pour créer des séquences animées à la Maya !

Bref, pour de beaux objets 3D, le mieux est d'utiliser Blender (gratuit mais une vraie usine à gaz) ou 3DS Max de AutoDesk, le meilleur que je connaisse et le plus agréable à utiliser. Un peu cher c'est tout. Surtout si on ajoute cela au budget de la quinzaine de logiciels présents sur mon diagramme !

Avec 3DS Max vous allez créer des objets 3D que vous sauvegarderez en format 3DS, pas au format Max ni Dxf ! Et oui... Swift 3D et Xaml 3D ne savent importer que le format 3DS... Au passage vous perdrez certains éléments ou certaines textures. Rien

ne sert donc de trop compliquer. Il faut finir le travail sous Xaml 3D pour être sûr que ce qui est fait est supporté par l'export 3D en Xaml.

Si vous voulez faire des animations, cela se corse ! On a vu que les animations Xaml produites par Swift 3D ne sont pas du Xaml 3D mais des séries de plans 2D en couche, en revanche Swift 3D sait produire des AVI, comme 3DS Max. Un petit AVI peut souvent être bien plus joli qu'une animation d'objets 3D faites à la main dans Expression Blend. Dans certain cas c'est donc une alternative intéressante.

Reste que l'AVI devra être encodé correctement. C'est là qu'intervient Expression Media Encoder. Il peut même produire une page Web avec lecteur Silverlight intégré. Cela est intéressant pour une longue séquence présentée comme un petit film.

Bien entendu qui dit petit film dit montage... Le montage peut être réalisé avec Windows Movie Maker, simple et gratuit. Mais simple... Il faudra certainement passer à Adobe Première pour produire un vrai petit film bien monté.

Qui dit film dit son. Vous pouvez facilement prendre un MP3 et le coller dans le projet Movie Maker. Mais, droits de diffusion mis à part, cela va rester très sommaire.

Pour produire une bande son synchronisée avec les images je travaille personnellement avec Cubase ou Ableton Live. Les synthés et la musique c'est plus ma spécialité que le dessin c'est donc la partie que je préfère ! Créer l'image puis créer le son pour cette dernière est une expérience vraiment intéressante. Si vous êtes musicien vous vous éclaterez dans cette phase ! Sinon, pensez à engager un compositeur (tarifs sur demande !).

Etc !

Je vais m'arrêter là, car il est impossible de décrire ici tous les chemins du diagramme, toutes les justifications de prendre tel ou tel chemin pour tel ou tel résultat. Et puis c'est aussi la valeur ajoutée que j'ai créée pour e-naxos, j'aime partager, la preuve ce billet, mais je ne vais pas vous livrer des années d'expérimentation et de savoir-faire accumulé comme ça non plus... Et puis, sans mesquinerie aucune, aucun projet n'est identique et chacun nécessite un conseil personnalisé et approprié. C'est à cela que servent les gens comme moi, laissez-moi mon utilité ☺

J'espère en tout cas que ce petit tour d'horizon et le diagramme vous aideront à vous faire une idée du cycle de production d'interfaces professionnelles "graphisées". Si vous êtes développeur cela vous aidera à envisager votre collaboration avec des

infographistes, et si vous êtes clients cela vous aidera à mieux comprendre que le prix qui vous sera demandé est le reflet d'un vrai investissement et d'un vrai travail.

Dans tous les cas, bon voyage au pays des logiciels de la prochaine génération !

Sauts de ligne dans un "Content"

Une astuce très courte : Comment insérer un saut de ligne lorsqu'on fixe la valeur d'une propriété `Content` en Xaml (étant entendu qu'on désire placer un texte et non d'autres objets) ?

Des propriétés `Content` il y a beaucoup sous XAML. Un bouton, un `CheckBox`, sont autant d'exemples de classes offrant une propriété "`Content`", généralement de type `ContentPresenter` (ou dérivé). L'avantage de ces champs est de pouvoir recevoir n'importe quel contenu. Par défaut on peut y placer un simple texte, sans même avoir à créer un objet `TextBlock`.

L'utilisation de chaines pour le contenu des classes indiquées est très fréquente. C'est pour cela que tout le comportement a d'ailleurs été optimisé pour rendre simple cette utilisation (rien à faire d'autre que de taper le texte) sans perdre la possibilité de placer un contenu complexe (une image, un contrôle, toute une arborescence visuelle…).

Mais tout aussi fréquemment on aimerait bien pouvoir insérer un saut de ligne dans le texte tapé. Ce que l'éditeur de Blend ou de VS ne supportent pas (le `Return` valide la saisie). Quant à tenter un "`\n`" dans la chaine, cela ne marche pas, pas plus qu'une balise `<LineBreak/>` car nous ne sommes pas dans le `Run` d'un `TextBlock`.

Il existe pourtant une solution simple. Le "`\n`" n'est pas si loin… encore faut-il saisir *directement le caractère spécial en hexadécimal et l'insérer comme tout caractère de ce type en respectant la syntaxe XML*.

Ainsi, pour couper en deux lignes le `Content` d'un `CheckBox` (son libellé donc) on écrira :

```
<CheckBox Content="Accès à tous les flux&#xa;et tous les clients"
Margin... />
```

XAML 94 - Insertion d'un saut de ligne dans une propriété Content

Ce qui donnera un affichage de ce type :

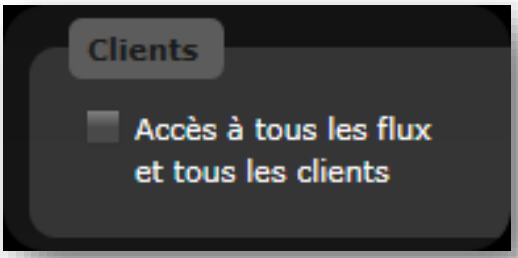


Figure 105 - capture -Insertion d'un retour à la ligne dans le Content d'une CheckBox

Le secret réside donc dans le caractère `#xa;` c'est à dire le code hexadécimal `A`, soit le décimal `10`, donc le caractère `Line Feed ASCII`, signifiant *Saut de Ligne*... Vous remarquerez au passage que c'est exactement la signification de "`\n`" en C# (ASCII 10), le "`\r`" valant pour le retour charriot (ASCII 13) inutile dans le cas présent.

Conclusion

Ce n'est vraiment pas très savant comme astuce, mais je sais qu'elle rendra service. Profitez-en, c'est Dot.Blog, c'est sympa et c'est gratuit !

SVG to Xaml

SVG ? Mais si vous savez, ce vieux format dont personne n'a voulu pendant des lustres (ça date de 1999[9] le millénaire précédent !) et qui devient (veut devenir serait plus exact) un concurrent de XAML avec HTML5 ... Alors vous replacez ? Ok. Bon, donc ce fantôme revient nous hanter et il semble utile dans certains cas de pouvoir s'approprier sans vergogne certains dessins SVG mais en les mettant au format XAML...

Du coup il faudrait un petit utilitaire pour faire ça.

Et ça existe.

C'est un projet assez ancien (forcément, on parle de SVG) sur CodePlex et qui permet de convertir du SVG en XAML. Il y a même un utilitaire en ligne de commande qui produit en même temps le PNG équivalent. Très pratique donc.

[9] Souvent on me demande pourquoi je n'aime pas la mode HTML 5 ou même HTML/CSS/JS sous toutes ses versions. Vous avez ici l'explication. C'est un fatras de vieilleries accolées ensembles sans aucune cohérence. Des trucs qui isolés ne valaient rien, et qui ensemble ne donnent pas un truc nouveau mais un gros tas de vieilleries. Franchement ce n'est pas sérieux.

La librairie est à la base un *beautifier* de XAML, et elle se base sur une autre lib de CodePlex, SharpVectors appelé aussi “SVG#”, ça ne s’invente pas !

Le tout est ici : http://xamltune.codeplex.com/

Bon pillage[10] des icônes SVG pour vos applications XAML !

Dessiner des horloges, un peu de trigonométrie !

Tout le monde sait faire un cercle, enfin une Ellipse particulière sous XAML ayant une hauteur et une largeur identiques. En revanche placer les marques des minutes ou placer correctement le texte des heures dans le cadran n'est pas toujours évident.

Si on tente de le faire "à la main", et même sous Blend, c'est une véritable galère (imaginez 60 petites ellipses pour les secondes à répartir soigneusement, sans parler d'un éventuel changement de dimension qui mettrait tout par terre !).

Aussi génial soit Blend il y a des choses qui n’ont de sens qu’effectuées par code.

Dessiner une horloge est un bon exemple. Mais malgré la croyance populaire tous les informaticiens ne sont pas forcément des "bêtes" en math ! On peut être un expert en SQL sans se rappeler ses cours de trigonométrie et on peut développer toute une architecture multithreadée sans même savoir ce qu'est un Groupe de Lie. Mais parfois ce genre de connaissance peut manquer notamment dès qu'on aborde le traitement d'image ou de dessin.

[10] Pillage qui doit s’entendre pour des sources libres de droits bien entendu…

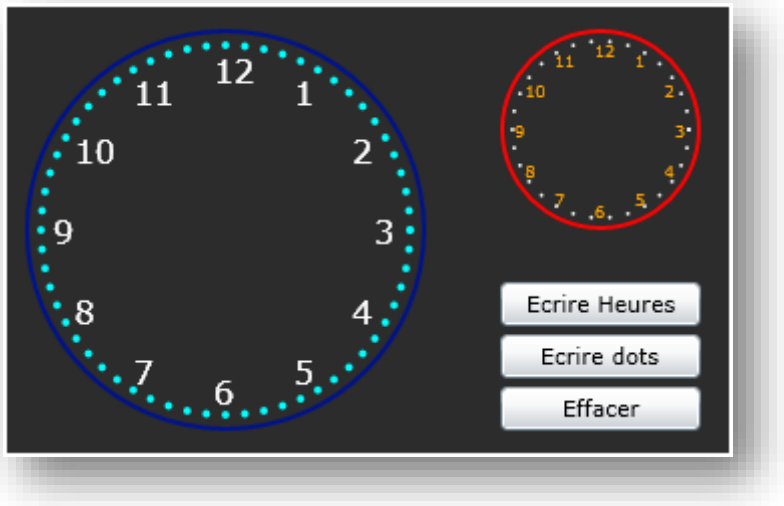


Figure 106 - Dessiner des horloges

Le but du jeu est donc ici de répartir harmonieusement des objets sur un cercle. Dans un cas il s'agit de texte (les heures) dans l'autre des cercles (les dots). Mais avant de regarder le code, rappelons quelques points mathématiques :

L'unité `Math` fonctionne en radians c'est à dire qu'un cercle vaut 2 Pi (donc deux fois 3.1415926...). C'est bête parce qu'en général on utilise plus souvent les degrés, ce qui est plus facile à se représenter mentalement quand on n'est pas un mathématicien de métier.

