www.e-naxos.com

Formation – Audit – Conseil – Développement

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

Cross-plateforme Windows / Android / iOS

UX Design

ALL DOT.BLOG

TOME 2

Méthodes & Frameworks MVVM

Tout Dot.Blog par thème sous la forme de livres PDF gratuits !

Reproduction, utilisation et diffusion interdites sans l'autorisation de l'auteur

Olivier Dahan

odahan@gmail.com

Table des matières

Métaphysique de la Méthodologie ... 9
La Méthodologie ... 9
Se poser les bonnes questions ... 9
Avoir des habitudes est une mauvaise habitude ... 9
Le cul des chevaux romains et la Nasa ... 10
La fréquence d'échantillonnage des CD musicaux ... 12
Conclusion ... 13
Le retour du spaghetti vengeur ... 13
Genèse d'un malaise ... 14
Espèce de spaghetti ! ... 15
Le code qui rend fou ... 17
Le Binding au pays des X-Files ... 18
Vision d'horreur ... 19
Le salaire de la peur ... 20
David Vincent au pays des tags ... 21
O Tempora, O Mores ! ... 21
Faut-il brûler Xaml ? ... 22
Il faut sauver le soldat Xaml ! ... 22
Un peu de sérieux… ... 23
Patterns & Practices : les guides de bonnes pratiques à connaître par coeur ! ... 23
Patterns & Practices ... 24
Les projets clé ... 24
App Arch Guide 2.0 Knowledge base ... 24
Application Architecture Guide (le livre) ... 25
Enterprise Library ... 25
Web Client Software Factory ... 25
Composite WPF and Silverlight ... 26
Smart Client Guidance ... 26
Unity ... 26
WCF Security Guidance ... 26
Web Service Software Factory ... 26

Guidance Explorer 27
MS Health Common User Interface 27
Conclusion 27
Prism v4 (guidance and patterns for WPF and Silverlight) 28
PEX : Tests unitaires intelligents pour Visual Studio 29
MEF - Managed Extensibility Framework - De la magie est des plugins ! 30
Une gestion de plugin simplifiée 30
MEF et les autres 30
Comment choisir ? 30
MEF – Le principe 31
MEF – Les avantages 31
Un Exemple ! Un Exemple ! 31
Conclusion 37
MVVM, Unity, Prism, Inversion of Control… 37
MVVM ou plutôt M-V-VM 37
Prism 40
Unity Application Block 41
Conclusion 44
MVVMCross, Okra... Mvvm, WinRT et cross-plateforme Android/Apple 45
Okra 45
Prism for Windows Store Apps 45
MVVMCross 45
MonoTouch et MonoDroid 46
MVVMCross : développez une fois pour toutes les plateformes 46
Conclusion 47
MVVM, je m'y perds ! (une réponse rapide) 48
MVVM ça change quoi au final ? 48
Le découpage à la MVVM 49
Les complications… 50
La barrière de la communication 50
Et le reste 51
Conclusion 51

MVVM est-il un vrai pattern ? 51
Masochisme ? 51
Qu'ce qu'un pattern ? 52
Les conséquences font le pattern 53
MVVM peut-elle être considérée comme un design pattern ? 54
Quelles conséquences à ce changement de statut ? 55
Et alors ? 56
Conclusion 56
Simple MVVM 57
Silverlight MVVM : les commandes 58
Le but 58
Structure simplifiée d'une application MVVM 58
Informations mais aussi commandes 59
Un exemple live 60
L'interface ICommand 61
Le Circuit d'une commande 62
Feedback des commandes 65
MVVM Light 67
L'initialisation des commandes 67
Le CommandManager manquant 69
Article: M-V-VM avec Silverlight 70
M-V-VM et Silverlight – De la théorie à la pratique [avec Silverlight 4, VS 2010 et Blend 4] 71
Références 72
Code Source 72
Préambule 73
Introduction à M-V-VM 73
Utiliser M-V-VM 87
Mise en œuvre 100
Conclusion 147
M-V-VM appliqué avec M-V-VM Light [WPF et Silverlight] 148
Table des figures 152

Code Source 153
Préambule 154
Pourquoi M-V-VM Light ? 154
Une documentation non-officielle ? 155
Se procurer M-V-VM Light 156
Visual Studio et Blend 157
Bref rappels sur M-V-VM 158
La librairie MVVM Light 160
Le pattern MVVM et MVVM Light 170
MVVM Light en pratique 187
Conclusion 255
Deux extensions gratuites pour MVVM Light 256
MVVM Light 256
Deux besoins non comblés “out of the box” 256
Conclusion 262
MVVM : simplifier le circuit des messages 263
MVVM et la messagerie 263
Le circuit du message de base 264
Le circuit d’un message avec réponse 265
Raccourcir le circuit ? 266
Une question de point de vue 267
Deux options 267
Faciliter le dialogue 269
Simplifier encore ? 269
L’approche de l’Injection de Dépendance 270
Conclusion 271
MVVM : Gérer les données de Design de façon propre 271
MVVM “Générique” 271
Silverlight 272
Principes de base des données de design 272
View, ViewModel et Model 273
Création d’un switch de compilation ou pas ? 278

L'ajout du code de génération des données de design 278
Le design 282
Conclusion 285
MVVM, Chronomètre et Illustrator 285
Voir le projet finalisé 285
MVVM Light 285
Expression Blend 285
L'objet graphique du projet 286
Création du projet 286
L'importation du graphique 287
Faire marcher le chronomètre 289
Le code 299
Conclusion 303
Le Livre Blanc de JOUNCE / MVVM et MEF avec Silverlight 4+ [Applications modulaires suivant MVVM] 304
Sommaire 305
Table des Figures et des Tableaux 308
Code Source 309
Préambule 310
MEF 310
MVVM 311
Jounce vs MVVM Light 311
Que propose Jounce ? 316
La Librairie 318
Les bases 327
BaseEntityViewModel 357
Les Commandes 373
La messagerie EventAggregator 392
Navigation simplifiée : Event Aggregator et ViewRouter 399
Les régions 404
Chargement de XAP 408
Logger personnalisé 411

Workflows ... 417
VSM Aggregator et GotoVisualState ... 424
Conclusion ... 435
Prism pour WinRT– Partie 1 ... 437
Prism or not Prism ? ... 437
WinRT pour le LOB ? ... 437
Dans ce contexte quelle est la place et l'intérêt de Prism ? ... 440
Prism pour WinRT ... 441
Conclusion ... 442
Prism pour WinRT– Partie 2 – Premier exemple de code ... 443
Prism pour le Windows Runtime ... 443
A quoi sert Prism ? ... 443
Créer une application WinRT avec Prism ... 445
Conclusion ... 454
Prism pour WinRT– Partie 3 – Navigation et Commandes ... 456
Navigation et Commandes ... 456
Naviguer depuis la MainPage ... 456
Utiliser un Behavior attaché pour gérer l'ItemClick ... 458
Gérer les commandes de navigation ... 460
Naviguer depuis une AppBar ... 467
Transmettre des informations entre ViewModels ... 467
Conclusion ... 469
Prism pour WinRT – Partie 4 – Gestion des états de l'application ... 469
Le cycle de vie des applications sous WinRT ... 469
Prism dans tout ça ? ... 472
L'application exemple ... 472
Ajouter des données ... 472
Exposer les données pour la Vue ... 474
Quelque chose qui cloche… ... 477
Propriétés auto-sauvegardées ... 477
ISessionStateService ... 479
Conclusion ... 482