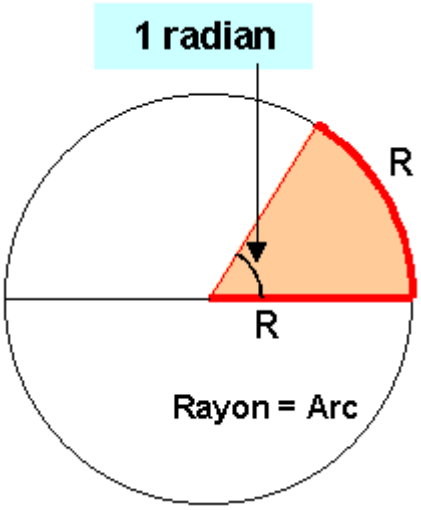


Figure 107 - Cercle trigonométrique et Radians

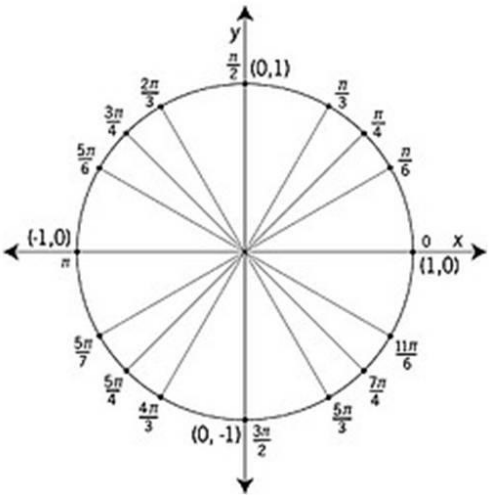


Figure 108 - Cercle trigonométrique

Comme on peut le remarquer en radian le 0 se trouve au milieu à droite du cercle et non pas en haut à midi comme on pourrait s'y attendre. Du coup, comme les fonctions mathématiques de .NET marchent en radians si on raisonne en degrés il faudra penser à cette petite spécificité lors de la conversion. Par exemple, la place de "*1 heure*" sur une horloge se situe à 300 degrés et non pas à 30° (360 divisé par 12).

Le code ci-dessous permet d'écrire les heures dans l'exemple live Silverlight dont la capture se trouve plus haut. En cliquant sur le titre du chapitre dans ce livre vous pouvez accéder à l'entrée de blog originale et jouer avec l'exemple.

```
private void writeHours(int radius, int offsetx, int offsety, Canvas parent,
double fontsize, Color color)
        {
            var h = 1;
            for (var angle=300;angle<=630;angle+=30)
            {
                var t = new TextBlock {
                        Foreground=new SolidColorBrush(color),
                        Text=h.ToString(),
                        FontSize = fontsize};
                h++;
                t.SetValue(Canvas.LeftProperty,
                   (radius * Math.Cos(DegreetoRadian(angle))) + offsetx);
                t.SetValue(Canvas.TopProperty,
                   (radius * Math.Sin(DegreetoRadian(angle))) + offsety);
                parent.Children.Add(t);
            }
        }
```

Code 52 - Placer un élément sur un cercle

Les paramètres sont les suivants :

`radius` : rayon du cercle virtuel sur lequel les heures sont posées

`offsetx` et `offsety` : positionnement du centre du cercle dans le canvas parent, sachant que le 0,0 se trouve en haut à gauche.

`parent` : le `Canvas` parent

`fontSize` : taille de la fonte

`color` : la couleur de la fonte

Pour dessiner les dots on utilise le même principe (sans s'encombrer des détails du point de départ décalé puisque toutes les dots sont identiques).

Le code n'est pas optimal et on peut certainement faire mieux, mais cela vous donne un point de départ ... Pour compléter le tout, vous pouvez télécharger le projet : ClockHours.zip (61,42 kb)

La multi-sélection dans les ListBox

Les ListBox supportent la multi-sélection, cela peut parfois être très utile.

Figure 109 - Gloubiboulga et MultiSélection

La propriété de la `ListBox` qui nous intéresse s'appelle sans malice : `SelectionMode` et elle peut prendre les valeurs suivantes : `Single`, `Multiple`, `Extended`. En mode `Single` on retrouve le comportement par défaut mono sélection. Les deux autres modes permettent d'accéder au comportement multi sélection. En mode `Multiple` la sélection s'opère par le clic sur un item, en enfonçant *Ctrl* ou *Shift*. Le mode *Extended* fait que le *Shift* permet de sélectionner des étendues.

Jouez avec notre ami Casimir dans l'exemple live sur Dot.Blog – cliquez sur le titre de ce chapitre pour accéder au billet original - et grâce à la ListBox Silverlight confectionnez votre propre Gloubiboulga en partant de la recette originale et de ses options (pour les plus gourmands !)

Styles Cascadés (BasedOn Styles)

Les styles cascadés amènent une simplification intéressante dans la création de feuilles de style XAML. On retrouve cette syntaxe dans tous les profils XAML normalement, de WPF à WinRT.

En soi rien de nouveau à l'ouest puisque c'est le principe même des feuilles de styles CSS (qui y puisent d'ailleurs leur nom). Mais le CSS s'applique à quelques éléments simples HTML alors que là nous parlons de styles Silverlight, c'est à dire d'objet complexes pouvant définir tout un visuel, animations comprises.

Figure 110 - Exemple de styles "based on"

L'image ci-dessus montre 4 boutons. Tous sont des boutons standards du framework.

Le premier, intitulé "`Base SL`" possède le style Silverlight par défaut

Le second, "`Normal`" est décoré par le style "`BoutonNormal`"

Le troisième "`Gros`" est décoré par le style "`BoutonGros`"

Et le quatrième "`Alarme`" est décoré par le style "`BoutonGrosAlarme`"

Visuellement c'est plutôt moche, je vous l'accorde, mais le but du jeu est de voir l'effet du cascading styling...

Le style "`BoutonNormal`" est défini comme suit :

```
<Style x:Key="BoutonNormal" TargetType="Button">
      <Setter Property="Width" Value="90" />
      <Setter Property="Height" Value="30" />
      <Setter Property="HorizontalAlignment" Value="Left" />
      <Setter Property="VerticalAlignment" Value="Bottom" />
      <Setter Property="BorderThickness" Value="2"/>
</Style>
```

XAML 95 - Définition d'un style pour la classe Button

Là où les choses deviennent plus intéressantes, c'est dans le style "`BoutonGros`" ci-dessous où l'on voit apparaître l'attribut `BasedOn` qui permet de fonder le style courant sur celui qu'on indique :

```
<Style x:Key="BoutonGros"
          BasedOn="{StaticResource BoutonNormal}"
          TargetType="Button">
  <Setter Property="Width" Value="180" />
  <Setter Property="Height" Value="60" />
  <Setter Property="FontFamily" Value="Comic Sans MS"/>
</Style>
```

XAML 96 - Définition d'un style "based on" pour la classe Button

Enfin, le dernier style se fonde lui-même sur le précédent par le même mécanisme, le niveau de cascading n'étant pas limité. On peut voir notamment que le changement de famille de fonte introduit dans le style "BoutonGros" s'est propagé au style "BoutonGrosAlarme" (la célèbre fonte Comic Sans qui fait hurler les Designers – quitte à faire moche j'assume !).

```
<Style x:Key="BoutonGrosAlarme"
          BasedOn="{StaticResource BoutonGros}"
          TargetType="Button">
  <Setter Property="Width" Value="160" />
  <Setter Property="Height" Value="40" />
  <Setter Property="FontSize" Value="18"/>
  <Setter Property="FontWeight" Value="Bold"/>
  <Setter Property="Foreground" Value="Red"/>
  <Setter Property="BorderThickness" Value="4"/>
  <Setter Property="BorderBrush" Value="#FFFF0202"/>
</Style>
```

XAML 97 - Autre définition de style utilisant "based on"

Voilà, c'est tout simple, mais cela peut radicalement simplifier la création de gros templates pour des applications. Tous les avantages du Cascading Style Sheet de HTML dont l'intérêt ne se démontre plus, mais appliqué à des objets et à la sophistication de Silverlight. Que du bonheur...

Un peu de design avec Blend

Du code, toujours du code... il n'y a pas que le code dans la vie, il y a aussi le design ! Avec Blend c'est plus facile (normal, il est fait pour ça). Voyons au travers d'un exemple quelques astuces de mise en œuvre graphique...

L'exemple

Pas de quoi se rouler par terre je vous l'accorde, mais c'est un simple prétexte. Il n'y a pas d'interaction, juste à regarder. Mais comme ça bouge n'hésitez pas à cliquer sur le titre de ce chapitre pour accéder à l'exemple live sur Dot.Blog…

Figure 111 - Un peu de design

Les trucs

C'est dans le planisphère qui tourne que se situent les astuces. Aucune n'est délirante je vous le dit tout de suite. Ce qui m'intéresse dans ce billet est de vous montrer comment les outils mis à disposition par Blend peuvent rendre la conception d'une interface plus aisée. Un autre but est de vous donner des exemples d'utilisation de ces outils pour libérer peut-être un peu plus encore votre imagination.

Le texte "e-naxos"

Quelques mots sur ce bout de texte. Vous aurez reconnu le nom ma société, un peu de pub ne fait pas de mal :-) En réalité le principe du logo est que le "o" est remplacé par une sphère, originellement Neptune. Voici le logo original pour rappel (c'est pas bien amené ça ? !) :

Figure 112 - Le logo original

Dans l'exemple d'aujourd'hui j'ai donc repris (en très gros) le principe du logo original en prenant des libertés pour simplifier. Pour séparer le texte il suffit de taper "e-nax" d'un côté et "s" de l'autre. Rien de sorcier. Appliquer un effet de drop shadow est aussi simple que le drag'n drop on ne s'y arrêtera pas non plus.

Mais pour faire les réglages du texte (je n'étais pas sûr de la taille finale du projet au départ, ni de celui du planisphère) avoir deux `TextBlocks` séparés c'est enquiquinant. Je suis fainéant et l'idée de refaire le même choix de police, de taille, etc, sur deux objets m'ennuie rien que d'y penser…

Autre problème de mise en œuvre pure, voulant pouvoir adapter le texte à la taille du planisphère je voulais avoir toute la souplesse possible. La taille d'une fonte n'est pas assez. Il faut pouvoir déformer éventuellement le texte (un peu plus large, ou plus haut). Et là, gérer deux fois la même (exactement la même) série de transformations sur les deux textes est une perspective qui ne me disait rien du tout. Toujours cette satanée fainéantise ? Oui, bien sûr, mais pas seulement… Dans la réalité quand on fait une telle mise en page, les transformations ne sont pas appliquées au cordeau, mais à l'œil. On hésite. On fait des runs. On se demande… un peu plus large ? non. un peu plus haut ? … peut-être. Bref on tripote. Et se souvenir exactement de tous les tripotages sur un objet pour les reporter à l'identique sur un second relève de la plaisanterie, on n'y arrive jamais.

Pour régler ce problème il fallait une solution pour travailler sur les deux parties de texte en même temps tout en ayant la souplesse requise. Ici j'ai choisi de transformer le texte "e-naxs" (sans le "o") en Path.

Une fois le Path dégroupé j'ai pu tirer le "s" à sa place, Blend m'aidant à conservant l'alignement horizontal avec l'autre bout resté à sa place. Ensuite j'ai déplacé le premier bout de texte. A noter, le creux des lettres (comme le trou du "e" ou du "a") semblent avoir disparu. Ils vont revenir. Une fois les deux morceaux de texte placés, il suffit dans l'arbre visuel de les sélectionner ensembles et de faire ce qu'on veut,

notamment de les étirer dans un sens ou l'autre jusqu'à avoir trouvé le bon rapport visuel. Bien entendu cela écarte ou ressert les deux parties de texte, ce qui dérange pour le placement autour du planisphère. Mais il suffit de lever la sélection en cours et il est facile à nouveau de déplacer la partie gauche ou la partie droite pour qu'elles entourent la sphère correctement. On peut aller et venir entre sélection de tous les Paths pour avoir exactement les mêmes modifications et le mode désélectionné pour placer l'un ou l'autre.

Une fois le placement terminé (en réalité on y revient plusieurs fois, parce que la sphère change de taille par exemple) il suffit de resélectionner tous les Paths et de demander à Blend de refaire un chemin composé (compound path). Les creux des lettres sont de nouveaux visibles (ils ont toujours été là, mais dans un chemin composé les parties intérieures sont évidées donc visible alors qu'en version isolée chaque lettre est pleine bien que les chemins représentant les creux soient bien entendu toujours présents).

Bref, une astuce de mise en page qui peut être utilisée de mille façons mais surtout lorsqu'on doit appliquer des transformations à l'identique à des bouts de texte qui, normalement, sont isolés. Cela suppose que ce texte ne soit pas modifiable. Il faut donc une grande vigilance pour éviter les coquilles. Une fois converti en Path un texte ne se corrige plus !

Le planisphère

Je voulais animer un planisphère dans la boule. Le faire en générant AVI avec Swift 3D aurait été possible et plus simple. Appliquer une texture à une sphère est un job de base pour un soft de 3D. Faire tourner la sphère devant une caméra et en tirer une séquence animée relève aussi du b-a.ba.

Mais je ne voulais pas utiliser une ficelle aussi grosse (dans tous les sens du terme, un AVI ça pèse lourd aussi !).

Comment obtenir l'effet avec uniquement les outils de base de Blend ?

Deux aspects étaient à régler : l'aspect 3D de la sphère et le défilement 3D de la map.

La sphère 3D

Il s'agit là d'un tutor de base sur les objets "glossy". Pour obtenir l'effet d'une sphère on prend un rond (`Ellipse` xaml) et on lui applique un dégradé radial qu'on fait varier du noir au transparent en écrasant le dégradé pour qu'il forme une sorte

d'anneau. Choisir le bon anneau, les bonnes valeurs de noir et d'opacité est affaire de goût mais surtout de résultat. C'est assez délicat à régler pour obtenir le bon effet.

Ensuite on place un second rond (une autre `Ellipse`) par-dessus pour créer l'éclairage. On remplit ce rond d'un autre dégradé radial allant du blanc au transparent, mais dans l'autre sens : le plus clair et le plus visible au centre, le transparent à l'extérieur. Ici encore le réglage du dégradé est délicat.

Seulement cet effet est centré dans la sphère. C'est moche et cela ne suit pas forcément la direction de la lumière de l'ensemble. Pour cela Blend fourni un outil de modification du background : le `Brush Transform` (transformation de brosse). Il suffit, à l'aide de ce dernier, de déplacer le gradient pour obtenir l'effet désiré.

La map

J'ai rapidement cherché sur le Web une carte du monde simple. Une belle image à plat.

Plusieurs problèmes se posent alors : comment faire entrer l'image dans le cercle sans déborder, comment l'animer pour donner l'impression qu'elle tourne alors qu'elle est plate, et comment lui donner un aspect bombé pour l'illusion de la 3D ?

Première astuce : on créé un nouveau cercle (toujours une `Ellipse` xaml) qu'on placera en dessous des deux autres (ceux créant l'effet sphère). Pas de Stroke. Mais on va utiliser une astuce : l'ellipse est un Path qui supporte le mode `Tile`. Originellement (et sous WPF cela fonctionne d'ailleurs très bien) un tile est une répétition d'une même image pour créer un motif répétitif. Sous Silverlight le tile ne marche pas… C'est à dire que la création d'un motif répétitif ne répète rien. Silverlight 4 ne corrige pas ce problème. Mais on s'en moque un peu vu que ce n'est pas cet effet qu'on recherche ici. En plaçant l'image de la map en mode tile cela nous permet surtout de clipper cette dernière dans le cercle et de pouvoir animer le tile, donc l'image de fond grâce à un groupe de transformation rattaché au tile.

Grâce à cette astuce, il est maintenant possible de créer un `StoryBoard` en mode "`repeat forever`" faisant glisser la map de droite à gauche. Il faut bien régler les points de bouclage pour que l'impression de rotation de la map soit la meilleure.

L'effet 3D de la map

On a beau dire, partant d'une map plate la translation a du mal à convaincre. L'effet de sphère ajouté avec les ellipses trompe un peu l'œil mais pas suffisamment.

Hélas pas de 3D XAML en Silverlight. Il y a de la 3D mais en mode XNA qui ne m'intéresse absolument pas.

L'idée consiste alors à utiliser l'un des nombreux pixels shaders fourni avec SL4, ici le `Magnify`. On le pose sur l'ellipse contenant la map et ensuite il faut trouver les bons réglages ! Et ce n'est pas facile :-)

Une fois l'effet bien réglé, on fait tourner l'animation et, ô miracle, le "bombage" créé par Magnify renforce l'effet visuel d'une sphère 3D qui tourne.

Rien n'étant parfait, les réglages les meilleurs que j'ai trouvés avaient tendance à faire "déborder" le dessin en dehors de l'ellipse. Fâcheux, quand ça déborde ça fait sale.

Pour régler ce dernier problème, on reprend une ellipse, de la même taille que la précédente et grâce à Blend on demande "`Path/Make Clipping Path`". Il suffit de sélectionner la cible (l'ellipse contenant la map) et le tour est joué : l'effet déborde toujours, mais il est clippé et visuellement on ne voit plus rien.

L'illusion est créée, le planisphère tourne, sans 3D, en partant d'une image plate.

L'image originale étant un jpeg, pas de transparence. Et je voulais conserver un effet "bulle". C'est donc sous Photoshop que s'effectuent les derniers réglages : colorisation, création d'un png avec transparence, etc.

On change l'image originale par le png avec sa transparence et le travail est terminé.

Reste à déclencher l'animation, toujours sans code, et pour cela on place un Behavior "`ControlStoryBoardAction`" sur l'ellipse qui sera animée. Le trigger se fait sur l'événement `Loaded`, histoire de s'assurer que tous les morceaux sont bien chargés avant de déclencher le `Begin` de l'animation. Ça démarre donc tout seul et sans souci.

Pour mieux analyser tout cela le code source est obligatoire : SLWord.zip

Conclusion

Une sorte de tutor graphique sans image ça peut sembler étrange et à contre-emploi. Mais je ne voulais pas vous distraire avec des petits mickeys. C'est la démarche qui compte ici bien plus que les images et même plus que le rendu final qui n'est pas une œuvre d'art. Comme je le disais en introduction c'est un prétexte. Pour parler de Blend. Car j'entends encore trop souvent certains développeurs affirmer que tout peut être fait sous Visual Studio. C'est une aberration totale. Même VS 2012 et sa puissance ne sont d'aucune aide dans les manipulations ici exposées. C'est comme

ceux qui affirmaient qu'un site Web ne nécessite que le bloc-notes. A la niche les rétrogrades dinosauresques et réactionnaires ! ☺

Il faut absolument un logiciel dédié au design. Pas au dessin, au design. Un outil capable d'importer du Photoshop, de l'illustrator, mais qui soit aussi un soft de dessin. Il doit aussi contenir les prémices et l'intelligence des actions que l'application devra réalisée (behaviors par drag and drop par exemple). A mi-chemin entre Illustrator, Flash et Visual Studio, Blend est un produit unique. Plus on l'utilise plus on l'aime…

Bref, j'espère avoir ouvert quelques petites portes avec ce billet, quelques envies aussi, ne soyons pas trop présomptueux. Ma devise est "ne pouvant les sauver tous, si jamais je n'en sauve qu'un seul, alors tout cela a un sens" !

138 sons gratuits pour vos applications

Ajouter des petits clicks, des petits bruits pour souligner une validation, avertir l'utilisateur, marquer une erreur de saisie ou de comportement du programme peut s'avérer intéressant dans beaucoup d'applications pour peu qu'on n'abuse pas du procédé et que les sons en questions soient assez courts. Mais quels sons ? Je vous propose ici une petite bibliothèque de 138 sons prêts à l'usage.

L'importance du son

Pour créer un environnement immersif il ne suffit pas que l'application soit fluide, réactive, animée, il faut aussi penser à son environnement sonore. La sonorisation d'une application est comme tout le reste un travail délicat qui demande à connaître l'environnement physique des utilisateurs, leurs habitudes, les fonctions du logiciel, etc. Il s'agit d'un travail de design au même titre que l'UI.

La bonne illustration sonore d'une application peut la rendre plus sympathique, plus réaliste ou plus futuriste, mais les sons peuvent et doivent jouer un rôle fonctionnel. Avertir par des effets sonores qu'une saisie est erronée ou au contraire qu'elle a bien été prise en compte évite à l'utilisateur de scruter l'écran à la recherche d'un message d'erreur ou de validation par exemple.

Designer la « bande son » d'une application, ce qui peut aller jusqu'à de vraies musiques, est donc une tâche qui, sérieusement, ne peut être effectuée qu'en relation étroite avec les concepteurs et les designers. On ne peut pas sonoriser une

application « à distance ». Une « charte sonore » se conçoit dans la cohérence comme une charte graphique, elle créée une signature unique et reconnaissable faisant partie intégrante de l'originalité du logiciel.

La petite librairie que je vous offre aujourd'hui ne peut donc en aucun cas prétendre résoudre tous les besoins en la matière pour n'importe quelle application. Il s'agit principalement de satisfaire rapidement les besoins les plus courants et d'attirer votre attention sur cet aspect du design encore plus souvent ignoré que la partie purement graphique…

Des sons résolument électroniques

C'est un parti pris. J'ai conçu cette librairie avec des générateurs spéciaux reproduisant des sons de type ancienne console de jeu, pour la plupart assez courts, en format Wav 44100 KHz 16 bit mono. Un peu dans l'esprit du SID – Sound Interface Device – connu aussi sous son vrai nom de MOS 6581 et qui était le microprocesseur spécialisé dans la génération des sons du Commodore 64.

Figure 113 - Le vaisseau spatial va décoller !

Pour être universelle cette librairie est au format WAV, format pur non compressé et non altéré lisible par tous les systèmes audio de toutes les plateformes à ma connaissance.

Il est possible de compresser le format WAV dans tous les autres formats comme MP3, il existe de nombreux logiciels gratuits pour faire ce genre d'opération.

Tous les sons ont une taille comprise entre 3 et 60K majoritairement, quelques-uns faisant dans les 200K. Cette restriction est un choix délibéré afin que les sons ne pèsent presque rien dans les ressources des applications ce qui est important avec Silverlight, Windows Phone ou WinRT.

Les sons sont normalisés et non traités, si vous êtes un peu bricoleur audio vous pourrez bien entendu les post-traiter pour ajouter des effets (réverbération, écho, renversement temporel, etc). Mais faites attention à conserver l'esprit de chaque son : par exemple une validation ou un son pour attirer l'attention doivent être brefs, dans une gamme de fréquence bien précise, avoir une enveloppe adaptée... sinon l'effet est perdu. Et lorsqu'un son n'est plus dans l'esprit de sa fonction il devient tout simplement une nuisance !

Un choix adapté aux cas les plus fréquents

La sélection est par force assez arbitraire, toutefois j'ai essayé de regrouper les sons par type pour couvrir les principaux besoins usuels, même si cela reste assez subjectif.

On trouve ainsi 12 catégories :

- 15 Alarmes. Conçus pour marquer une situation exceptionnelle.
- 10 "Attention". Conçus pour attirer l'attention.
- 21 Bips. Des bips sonores de tout genre.
- 3 Bubbles. Des sons évoquant des bulles sans vouloir imiter le son de l'eau.
- 10 "error". Conçus pour attirer l'attention sur une erreur dans l'application.
- 10 explosions. Ça explose...
- 10 "gaz". Des sons « gazeux »
- 17 "laser style". Le type de son « laser » classique de la SF.
- 20 "divers". Difficiles à classer mais intéressants à bien placer !
- 3 "scratches". Effet de scrath.
- 6 "validations". Conçus pour évoquer une validation par l'application (acceptation d'une valeur, acquiescement, prise en compte...)
- 8 "weapons". Des bruits d'armes de SF.

Conception et copyrights

Comme je l'évoquais plus haut ces sons ont été conçus par moi-même à l'aide de synthétiseurs numériques dont je n'ai utilisé que le minimum de fonctions pour rester dans l'esprit "console de jeu" ou Commodore 64.