Prism pour WinRT – Partie 5 – Communiquer sans se connaitre (L'EventAggregator) 482
Dans les épisodes précédents… 483
L'ombre de Prism 4 483
Messagerie ou agrégateur d'évènements ? 483
L'Event Aggregator 484
L'EventAggregator en action 485
S'abonner en tâche de fond 492
Threads et EA 493
Filtrage 494
Conclusion 494
Prism pour WinRT–Adresses utiles 495
Prism pour WinRT – source et documentation à jour 495
Prism pour WinRT : une application exemple 496
Itinerary Hunter, le chasseur d'itinéraire 496
Une bonne utilisation des guidances 497
Le code source 497
Quelques copies d'écran 498
Conclusion 500

Métaphysique de la Méthodologie

Pourquoi diable le format par défaut d'un CD audio est-il de 44100 échantillons par seconde ? La raison se trouve dans le poids des habitudes... Comme la taille des réservoirs de la navette spatiale américaine. Tout cela impose une petite réflexion sur la Méthodologie, avec humour...

LA METHODOLOGIE

Quand je parle ici de Méthodologie, je parle de l'ensemble des méthodes et procédures qui sont suivies dans un métier ou un autre pour arriver à de "bons" résultats. J'englobe ainsi largement tout ce qui est "norme", "normalisation" autant que les Méthodologies de développement telles qu'on les utilise dans notre beau métier (RUP, Agile, Extreme Programming, Design patterns divers et variés....).

SE POSER LES BONNES QUESTIONS

Lorsque vous êtes face à un problème à résoudre il convient toujours d'essayer d'en extraire des patterns connus et d'utiliser des solutions éprouvées aux conséquences connues et prévisibles. C'est tout l'art de l'utilisation des Design Patterns notamment.

De même certains prônent telle ou telle méthode de développement comme étant la seule voie logique à suivre pour construire un logiciel fiable tout en respectant les délais impartis, en maitrisant les couts de production le tout en minimisant les couts de maintenance (ouf!) ...

De RUP à Extreme Programming, tout le monde chante la même chanson. On nous a tellement gavé d'UML et autres méthodes miracles (UML n'étant pas une méthode d'ailleurs mais une simple norme) que le sujet est passé de mode... Trop de méthode tue la méthode.

Reste donc l'informaticien face à son logiciel à écrire. Comme avant. Avant l'invention de toutes ses méthodologies. Sinistre pied de nez à l'histoire d'une science pourtant très jeune.

Reste donc à se poser les bonnes questions. Et à trouver les bonnes réponses...

AVOIR DES HABITUDES EST UNE MAUVAISE HABITUDE

C'est comme les manies, les tics, les tocs : avoir des habitudes est une mauvaise habitude.

Savoir se renouveler est essentiel.

Mais je ne suis pas venu aujourd'hui pour blogger des heures en vous faisant la morale ou en prenant un ton doctoral pour vous insufflez l'Art du Questionnement Métaphysique. Il y a ceux qui ont écouté et compris leurs cours de philo à l'école, et ceux qui ont toujours cru que c'était une heure de permanence et qui n'ont jamais compris pourquoi il y avait en plus un

animateur spécialisé qui disait des trucs bizarres ce qui empêchait de finir de réviser ses maths tranquillement... A ceux-là je ne peux rien dire de plus, et les premiers n'ont pas besoin de mes conseils.

Je vais donc plutôt vous raconter deux histoires.

Vous connaissez peut-être la première car elle a circulé presque sous forme de spam sur le Net à une époque lointaine.

Vous serez peut-être plus surpris par la seconde qui tend à rendre totalement crédible la première.

LE CUL DES CHEVAUX ROMAINS ET LA NASA

J'ai connu une variante de cette histoire avec des bœufs. Celle que je vais vous donner ici parle de chevaux. Je pense que les deux sont équiprobables. Au départ on pense à une grosse plaisanterie, un truc drôle inventé par un type qui s'ennuyait un jour. Un des premiers spam du Net selon mes souvenirs.

Et puis on y réfléchit. On regarde autour de soi, on s'interroge. Et on se dit que même si c'est faux, cela fait du bien de poser le problème. Mais quand vous aurez lu la seconde histoire, dont la véracité est elle totalement démontrée, vous ne rirez plus. Et peut-être saurez-vous vous poser les bonnes questions et remettre certains acquis en question. Vous repenserez alors à ces deux histoires, et peut-être un peu à Dot.Blog et à votre serviteur !