Le fichier Zip

Sounds.zip (3,48 mb)

Animations, Sons, Synchronisation, Ease in/out sous Blend

Effets visuels et sonores font partie des nouveaux éléments avec lesquels le concepteur de logiciel doit apprendre à jouer pour créer des interfaces attrayantes. Les réflexes du développeur le porte à se jeter sur son clavier, tout comme ceux du graphiste le pousse à sortir son bloc et son crayon. Comme tout réflexe il faut savoir s'en méfier...

Prenons l'exemple de l'affichage ci-dessous qui simule un oscilloscope :

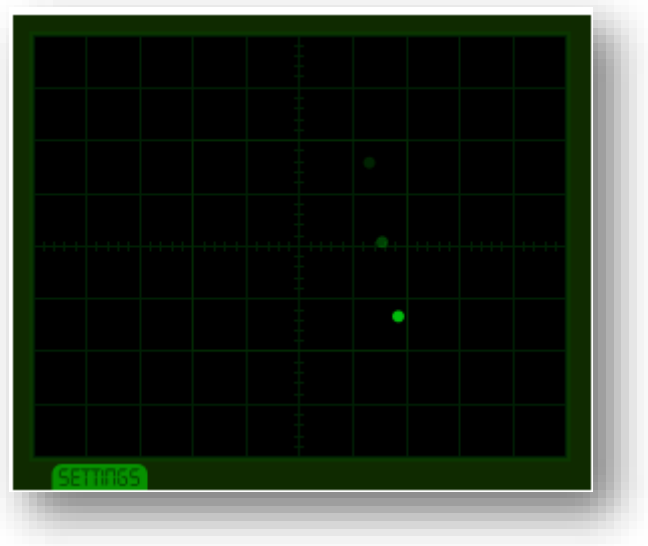


Figure 114 - Simulation d'un oscilloscope

(Cliquez sur le titre de chapitre pour accéder au billet original sur Dot.Blog et jouer avec l'exemple Live. Passez la souris sur "settings" pour ouvrir la fenêtre de paramètres qui permet de jouer sur la luminosité des carrés de ciblage, le flouté du spot et sur le volume du bip)

Pour créer l'effet du spot et de ses échos visuels le développeur aura donc tendance à se ruer sur son clavier et, s'inspirant des sprites (voir mon billet sur la création d'un effet de neige), il voudra créer une classe pour le spot, puis recherchera la formule mathématique de la courbe du déplacement du spot pour animer ce dernier, les échos étant créés à la volée avec une animation d'opacité décroissante. Tout cela va fonctionner parfaitement. Au bout du compte on disposera même d'un véritable affichage de type oscilloscope réutilisable avec des courbes différentes.

Mais s'agissant juste d'un affichage, d'un effet visuel, n'est-ce pas un peu lourd ?

On peut en effet réaliser la chose bien plus rapidement et sans avoir de debug à faire (puisque pas de code) si on "pense" design, pur dessin. Et au lieu d'ouvrir Visual Studio, ouvrons Blend...

Bien plus que l'exemple lui-même (assez simple) c'est cette démarche sur laquelle je souhaite attirer votre attention dans ce billet, créer un faux oscilloscope n'ayant pas beaucoup d'intérêt en soi.

Côté réalisation comment cela se passe alors ? Deux contraintes nous sont données : animer un spot selon une courbe "réaliste" et avoir un rendu de la rémanence de l'écran.

Prenons le déplacement pour commencer: il suffit de positionner un cercle à gauche de l'écran de l'oscilloscope (une image réalisée sous Design), d'ouvrir un `StoryBoard` et de créer une `keyframe` à 2 secondes du début puis de déplacer le cercle jusqu'à le faire sortir de l'écran. Le mouvement horizontal est réglé. Concernant le mouvement vertical et la simulation d'une courbe amortie, l'astuce consiste à déplacer en début d'animation le spot sur l'axe `Y` pour le faire monter. C'est tout, l'effet de courbe sera rendu en utilisant astucieusement les modes d'Ease in et out de Blend.

Le point de déplacement Y sera doté d'un *Quintic In* pour amortir le mouvement, le dernier point de l'animation sera lui doté d'un *Elastic Out* avec 5 oscillations et un *Spingness* de -2. On peut bien entendu ajuster ces éléments à sa guise.

L'effet du déplacement est ainsi rendu sans développement. Bien entendu il s'agit d'un simple affichage qui n'effectue pas le rendu d'une "vraie" courbe passée en paramètre. Mais ce n'était pas ce qu'on voulait. Juste un effet visuel "réaliste" en y passant le moins de temps possible.

Pour l'écho visuel ? La ruse est un peu grosse mais elle passe plutôt pas mal : il suffit de copier le spot et son animation deux fois puis de décaler légèrement les timelines

vers la droite. On change ensuite l'opacité du 1er et du 2d écho (de façon décroissante). L'effet est ainsi rendu sans aucun code.

Un oscilloscope plus vrai que nature ! Il ne lui manque que la parole pourrait-on dire. Justement, ajoutons un son pour "faire plus vrai" (disons plutôt "plus cinéma" car les vrais oscillos ne font pas de bip). Au départ on avait envisagé de lancer l'animation du spot en mode bouclage infini. Hélas les storyboards n'ont qu'un seul événement, *Completed*, qui ne sera jamais déclenché en plus (puisque bouclage infini). Comment attraper le début de l'animation pour jouer le son et le synchroniser avec l'affichage du sport ?

Pas facile, et pas vraiment de solution pour l'instant. Mais il n'est pas interdit de faire marcher ses neurones ! Nous allons simplement supprimer le mode infini à la boucle d'animation du spot. Ensuite nous allons pouvoir gérer l'événement `Completed` de celle-ci. Dans le gestionnaire il suffira alors de relancer l'animation, on en profitera alors pour activer le son qui est placé dans un `MediaElement` (donc `Stop()` puis `Play()`).

Le son est maintenant synchronisé avec l'affichage !

Mais si le son devait être déclenché au milieu de l'animation ? Hmm.. Là nous serions un peu coincé je pense. Tout de suite le développeur dirait "ha je l'avais bien dit, si j'avais fait une gestion de sprite j'aurais presque fini!". Peut-être. Mais revenir à une animation pas à pas pour avoir la main n'est toujours pas ce qu'on veut.

Il est dommage que XAML ne puisse pas animer le play et le stop d'un `MediaElement`, c'est un point faible. Mais si tel était notre besoin, là encore, il serait possible de réfléchir un peu au lieu de repartir sur une animation pas à pas complexe.

Par exemple nous pourrions créer un petit `UserControl` possédant une propriété `PlaySound` (une propriété de dépendance bien entendu sinon elle ne sera pas animable) pouvant passer de 0 à 1 pour bénéficier de l'animation des `Doubles` sachant que seul le passage à la valeur exacte 1.0 déclenchera la séquence Stop/Play du `MediaElement` intégré. Dès lors ne resterait plus qu'à poser ce contrôle sur notre application et d'animer sa propriété `PlaySound` en la faisant passer de 0 à 1.0...

Voilà un bon exercice sur lequel je vous laisse vous amuser un peu. La solution dans un prochain billet !

Pour le reste de l'exemple il n'y a rien de bien compliqué. La petite fiche des settings est dessinée sous Expression Design, son titrage a été converti en Paths pour éviter

d'avoir à intégrer la fonte à l'application. Les sliders sont reliés aux propriétés qu'ils modifient par la possibilité de binding élément à élément.

J'allais oublier, le code du projet (l'exemple live est en Silverlight) : MovingSpot.zip (91,58 kb)

AnimatableSoundPlayer. Ou comment synchroniser du son sur une animation

Dans le billet précédent je vous parlais d'animation et de sons et je vous présentais une petite démonstration (l'oscilloscope) dont le but était de montrer qu'on pouvait rapidement obtenir un effet visuel assez complexe sans programmation pour peu qu'on se donne la peine d'utiliser Blend et ses neurones... Pour synchroniser le bip avec le spot lumineux j'avais alors utilisé une ruse en indiquant qu'hélas Silverlight ne permettait pas de **synchroniser du son dans une animation**.

Je vous proposais alors une solution plus élégante en promettant une implémentation le temps de vous laisser réfléchir.

Regarder d'abord la démo ci-dessous (accès en cliquant sur le titre de ce chapitre pour aller sur la page du billet original sur Dot.Blog). En cliquant sur "`Start`" vous lancerez une animation (`StoryBoard`) dont le but premier est de déplacer la boule verte un peu comme dans un billard. Elle rebondit sur des taquets pour finir dans un gobelet. Bien entendu tout cela est assez moche, les beaux dessins ne font pas partie de la question à traiter ! Mais, en revanche, et si votre carte son fonctionne, vous noterez que plusieurs sons peuvent être entendus en parfaite synchronisation avec les "chocs" de la boule sur les taquets ou dans le gobelet final. Je vous laisse essayer et on en reparle :

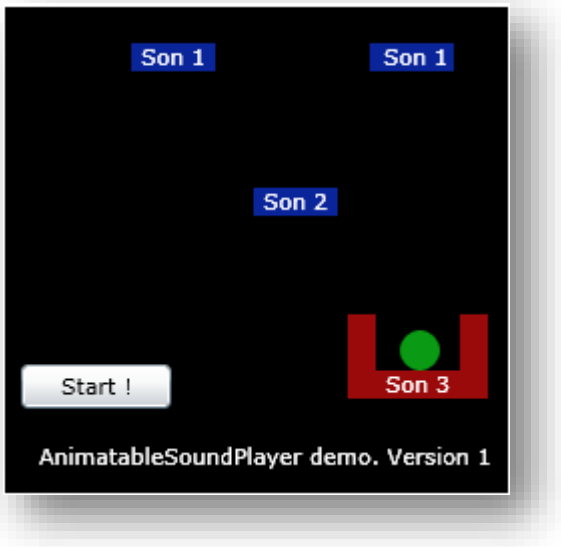


Figure 115 - capture de l'application exemple utilisant du son synchronisé à l'animation

Bon. Vous avez vu et aussi entendu ?

Comment est-ce possible (sachant que pour l'utilisateur Blend qui créé l'animation tout cela ne réclame aucun code) ?

Forcément il y a du code... En fait un petit `UserControl` dont le principe est fort simple.

AnimatableSoundPlayer

C'est son petit nom. Un joueur de sons animable. Joueur de sons car ce `UserControl` n'a pas de visuel. On pourrait en réalité jouer des vidéos de la même façon il suffirait de relooker le contrôle, mais ce n'était pas le but. Joueur de son animable car ce qui manque au `MediaElement` c'est bien d'avoir des propriétés "*animables*" pouvant être modifiées sur une timeline.

Le `UserControl AnimatableSoundPlayer` est ainsi une coquille presque vide, il ne contient qu'un `MediaElement`. Tout le reste est du code, fort peu en réalité.

Dans un premier temps j'ai ajouté plusieurs propriétés de dépendance qui relaient les propriétés du `MediaElement` : la `Source` (une Uri), le `Volume`, etc. Ensuite j'ai fait de même pour les principales méthodes (`Play`, `Stop`...).

Deux choses ont été ajoutées. Côté méthodes j'ai créé `PlayFromStart()`. Elle ne fait que faire "`Stop(); Play();`". C'est tout bête mais très souvent on a besoin

d'enchaîner ces deux méthodes pour s'assurer qu'un son est bien rejouer depuis le début, il faut rembobiner la bande avec `Stop()` pour réécouter un son.