Le cul des chevaux romains

La distance standard entre 2 rails de chemin de fer aux USA est de 4 pieds et 8,5 pouces. C'est un chiffre particulièrement bizarre. Pourquoi cet écartement a-t-il été retenu ?

Parce que les chemins de fer US ont été construits de la même façon qu'en Angleterre, par des ingénieurs anglais expatriés, qui ont pensé que c'était une bonne idée car cela permettait également d'utiliser des locomotives anglaises.

Pourquoi les anglais ont construits les leurs comme cela ?

Parce que les premières lignes de chemin de fer furent construites par les mêmes ingénieurs qui construisirent les tramways, et que cet écartement était alors utilisé. Pourquoi ont-ils utilisé cet écartement ?

Parce que les personnes qui construisaient les tramways étaient les mêmes qui construisaient les charriots et qu'ils ont utilisé les mêmes méthodes et les mêmes outils. Pourquoi les charriots utilisaient-ils un tel écartement ?

Parce que partout en Europe et en Angleterre, les routes avaient déjà des ornières et un espacement différent aurait causé la rupture de l'essieu du charriot.
Pourquoi ces routes présentaient-elles des ornières ainsi espacées ?
Parce qu'elles dataient du temps des romains et furent construites par l'empire romain pour accélérer le déploiement des légions romaines.
Pourquoi les romains ont-ils retenu cette dimension ?
Parce que les premiers charriots étaient des charriots de guerre romains. Ces charriots étaient tirés par deux chevaux. Ces chevaux galopaient côte à côte et devaient être espacés suffisamment pour ne pas se gêner. Afin d'assurer une meilleure stabilité du charriot, les roues ne devaient pas se trouver dans la continuité des empreintes de sabots laissées par les chevaux, et ne pas se trouver trop espacées pour ne pas causer d'accidents lors du croisement de deux charriots.

[Variante : il s'agit des bœufs des charriots romains tirant de lourdes charges. Ces charriots ont créé des ornières. Vache ou cheval, on peut le voir encore aujourd'hui par exemple en suivant les vestiges de la "Voie Romaine" en France. Les pavés sont nettement enfoncés de part et d'autre de la voie créant deux sillons. Faire rouler un charriot aux dimensions différentes devait y être impossible, un peu comme demander à une femme de marcher rapidement en haut-talons sur une chaussée en pierre comme on en trouve dans le vieux Paris ou d'autres villes ayant un centre historique...]

Nous avons donc maintenant la réponse à notre question d'origine.
L'espacement des rails aux USA (4 pieds et 8 pouces et demi) s'explique parce que plus de 2000 ans auparavant, sur un autre continent, les charriots romains étaient construits en fonction de la dimension du cul des chevaux de guerre [ou de celui des bœufs]. ...

Et maintenant, la cerise sur le gâteau.
Il y a une extension intéressante de cette histoire concernant l'espacement des rails et l'arrière train des chevaux.
Quand nous la navette spatiale américaine sur son pas de tir, nous pouvons remarquer les deux réservoirs additionnels attachés au réservoir principal [la navette vient de faire son dernier vol, bientôt personne ne comprendra plus cette histoire, elle est d'ailleurs au musée maintenant que je remets en page ce PDF]. C'est la société THIOKOL qui fabriquait ces réservoirs additionnels dans leur usine de l'UTAH.

Les ingénieurs qui les ont conçus auraient bien aimé les faire un peu plus larges, mais ces réservoirs devaient être expédiés par train jusqu'au site de lancement. La ligne de chemin de fer entre l'usine et Cap Canaveral emprunte un tunnel sous les montagnes rocheuses. Les réservoirs additionnels devaient pouvoir passer sous ce tunnel. Le tunnel est légèrement plus large que la voie de chemin de fer, et la voie de chemin de fer est à peu près aussi large que les arrières trains de deux chevaux romains.