La seconde chose ajoutée est la plus importante. Il s'agit de la **propriété de dépendance** `AnimatablePlay` de type `double`. Pourquoi double ? Simplement parce XAML sait bien animer des doubles et pas forcément autres choses... Le mécanisme intéressant se trouve dans la méthode `AnimatablePlayChanged`, le *callback* de modification de la valeur initialisé lors de la création de la propriété de dépendance. A l'intérieur de ce callback nous trouvons la logique de cette propriété :

```
#region AnimatablePlay property
        /// <summary>
        /// Gets or sets the animatable play.
        /// Value = 0 means <see cref="Stop"/>
        /// Value sup.to 0 means <see cref="PlayFromStart"/>
        /// Value inf.to 0 means <see cref="Play"/> / <see cref="Pause"/>
        /// </summary>
        /// <value>The animatable play.</value>
        [Category("Media")]
        public double AnimatablePlay
        {
            get { return (double)GetValue(AnimatablePlayProperty); }
            set { SetValue(AnimatablePlayProperty, value); }
        }

        public static readonly DependencyProperty
            AnimatablePlayProperty =
             DependencyProperty.Register("AnimatablePlay",
                                         typeof(double),
            typeof(SynchedSoundPlayer),
            new PropertyMetadata(0.0d, AnimatablePlayChanged));

        private static void AnimatablePlayChanged(DependencyObject d,
                                  DependencyPropertyChangedEventArgs e)
        {
            var mp = ((SynchedSoundPlayer)d);
            if ((double)e.NewValue > 0d) mp.PlayFromStart();
            if ((double)e.NewValue == 0d) mp.Stop();
            if ((double)e.NewValue < 0d)
            {
                if (mp.internalMP.CurrentState ==
                          MediaElementState.Playing) mp.Pause();
                else mp.Play();
            }
        }

        #endregion
```

Code 53 - Extrait du code de SynchedSoundPlayer

Simple et efficace : toute valeur positive déclenche un "`PlayFromStart`", d'où l'utilité de cette méthode qui nous assure que le son est bien rejoué depuis le début. Toute

valeur *nulle* appelle "`Stop`" et toute valeur négative entraîne un cycle *Play/Pause* selon l'état actuel du contrôle.

Le `UserControl` relaie aussi l'événement `MediaFailed` si on désire être prévenu en cas de difficulté rencontrée par le `MediaElement` pour charger le son (qui peut être dans le Xap comme dans la présente démo ou sur un serveur distant ou même en ressources internes de l'application).

Grâce à cette convention il devient très simple de synchroniser des sons avec une animation !

Si on suit l'exemple live proposé plus haut : Trois sons sont joués, un numéro est indiqué sur chaque "obstacle" pour mieux se repérer. Sur ces trois sons il y en a un qui utilisé deux fois. Pour synchroniser ces trois sons on reprend la timeline de la boule verte et à chaque fois qu'on désire qu'un son soit joué on incrémente sa propriété `AnimatablePlay`. Simple. Vu la taille des valeurs maximales d'un double, on ne risque pas de tomber sur une limite... surtout qu'il n'est pas interdit de faire un retour à zéro quand on le veut.

Un petit détail à savoir : quand on créé une animation, par défaut XAML effectue une interpolation des valeurs fixées dans les `keyframes` qui se suivent. Si vous tapez la valeur `1` à la frame `A` et que vous tapez `2` à la frame `B`, durant le temps qui sépare `A` de `B` la valeur augmentera progressivement de `1` vers `2`. C'est le comportement par défaut, ce qui est bien utile puisque justement le plus souvent on désire avoir une continuité dans le changement des valeurs (déplacements notamment). En revanche il y a des cas où on préfère que la valeur change brutalement, une "anti animation" en quelque sorte. Cela est parfois utile. Pour utiliser `AnimatableSoundPlayer` ce n'est pas une option, *c'est une obligation*. Il faut que les valeurs ne changent que lorsqu'une nouvelle `keyframe` est rencontrée. Sinon le son sera rejoué sans cesse depuis le début à toutes les valeurs intermédiaires, ce qui n'est pas du tout l'effet recherché.

Pour arriver à ce résultat il suffit de ne pas oublier de paramétrer les `keyframes` servant à animer le son afin qu'à la place d'une `EasingDoubleKeyFrame` le type devienne `DiscreteDoubleKeyFrame`. Sous Blend il suffit de cliquer sur la keyframe en question et de la passer en mode "**Hold in**". C'est la seule contrainte du composant.

Et voilà ! Avec un peu d'imagination on peut parfaitement créer un composant réutilisable qui ne prend que quelques lignes de code et qui permet de synchroniser du son dans une animation XAML. Ce n'est pas magique ?

Je suis certain que vous trouverez des tas d'améliorations à porter au composant, alors n'hésitez surtout pas à m'en faire part je pourrais même diffuser vos versions modifiées si vous le voulez.

Le code du projet fourni contient l'exemple complet ainsi que le code du composant et les sons utilisés.

Quelques restrictions : Si vous modifiez le code obligez-vous à publier le code source de votre version gratuitement. Si vous publier un article ou un billet de blog, soyez sympa et indiquez le lien vers mon billet. Si vous utilisez mon code dans un projet, commercial ou non, dites le moi ça me fera plaisir.
Si vous respectez ce petit deal, alors faites ce que voulez du code. Dans la négative que les foudres du Grand Bug Céleste s'abattent sur vous et chaque octet que vous coderez jusqu'à la fin de vos jours (c'est bien horrible ça non ?).

Le code du projet : SyncSound.zip (209,05 kb)

Détection de touches clavier multiples

XAML est conçu pour s'intégrer dans une chaîne de développement "sérieuse". Son but est principalement la création d'applications riches, le plus souvent des applications métier. Mais XAML peut aussi servir à créer des jeux même si d'autres plateformes sont plus spécialisées et mieux adaptées. En réalité les techniques visuelles du jeu ne sont pas à négliger pour **créer des applications professionnelles disposant d'interfaces innovantes**. C'est pour cela que j'aborde le sujet du jour, même une application de gestion moderne doit avoir une UI à la hauteur, immersive et **ludique**, et ainsi se programmer avec les techniques plutôt réservées aux jeux.

Raison de plus pour se pencher sur la question. Dans d'autres billets je vous ai parlé de la gestion du son (notamment en vous proposant un petit composant capable de rendre possible la synchronisation son/animation), aujourd'hui je vais aborder la **gestion du clavier**. Pour ce qui est de la boucle de jeu, je vous renvoie à mon billet exposant l'exemple de la neige qui tombe.

D'abord un petit exemple live (donc en Silverlight sur Dot.Blog – cliquez sur le titre de ce chapitre pour accéder au site et à l'exemple). Une fois que vous aurez cliqué sur le rectangle "*click here to start*" vous pourrez déplacer le carré rouge et changer sa taille. Les déplacements se font à l'aide des 4 flèches de direction. Pour agrandir l'objet on utilise la combinaison de touche `Shift-KeyUp` (et `Shift-KeyDown` pour diminuer la

taille). La touche `Ctrl` est un accélérateur pour les mouvements. `Ctrl-KeyUp`/`Down`/`Left`/`Right` permet donc d'aller plus vite. Ce qui est intéressant dans cet exemple est bien entendu la **détection des appuis simultanés sur les touches** et la gestion de ceux-ci. Jouer deux secondes et on y revient (par exemple gauche et droite en même temps pour aller en diagonal).

Capture fixe de l'application testable en ligne :

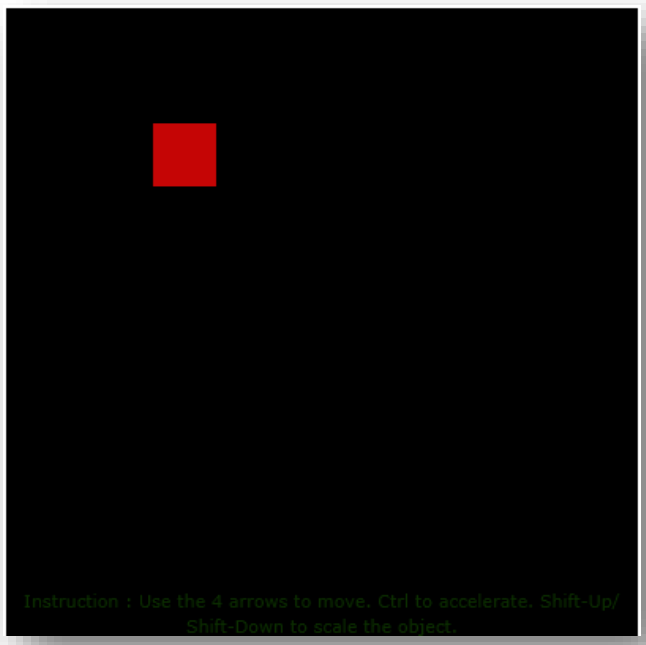


Figure 116 - capture de l'application exemple détection mutlti touche

Tout le problème ici est donc de détecter plusieurs touches appuyées en même temps. Selon les profils XAML on dispose d'une gestion de clavier plus ou moins simple. Pour être générique nous allons nous débrouiller avec les événements `KeyDown` et `KeyUp`.

Pour une gestion de déplacement rudimentaire (les 4 directions par les 4 flèches) cela pourrait parfaitement convenir. Mais comment gérer les déplacements en diagonal (`KeyUp+KeyLeft` par exemple) ou pire des choses du genre `Shift-Ctrl-KeyUp`/`Barre d'espace` ? Il faudrait se rappeler dans l'événement `KeyDown` si l'une ou l'autre des touches a été enfoncée et si elle est toujours *Down*... De plus aucune assurance de l'ordre dans lequel les touches vont arriver (la simultanéité n'existe pas du point de vue du pilote clavier, une touche est forcément avant l'autre).

Bref cela semble d'un seul coup se compliquer énormément !

Oui et non. Oui puisque la solution n'est pas directe, non parce que celle-ci n'est pas si compliquée à trouver.

La classe KeyLogger

Ne cherchez pas dans l'aide du Framework, cette classe n'existe pas ! Il s'agit justement de la classe que je vous propose aujourd'hui de créer pour gérer ce petit problème des touches *simultanées*.

Problématique

On doit pouvoir à tout moment dans la boucle principale du jeu de l'exemple tester si certaines combinaisons de touches sont actives ou non. On doit aussi pouvoir tester l'appui sur une seule touche. Dans une application de gestion il n'y a pas de « boucle de jeu », on utilisera alors une autre stratégie pour tester les touches, comme par exemple vérifier les informations du `KeyLogger` à chaque événement clavier (d'autres méthodes sont envisageables selon le contexte).

Solution

Il n'y a pas de miracle, pour savoir si une ou dix touches sont appuyées il faut se souvenir de toutes les touches ayant reçu un `KeyDown` et pas encore de `KeyUp`. Se souvenir, en informatique, cela signifie stocker des données. C'est bien ce que nous allons faire en créant un tableau dont les éléments seront des booléens et dont l'index sera la valeur de l'énumération `Key` (classe du Framework listant les touches accessibles du clavier). Ne reste plus qu'à gérer nous-mêmes les `KeyUp` et `KeyDown` pour mettre à vrai/faux les cases correspondantes dans le tableau. Pour tester une combinaison de touches (ou une touche) il suffit alors d'interroger le tableau pour savoir si les touches que nous voulons tester sont toutes à true.

Le code

Une fois le problème et la solution posés, il ne reste plus qu'à coder. Le `UserControl` principal de l'application exemple commence par créer une instance du `KeyLogger` et l'attache à lui-même. On pourrait fort bien attacher un `KeyLogger` différent à différentes zones écran, ici nous optons pour un seul gestionnaire centralisé ce qui semble être le cas d'utilisation le plus fréquent.

La détection du clavier ne marche que si le contrôle a le focus. Donner par défaut celui-ci au `UserControl` n'est pas forcément évident, selon les versions de XAML les astuces différent. D'où la petite astuce du bouton de démarrage : en cliquant dessus je lance l'animation qui le fait disparaître et qui diminue l'opacité des instructions mais surtout j'en profite pour réclamer le focus. Cette technique a l'avantage de fonctionner à tous les coups : en cliquant sur l'invite l'utilisateur permet l'activation du focus… C'est une forme de manipulation, faire faire à autrui ce qu'on ne peut faire directement sans qu'il sache qu'on se sert de lui. L'informatique c'est un peu machiavélique parfois !

Le code ci-dessous n'est pas d'une grande complexité, je vous laisse le lire. Il est de plus commenté.

```
public class KeyLogger
    {
        #region Private Fields
        readonly bool[] isPressed = new bool[255];
        private FrameworkElement targetElement;
        #endregion

        #region Property
        /// <summary>
        /// Gets or sets the target element.
        /// (Setting the target is equivalent to calling Attach);
        /// </summary>
        /// <value>The target element.</value>
        FrameworkElement TargetElement
        {
            get { return targetElement; }
            set { Attach(value); }
        }
        #endregion

        #region Public methods
        /// <summary>
        /// Clears the key array.
        /// </summary>
        public void ClearKeyPresses()
        {
            for (int i = 0; i < 255; i++)
            {
                isPressed[i] = false;
            }
        }

        /// <summary>
        /// Clears a given key
        /// </summary>
        /// <param name="key">The k.</param>
        public void ClearKey(Key key)
        {
            isPressed[(int)key] = false;
        }
```

```
        /// <summary>
        /// Attaches the logger to the specified target.
        /// </summary>
        /// <param name="target">The target.</param>
        public void Attach(FrameworkElement target)
        {
            if (target == null)
            {
                Detach();
                return;
            }
            if (targetElement != null) Detach();
            ClearKeyPresses();
            targetElement = target;
            target.KeyDown += target_KeyDown;
            target.KeyUp += target_KeyUp;
            target.LostFocus += target_LostFocus;
        }

        /// <summary>
        /// Detaches the logger from the specified target.
        /// </summary>
        public void Detach()
        {
            if (targetElement == null) return;
            targetElement.KeyDown -= target_KeyDown;
            targetElement.KeyUp -= target_KeyUp;
            targetElement.LostFocus -= target_LostFocus;
            ClearKeyPresses();
            targetElement = null;
        }

        /// <summary>
        /// Determines whether a given key is pressed.
        /// </summary>
        /// <param name="key">The key.</param>
        /// <returns>
        /// <c>true</c> if the key is pressed]; otherwise, <c>false</c>.
        /// </returns>
        public bool IsKeyPressed(Key key)
        {
```

```
            return isPressed[(int)key];
        }

        /// <summary>
        /// Determines whether a given set of keys are pressed altogether.
        /// </summary>
        /// <param name="keys">The key array.</param>
        /// <returns><c>true</c> if all keys are pressed; otherwise,
                                                    <c>false</c>.</returns>
        public bool AreAllKeyPressed(Key[] keys)
        {
            var i = 0;
            foreach (var k in keys)
            {
                if (IsKeyPressed(k)) i++;
            }
            return i == keys.Length;
        }
        #endregion

        #region private stuff
        private void target_KeyDown(object sender, KeyEventArgs e)
        {
            isPressed[(int)e.Key] = true;
        }

        private void target_KeyUp(object sender, KeyEventArgs e)
        {
            isPressed[(int)e.Key] = false;
        }

        private void target_LostFocus(object sender, EventArgs e)
        {
            ClearKeyPresses();
        }
        #endregion

    }
```

Code 54 - Code du KeyLogger

Conclusion

Remplacer le rectangle rouge par une petite fusée, ajoutez des blocs rocheux, détectez la barre d'espace pour tirer, utilisez la gestion de sprites pour animer les tirs de laser, agrémentez d'une gestion de collision, complétez par une gestion de l'inertie des mouvements... et vous aurez une réplique de Asteroid ! Yaka. ☺

Vous pouvez aussi penser à une application industrielle où les touches du clavier permettent de manipuler une machine-outil et sa représentation graphique à l'écran ou à une application de gestion permettant de modifier les données d'un compte graphiquement, le clavier et les multitouches jouant le rôle de raccourcis vis-à-vis de la souris. Etc.

Déplacer des objets ou contrôler des animations via le clavier est l'une des bases de la création de jeux. Avec l'exemple de la neige qui tombe nous avions déjà vu la gestion des sprites, vous disposez maintenant des briques pour commencer à réfléchir à des petits jeux sympas, ou bien à l'UI de l'application de gestion de demain qui explosera le box-office...

Code source du projet : KeyTrapper.zip (78,30 kb)

Détecter les touches Alt, Ctrl ...

Je vous ai présenté ma classe `KeyLogger` qui permet de détecter toutes les touches et notamment la pression de plusieurs touches en même temps. Cela est idéal pour les jeux notamment. Mais dans certains cas on a juste besoin de détecter la pression sur les touches de modification telles que `Control`, `Alt` ou `Pomme` sur un Mac (dans le cas de Silverlight pour cette dernière).

Si le besoin est ponctuel et si la détection simultanée d'autres touches n'est pas importante, on peut utiliser un code plus simple (quoi que la classe KeyLogger soit déjà écrite pour vous et qu'il suffit de l'utiliser).

Pour connaître les "modificateurs" (*modifiers* en anglais) il suffit de consulter `KeyBoard.Modifiers` qui retourne tous les modificateurs appuyés à un instant donné.

Pour interpréter le résultat voici un bout de code :

```
1: var keys = Keyboard.Modifiers;
2: var shiftKey = (keys & ModifierKeys.Shift) != 0;
3: var altKey = (keys & ModifierKeys.Alt) != 0;
4: var appleKey = (keys & ModifierKeys.Apple) != 0; // pour les Mac
5: var controlKey = (keys & ModifierKeys.Control) != 0;
6: var windowsKey = (keys & ModifierKeys.Windows) != 0;
```

Code 55 - Détection des touches "modificateurs"

XML/XAML pretty printer gratuit

Il arrive souvent que du code XML soit produit "au kilomètre" sans mise en forme particulière. Même si Internet Explorer sait afficher un tel fichier en le mettant en forme automatiquement, on souhaite parfois disposer d'une version formatée lisible par un humain.

ODPrettyXml, un utilitaire console très simple qui ne fait que ça... Il traite les fichiers XML, mais aussi du XAML sans souci. Toutefois vous remarquerez que `ODPrettyXml` travaille toujours sur un fichier de sortie différent de l'original, certaines transformations pourraient avoir des effets non souhaités. L'utilitaire est donc avant tout conçu comme un "*pretty printer*" dont la vocation est de rendre le document plus lisible pour un humain. Les fichiers produits, même s'ils restent fonctionnels, n'ont pas vocation à être utilisés en programmation.

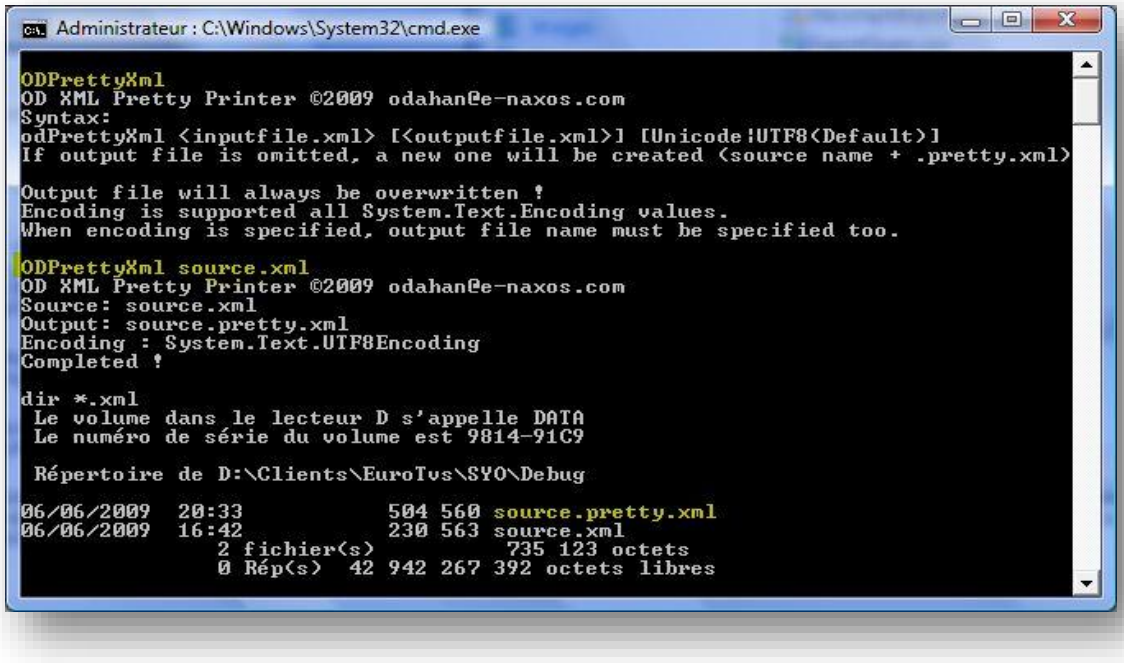
Pour le mode d'emploi, tapez `ODPrettyXml` sous console, l'aide sera affichée. Le programme ne demandant aucune saisie, il est possible de l'utiliser dans des fichiers de commandes (ou des batchs).

La syntaxe la plus habituelle est "`ODPrettyXml <nom du fichier>`" qui fabriquera automatique un fichier de sortie de même nom se terminant par "`pretty`" suivi de l'extension du fichier original (par exemple: `toto.xml` donnera `toto.pretty.xml`).

Si vous tapez "`ODPrettyXml ?`" vous obtiendrez la liste de tous les encoders connus et utilisables avec leur code page. C'est le nom qu'il faut utiliser en 3eme paramètre de `ODPrettyXml`. Par exemple pour utiliser Unicode il faut taper "`ODPrettyXml <source> <sortie> utf-16`". Quand un encodeur est spécifié, il faut aussi saisir le nom du fichier de sortie (2d paramètre).

Dernière remarque, `ODPrettyXml` ne fait qu'encoder le fichier et le mettre en forme avec des indentations, notamment il ne contrôle pas si l'encodage demandé est conforme à celui déclaré dans le fichier source. Un fichier indiquant qu'il est codé en

UTF-8 peut être encodé en UTF-16, son entête indiquera toujours UTF-8, le fichier n'est pas modifié par `ODPrettyXml`.



Code 56 - Exemple d'exécution de PrettyXML

Téléchargement : odPrettyXml.exe (34,00 kb)

(Exécutable .NET 3.5+, mode console)

Code Source : odPrettyXml.rar (16,21 kb)

(Projet VS complet. Le fichier de signature électronique est absent vous devrez en créer un autre).

PS: l'aide du logiciel a quelques coquilles, à vous de les trouver et de les corriger ☺

XAML pour WPF

Les bases de XAML sont valables dans tous les profils où ce langage est intégré, toutefois il existe des contextes particuliers où certaines astuces ne prennent de sens que pour un profil XAML donné.

Tout ce qui a été dit jusqu'à maintenant s'applique bien entendu à WPF c'est pourquoi cette section qui traitera spécifiquement de WPF ne sera pas très longue.

La section suivante traitement des aspects relatifs au XAML de Silverlight.

Le Tome 6 de ALL DOT BLOG étant consacré à WinRT les spécificités de cette plateforme ne seront pas évoquées ici et le lecteur est invité à télécharger ce tome dédié à WinRT.

On notera enfin que certaines solutions traitées dans la section WPF ou Silverlight peuvent parfois s'appliquer à d'autres profils au prix d'adaptation plus ou moins légères. De même ce qui est spécifique WPF peut dans telle ou telle autre version du XAML devenir disponible au fur et à mesure des améliorations portées.

WPF et le focus d'entrée

Avec Windows 8 on sait que WPF sera le seul moyen de développer des applications échappant au market place, bien designées, hors sandbox et surtout capables de tourner sur 90% du parc Windows. Une nouvelle jeunesse s'ouvre donc pour cette techno vraisemblablement. Comment gérer le focus d'entrée dans une appli ? Voici un b.a.ba pas toujours bien maitrisé !