Conclusion :

Le moyen de transport le plus sophistiqué que l'homme n'a jamais conçu et qui permet d'aller dans l'espace a été construit en respectant des contraintes implicites basées sur la taille du cul des chevaux ou des bœufs des charriots de l'Empire Romain !

Le poids des habitudes, ce terrible recommencement des mêmes gestes sans se poser de question existera toujours. Il est même le moteur de ce qu'on appelle la bureaucratie qui envahit toutes nos institutions, mêmes les plus récentes comme les instances européennes qui n'ont pas une vieille histoire à trainer. Mais chaque bureaucrate la constituant a amené avec elle ses propres "culs de chevaux romains", le cul des chevaux romains de bureaucrates de 27 pays différents. Cela laisse songeur...

Aussi, la prochaine fois que vous avez des spécifications entre les mains et que vous vous demandez quel cul de cheval les a inventées, vous vous serez peut-être posé la bonne question...

La fréquence d'échantillonnage des CD musicaux

Ah ! La taille du postérieur des équidés romains ou de leurs bœufs ça sent bon le hoax, on peut en rire et évacuer la chose facilement. Des histoires, des légendes comme le Web sait en créer et en véhiculer.

Si vous suivez Dot.Blog, vous vous rappelez peut-être de mon billet intitulé :
Lequel est le plus foncé sous WPF/Silverlight : Gray ou DarkGray ?
J'y évoquais déjà l'histoire du cul des chevaux (ou des vaches) en démontrant que cette histoire pouvait certainement être vraie en prenant exemple sur la dénomination des couleurs sous WPF. Je vous conseille la lecture de ce billet, cela renforce tout ce qui est dit ici...

Bref, les vaches romaines ça fait folklore. Le nom des constantes de couleur dans WPF déjà ça fait moins rire. Mais vous rirez jaune une fois que j'aurais enfoncé le clou avec cette troisième histoire (la seconde de ce billet) concernant la fréquence d'échantillonnage des CD...

L'anecdote est tirée de Wikipédia, dans le Wiki sur le "Disque Compact".

La fréquence d'échantillonnage des CD

La fréquence d'échantillonnage de 44,1 kHz des CD musicaux est héritée d'une méthode de conversion numérique d'un signal audio en signal vidéo pour un enregistrement sur cassette vidéo qui était le seul support offrant une bande passante suffisante pour enregistrer la quantité de données nécessaire à un enregistrement audionumérique.

Cette technologie peut stocker six échantillons (trois par canal en stéréo) par ligne horizontale. Un signal vidéo américain NTSC possède 245 lignes utilisables par trame et 59,94 champs par seconde qui fonctionnent à 44 056 échantillons par seconde.
De même, un signal vidéo PAL anglais ou SECAM français possède 294 lignes et 50 champs qui permet aussi de délivrer 44 100 échantillons par seconde. Ce système pouvait aussi stocker des échantillons de 14 bits avec des corrections d'erreur ou des échantillons de 16 bits sans correction d'erreur. Il y eut donc un long débat entre Philips et Sony concernant la fréquence et la résolution de l'échantillonnage. Philips voulant utiliser le 44 100 Hz utilisé en Europe et une résolution de 14 bits ayant déjà développé des CNA 14 bits et Sony voulant imposer le 44 056 Hz utilisé au Japon et États-Unis et une résolution de 16 bits.
On connait la suite, la poire a été coupée en deux, la fréquence 44.1KHz Philips et la résolution 16 bits Sony ont été adoptées après d'âpres négociations.

Ainsi, la fréquence d'échantillonnage d'un morceau de musique au 3ème millénaire, ou même la fréquence des convertisseurs "Analogique / Digital" utilisés par les musiciens avec des synthétiseurs software sur des ordinateurs ultra récents, tout cela est directement dicté par la résolution des téléviseurs SECAM / PAL, une technologie d'après guerre vieille de 60 ans au minimum et totalement analogique... Autant dire qu'il y a autant de rapport entre un vieux poste de télévision français et un lecteur MP3 qu'entre... les réservoirs de la navette spatiale et la taille du cul des chevaux romains !