A l'ancienne

On voit souvent dans les boites de dialogue et les fenêtres d'une application WPF du code hérité de Windows Form qui ressemble à cela :

```
1: private void OnLoaded(object sender, EventArgs e)
2: {
3:     this.MytextBox.Focus();
4: }
```

Code 57 - Prise de Focus par code

Ce qui s'accompagne en général du code Xaml suivant dans la fenêtre en question :

```
1: <Window ...
2:         Loaded="OnLoaded">
```

Code 58 - Ecoute de "OnLoaded"

Cela fonctionne, mais franchement c'est se donner du mal pour pas grand-chose et c'est surtout répartir une même fonction logique entre code behind et code UI ce qui est un bon début pour se lancer dans la fabrication de code spaghetti !

WPF offre des moyens bien plus élégants pour réaliser cette opération.

A la page

L'expression "être à la page" n'est plus tellement "à la page", mais puisqu'on parle de page écran... WPF propose donc un moyen plus moderne pour affecter le focus d'entrée d'un dialogue. Par "moderne" je n'entends pas ce faux modernisme qui consiste à faire "autrement" uniquement parce que c'est plus récent. Je parle de moderniste dans le sens de "progrès" et "d'élégance".

Ici notamment il est possible d'assurer cette opération purement d'UI uniquement côté UI sans une seule ligne de code behind.

Pour cela on utilise le `FocusManager`. Son nom se passe de commentaire...

Cela donne :

```
1: <Window ...
2:         FocusManager.FocusedElement=
                        "{Binding ElementName=MyTextBox}">
```

XAML 98 - Utilisation du FocusManager

On remarquera que tout tient dans une ligne, en Xaml, et que le lien avec le contrôle qui doit prendre le focus s'effectue via un "element binding".

C'est clair, propre, localisé, ça concerne l'UI et c'est défini dans l'UI.

Bref, une raison d'aller revisiter votre code pour simplifier (ou enfin gérer...) le focus d'entrée de vos dialogues WPF !

Modern UI pour WPF

La cohérence est l'une des premières choses à considérer en matière de design. Aujourd'hui on peut être amené à développer des applications Modern UI sous

Windows 8 autant que des applications WPF en bureau classique. Assurer une homogénéité de look & feel est donc important. Voici comment y arriver simplement.

Figure 117 - Exemple de contrôles Metro / Modern UI

mahapps.metro

mahapps.metro est une librairie de code qui facilite grandement l'adoption du look "Metro" sous WPF. Du style de la fenêtre principal, ses ajouts de commandes dans la barre de commande, à la mise en forme des boutons, rectangulaires ou ronds, l'essentiel y est.

Le designer c'est vous !

Ce kit ne fera pas la mise en page pour vous, en revanche il vous offre des outils de base qui simplifient grandement l'adoption d'un look Modern UI sous WPF.

Des styles et des contrôles

Le kit ne propose pas seulement des styles, mais aussi des contrôles qui permettent de se rapprocher de l'expérience utilisateur de Modern UI. Par exemple l'anneau de

progression typique de Windows 8, le contrôle Panorama qui permet de simuler le menu Windows 8 ou le contrôle tabulation qui affiche en haut le nom des pages et qui anime les changements tout seul, se rapprochent beaucoup de l'interface Windows Phone / Zune / Windows 8.

Où trouver mahapps.metro

La documentation se trouve ici : http://mahapps.com/MahApps.Metro/ n'hésitez pas à regarder, toutes les explications sont accompagnées de captures qui permettent de se rendre compte du résultat.

Le projet Git-Hub est ici : https://github.com/MahApps/MahApps.Metro

Mieux, il y a package Nuget installable directement dans l'application en cours via le gestionnaire de package de Visual Studio.

Conclusion

Simple, efficace. Pas trop compliqué à utiliser (la documentation donne toutes les indications), personnalisable, bref un petit kit intelligent qui permettra de mettre un peu de fraîcheur Modern UI dans vos applications WPF, et surtout de leur garantir une unité de look & feel avec Windows 8.

Un menu gratuit à Tuile pour WPF ou comment se donner un air Windows 8 en bureau classique…

Un menu de type Tuile ça peut changer beaucoup de choses dans une application WPF (ou Silverlight), cela rend le logiciel plus facile à utiliser même sous Windows 7 en tactile, et puis c'est un petit vent de fraicheur qui permet de se passer des menus traditionnels. Gratuit ? Oui. Et avec le source. C'est un cadeau Dot.Blog…

Un menu à tuile

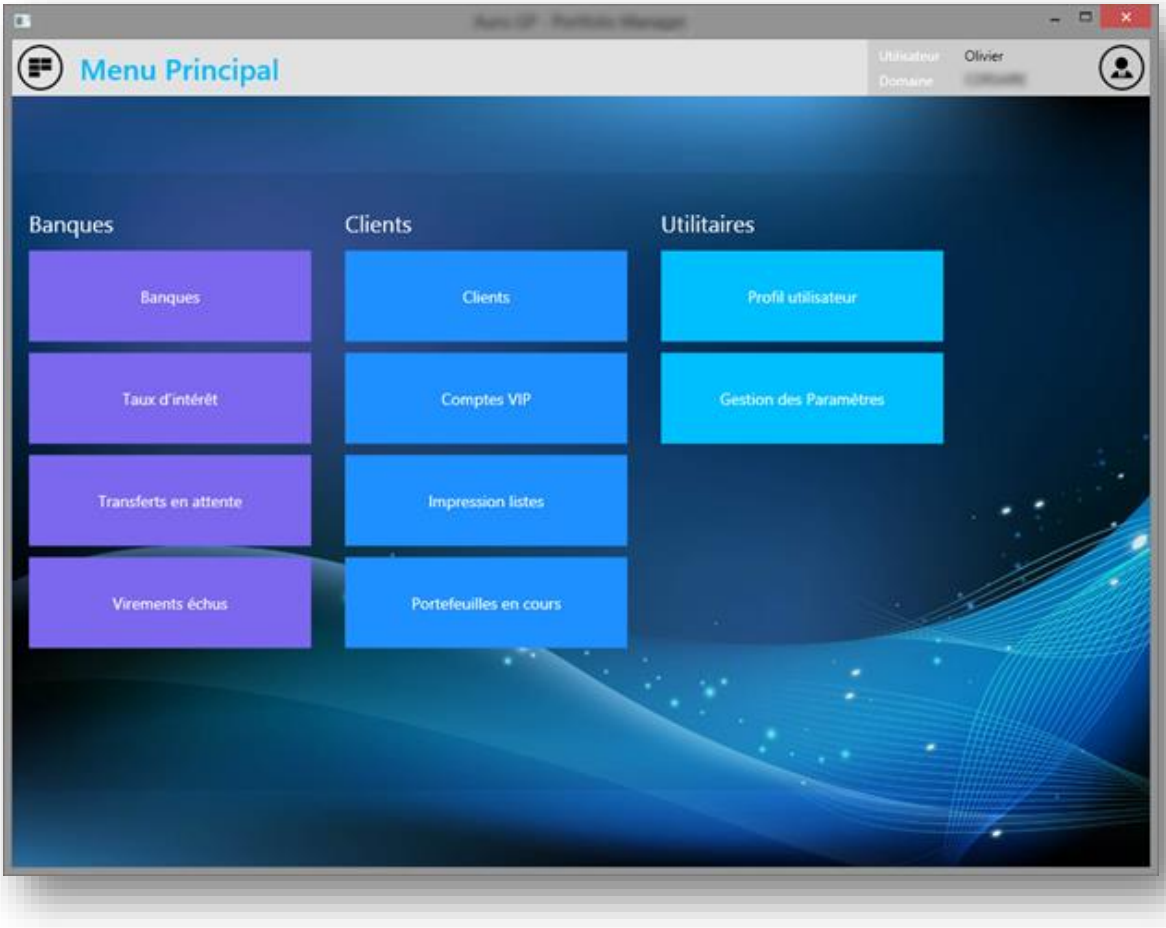


Figure 118 - Exemple du menu à tuile pour XAML

Le menu c'est tout ce qui est dans la zone ayant l'image vectorielle en fond. Le bandeau supérieur fait partie de la mise en page du logiciel.

Comme on le voit : il y a des tuiles de couleurs différentes, des groupes qui portent un nom, et une image de fond. La zone des tuiles est légèrement plus foncée ce qui dessine deux bandes en haut et en bas de cette zone qui délimitent le menu de façon subtile tout en laissant circuler les couleurs.

Sur cette capture on ne le voit pas, mais les groupes peuvent contenir plusieurs lignes de tuiles bien entendu. De même le nombre de groupes n'est pas limité, et si le menu dépasse vers la droite, comme sous Windows 8 on peut facilement scroller avec la molette de la souris. Un parti pris : ne pas afficher l'ascenseur horizontal, question de goût (mais avec le code gratuit de ce contrôle il est facile de faire réapparaitre l'ascenseur si vous le désirez).

Paramétrable et personnalisable

Le but du jeu était que ce menu soit paramétrable facilement. Il l'est.

Paramétrer c'est bien, personnalisé c'est encore mieux… Par exemple on pourrait travailler un peu plus le template de la classe Button utilisé pour faire les tuiles afin de donner un petit mouvement de rotation, ou un effet de lumière sur le mouse over ou plein d'autres idées.

Ici je ne suis pas allé très loin niveau design, ce menu est une juste une base assez sophistiquée pour être utilisable tel quel, mais assez ouverte pour laisser la porte ouverte à plein d'ajouts personnels (des icônes pour les tuiles, des tuiles vivantes pourquoi pas ! – ce ne serait pas très difficile à ajouter en fait).

La construction du Menu

De l'extérieur c'est simple : C'est un `UserControl` agrémenté de propriétés de dépendance.

Il suffit de le poser sur n'importe quelle surface (`Window`, `Page`, autre `UserControl`, etc) et de lui fournir la liste des tuiles à afficher, il se charge du reste. On n'oublie pas de capturer l'évènement de `click` sur les tuiles (un event global bien entendu qui passe toutes les infos nécessaires).

La mise en page de base du control est simple aussi : une grille séparée en 10% – 80% –10% pour créer l'effet de bandeaux haut et bas, une image en fond, un scrollviewer dans la partie centrale avec un stackpanel dedans…

Tout se joue dans le code qui fabrique le menu à la volée.

Chaque groupe est ainsi une grille contenant deux zones, celle du titre du groupe et une zone plus étendue avec un `WrapPanel` dans lequel sont ajoutées les tuiles, de simples boutons templatés donc.

Les difficultés

En fait il y en a une seule. Lorsque le control reçoit la liste des entrées de menu, par exemple par un binding depuis un ViewModel, les vues ne sont pas encore affichées. Toutes les mesures de type `ActualHeight`, `Width` etc, sont à zéro ou à `NaN`.

Pourtant les groupes utilisant un `WrapPanel`, il faut bien les contraindre en hauteur et en largeur sinon rien ne marche.

Il y a donc une astuce pour l'initialisation un peu trop précoce : le `Loaded` du control est géré pour marquer un flag. La séquence de construction du menu vérifie que ce flag est à true. Dans les situations décrites plus haut il ne l'est pas… Que faire ? On marque alors un second flag de type "à redessiner au plus vite!", et on attend… Quand `Loaded` se déclenche on peut positionner à vrai le premier flag et on vérifie qu'il n'y a pas un affichage "en attente", dans l'affirmative on lance enfin la séquence de création des groupes et des tuiles.

C'est un peu "sportif" comme montage, mais cela évite les inévitables plantages en situation réelle lorsque le menu est alimenté par un Binding venant d'un VM. Une fois le control affiché, tout changement qui interviendrait ne pose plus de problème, les composants ont tous des mesures correctes.

C'est le cas pour l'événement de resize qui reconstruit les groupes en fonction de la hauteur et de la largeur disponible.

Les entrées de menu

Je ne visais pas la perfection absolue mais en revanche l'utilisabilité la meilleure. Ainsi j'ai dû faire un compromis entre pureté et isolation du menu avec les besoins de l'application et réalisme…

Les menus sont construits comme une `List<MenuEntry>`, c'est à l'application de fabriquer cette liste et la fournir au menu.

Lorsqu'une tuile est cliquée, l'événement global retourne l'objet `MenuEntry` concerné.

Il fallait que les informations contenues servent à créer le menu mais aussi qu'elle puisse permettre à l'application de prendre des décisions sur le module à afficher.

La classe `MenuEntry` contient ainsi des informations propres à la conception du menu à tuile (nom de groupe, texte affiché, couleur de la tuile, taille de la fonte, nom de la commande à exécuter…) mais en plus elle stocke des informations utiles à l'application comme `IsVisible` ou `IsEnabled` qui permettent en fonction des droits de l'utilisateur de déterminer si une tuile est visible ou non, ou si elle est `Enabled` ou non. De même `MenuEntry` possède un `Tag` qui permet à l'application d'y stocker tout ce qui pourrait être nécessaire au lancement de la vue sélectionnée (une sorte de contexte donc), le tout complété par un marqueur `IsScreenReadOnly` qui permet de savoir si l'utilisateur à des droits complets (lecture/écriture) sur la vue ou seulement

des droits en lecture (ce que la VM de la vue doit être capable de gérer, le menu ne faisant que stocker cette info).

Bien entendu tout cela s'entend dans le contexte réel d'une application et beaucoup de code en dehors du menu s'occupe de lire les droits de l'utilisateur connecté, de fabriquer la liste des tuiles avec leurs paramètres puis de les interpréter lorsque la tuile est cliquée.

On peut fort bien utiliser le menu en faisant abstraction de ces propriétés spéciales.

Les tuiles sont donc groupées. Pour cela elles utilisent le nom de groupe que chaque entrée possède. Je conseille d'utiliser des constantes statiques pour ne pas risquer une distorsion de casse ou une faute qui créerait un groupe différent là où il n'en n'existe qu'un seul.

De même les tuiles sont rangées par ordre alphabétique. Mais chaque entrée possède un champ `Order` (par défaut à 1000) qu'il suffit de modifier pour modifier l'ordre d'affichage (en dessous de 1000 pour qu'il soit dans les premiers de la liste, au-dessus de 1000 pour l'envoyer en fin liste, avec tous les dégradés possibles).

Le reste, c'est dans le code :-)

Le code est documenté, je ne vais pas tout vous raconter, on ne prend possession d'un code que lorsqu'on fait l'effort de lire !

Alors je serai bref, en fin de billet vous trouverez un zip du projet menu !

Conclusion

Il s'agit d'un petit artifice qui peut rendre une application WPF plus jolie, plus moderne. Ce n'est aussi qu'une version 1.0 qui peut s'agrémenter de templates plus jolis ou d'autres fonctions. Mais l'essentiel est là, et ça fonctionne.

Simple à prendre ne main, documenté, c'est le genre de code que j'aime récupérer pour le bricoler. J'espère qu'il vous amusera autant qu'il m'aurait plu de pouvoir récupéré un équivalent !

N'hésitez à poster vos remarques, vos propositions, les bugs aussi que peut-être vous découvrirez, et peut-être même que ce projet deviendra un jour un projet CodePlex, le principe de ce petit menu à tuile m'amuse beaucoup. Qui se dévoue pour une version Silverlight ? (peu de chose à changer, voire rien à mon avis).

Enaxos Tile Menu Source (VS2012) Tile Menu

Des transitions de page à la Windows 8 pour WPF (et Silverlight)

Les menus à tuiles c'est joli mais pour donner encore plus de fraicheur Windows 8 à vos applications WPF, rien ne vaut une transition de page discrète à la Modern UI…

Ah Modern UI !

On dira ce qu'on voudra, mais cet OS est plein de bonnes idées. Je l'adore. Toutes mes machines en sont équipées, des portables aux machines neuves en passant par les anciennes qui ne mettent plus 10 minutes à booter et retrouvent ainsi une rieuse jeunesse. Sans parler de ma Surface.

> *Pour que les choses soit vraiment claires et que mon intégrité d'expert indépendant (c'est le statut d'un MVP) reste crédible, je ne serai pas totalement franc si je ne vous disais pas que sur PC j'ai bien remarqué quelques problèmes, je ne suis pas aveugle… il y a des choses qui me gênent vraiment.*
> *Par exemple il va falloir que Microsoft rendre le "S" à Windows, car du full-screen sur des PC suréquipés avec écrans multiples c'est tout simplement digne des lapins crétins. Le bureau de Windows doit être unifié, un seul bureau, pas deux, et si WinRT veut se faire une place sur les PC il devra apprendre à faire vivre ses applications dans des fenêtres. A moins que Microsoft ne choisissent définitivement l'option de la scission comme cela semble se profiler : un Windows "grand public" et un autre pour les professionnels… Bien évidemment cela éviterait d'avouer que pour les PC il y a eu une erreur d'approche, cela éviterait aussi l'impression étrange d'un OS à deux têtes incompatibles entre elles, certes. Mais j'aime la concorde et l'harmonie, et je préfèrerais franchement qu'après avoir unifié les form factors, Windows s'unifie lui-même !*

Alors si vous n'avez pas décidé de vous lancer uniquement sur les tablettes et les smartphones, si pour vous le marché du PC reste incontournable, vous savez qu'il ne reste plus que WPF comme choix raisonnable pour développer… De XP à Windows 8, seul WPF offre une compatibilité qui évite d'avoir à mettre ses œufs dans le même

panier et une modernité que seule Xaml peut apporter. En tant que développeur c'est un choix évident. En tant qu'éditeur de logiciel il faut aussi être présent sur WinRT même sur PC j'en suis convaincu – question d'image - mais pour vendre il faut un équivalent en WPF…

C'est pourquoi je vous ai proposé dernièrement un menu à tuile pour WPF (voir le lien en tête de billet) et que je vous propose aujourd'hui des transitions de page automatiques à la Windows 8. WPF permet de tout faire, sur de nombreuses plateformes, et avec un peu d'astuce on peut même développer pour WPF et WinRT avec un code partagé (notamment avec MvvmCross traité dans les 12 vidéos publiés sur YouTube cet été et le Tome 5 de All Dot Blog consacré au cross-plateforme). WPF est une solution qui répond à tous les besoins : la compatibilité avec les OS en place, le besoin de modernité, la qualité de l'UX et de l'UI, l'exploitation des compétences déjà acquises en entreprise avec .NET.

Un conteneur Animé

L'idée est simple, MS l'a fait, d'autres ont tenté le coup en faisant des contrôles plus ou moins bien réalisés, mais l'idée est de Microsoft.

WPF fonctionne en bureau classique Windows 8 et marche aussi sur Windows 7 et XP, tout de suite cela lui donne un intérêt particulier, surtout si on sait l'habiller un peu Metro…

Et le control Animé ?

Je vais vous donner le code source, vous pourrez jouer avec autant que vous voulez. Ne soyez pas si impatients !

Le principe

Le control est du type `ContentControl`. Il expose une propriété `Content`, et on y met ce qu'on veut pour changer l'affichage. Par exemple, dans une application MVVM c'est le *Shell* qui se sert de ce conteneur pour afficher les vues (personnellement j'utilise des `UserControl` pour les Views).

Seule différence avec le conteneur standard : il prend un cliché de l'affichage actuel, et génère des animations pour ce dernier et le nouveau contenu de telle façon à créer une animation souple, rapide et agréable (et sans memory leaks, du moins j'ai fait très attention à ce point).

L'effet se joue à peu de choses, quelques pourcents par ici, le choix de la fonction de easing, la durée des timelines, les valeurs initiales, le point de départ de l'animation sur l'axe des X, le petit coup de fading, etc.

Je suis arrivé à un réglage qui me plait. A vous de le personnaliser selon vos gouts !

Le cadeau

C'est le code, gratuit, que demander de mieux…

Enaxos Animation (VS 2012) Animated Content

A noter que le code devrait tourner sous Silverlight.

Conclusion

Un petit pas pour le développeur, un grand pas, peut-être, pour redonner gout au développement WPF. Eclipsé par Silverlight pendant un temps, il reste la seule alternative crédible pour faire des applications modernes qui marchent sur toutes les versions de Windows récentes. C'est une technologie sure, éprouvée, sans mystère, puissante, conforme à toutes les guidelines modernes et qui tourne sur tous les OS récents.

Développez pour Windows 8 en WinRT, c'est génial et c'est certainement l'avenir. Je ne veux pas vous en détourner.

Mais si vous voulez manger, pour l'instant en tout cas, n'oubliez pas que WPF est lui aussi vraiment génial … et qu'il tourne sur Windows 7 que les entreprises commencent seulement à adopter ainsi que sur les 34% d'XP toujours en fonction…

Avertissements

L'ensemble de textes proposés ici est issu du blog « Dot.Blog » écrit par Olivier Dahan et produit par la société E-Naxos.

Les billets ont été collectés en septembre 2013 pour les regrouper par thème et les transformer en document PDF cela pour en rendre ainsi l'accès plus facile. Les mois d'octobre et novembre ont été consacrés à la remise en forme, à la relecture, parfois à des corrections ou ajouts importants comme le livre PDF sur le cross-plateforme par exemple (TOME 5).

Les textes originaux ont été écrits entre 2007 et 2013, six longues années de présence de Dot.Blog sur le Web, lui-même suivant ses illustres prédécesseurs comme le Delphi Stargate qui était dédié au langage Delphi dans les années 90/2000.

Ce recueil peut parfois poser le problème de parler au futur de choses qui appartiennent au passé… Mais l'exactitude technique et l'à propos des informations véhiculées par tous ces billets n'a pas de temps, tant que les technologies évoquées existeront …

Le lecteur excusera ces anachronismes de surface et prendra plaisir j'en suis certain à se concentrer sur le fond.

E-Naxos

E-Naxos est au départ une société éditrice de logiciels fondée par Olivier Dahan en 2001. Héritière de *Object Based System* et de *E.D.I.G.* créées plus tôt (1984 pour cette dernière) elle s'est d'abord consacrée à l'édition de logiciels tels que la suite Hippocrate (gestion de cabinet médical et de cabinet de radiologie) puis d'autres produits comme par exemple MK Query Builder (requêteur visuel SQL).

Peu de temps après sa création E-Naxos s'est orientée vers le Conseil et l'Audit puis s'est ouverte à la Formation et au Développement au forfait. Faisant bénéficier ses clients de sa longue expérience dans la conception de logiciels robustes, de la relation client, de la connaissance des utilisateurs et de l'art, car finalement c'en est un, de concevoir des logiciels à la pointe mais maintenables dans le temps.

C#, Xaml ont été les piliers de cette nouvelle direction et Olivier a été récompensé par Microsoft pour son travail au sein de la communauté des développeurs WPF et Silverlight. Toutefois sa première distinction a été d'être nommé MVP C#. On ne construit pas de beaux logiciels sans bien connaître le langage…

Aujourd'hui E-Naxos continue à proposer ses services de Conseil, Audit, Formation et Développement, toutes ces activités étant centrées autour des outils et langages Microsoft, de WPF à WinRT (Windows Store) en passant par Silverlight et Windows Phone.

A l'écoute du marché et offrant toujours un conseil éclairé à ses client, E-Naxos s'est aussi spécialisée dans le développement Cross-Plateforme, notamment dans le mariage des OS Microsoft avec Android, les deux incontournables du marché d'aujourd'hui et de demain.

N'hésitez pas à faire appel à E-Naxos, la compétence et l'expérience sont des denrées rares !