CONCLUSION

Au début on rit, surtout quand on n'a jamais lu cette histoire sur la taille du cul des chevaux romains. En reprenant le billet sur les constantes de couleur de WPF déjà on rit moins. Quand on arrive à la raison de ce chiffre magique de 44100 pour le nombre d'échantillons par seconde d'un CD audio on pense que le monde est véritablement dirigé par les habitudes et la bêtise.

Je parlais de Métaphysique un peu pompeusement dans le titre de ce billet. Pompeux et racoleur, certes. Quoi que ...

Les us et coutumes, même les entreprises d'informatique les plus modernes en trainent des tas dès qu'elles ont plus de deux ans d'existence... Des habitudes déraisonnables mais acceptées par tous nous en trouvons tous les jours, même dans notre vie privée si on y réfléchit quelques instants...

Alors métaphysique, peut-être pas, mais philosophique, ce billet l'est certainement.

Le retour du spaghetti vengeur

Le code spaghetti est de retour ! Fuyez braves gens !

Sous-titré : *Du Rififi dans le Xaml*.

***Avertissement au lecteur** : ce billet, bien que bâti sur un fond technique préoccupant et une expérience réelle, utilise un formalisme un peu romancé. Ne cherchez pas d'extraits de code ici. Mais si vous avez un peu de temps, laissez vous porter par l'histoire. Si vous êtes pressés, revenez plus tard lire ce billet !*

GENESE D'UN MALAISE

Comme vous le savez je suis un passionné de WPF et de Silverlight, la puissance de Xaml servi par un Framework aussi riche que .NET et des outils de qualité comme Blend ou Visual Studio ne peuvent laisser de marbre (comment peut-il y avoir encore des gens travaillant sous Java ou Delphi ?). J'en suis tellement passionné que j'évangélise à tour de bras, ici et ailleurs, et que mes compétences autant techniques que de gardien des brebis égarées me valent d'avoir l'honneur d'être Microsoft MVP depuis plusieurs années (C#, Client App Dev et Silverlight).

Bref, si je dis tout cela c'est pour faire comprendre que bien que cultivant mon objectivité comme un joyau précieux garant de la liberté de mes neurones, je suis plutôt "pro" Microsoft et que, bien entendu, cela peut déplaire à certains comme en séduire d'autres… La diversité du monde en fait sa richesse, isn't it.

Et un partisan de Microsoft, MVP de surcroît, fan de Xaml, ne dira jamais le moindre mal de sa technologie fétiche… Et **soyez-en convaincu je ne briserai pas cette loyauté**, essentielle à

mes yeux, **mais il faut pourtant savoir tirer les sonnettes d'alarme** de temps en temps. D'ailleurs c'est une question de **crédibilité**, que vaudrait un expert sans liberté de parole ni de pensée… Et puis au fond vous verrez que c'est bien plus, comme à chaque fois, **l'humain qui en prend pour son grade** dans ce récit que **les outils, innocents, par nature**.
Tout cela pour parler franchement d'un risque, d'une dérive, et surtout d'un grand danger : **le retour du code spaghetti !** Et, comble de l'infamie, la peur de cette tare qu'on croyait du passé, je ne la brandis pas à propos de langages ésotériques comme F#, ou de solutions vieillissantes comme Java. Non. Le drame est bien là : c'est de WPF et de Silverlight dont je veux vous entretenir, et ce, au travers d'une anecdote récente (au moment de l'écrirture de ce papier en 2010).
J'ai fait il y a quelques temps un audit dont je tairais, vous le comprendrez aisément, le nom du client visité ainsi que la date exacte. Au demeurant une société ni trop jeune pour n'être pas assez structurée, ni trop vieille pour en être devenue ringarde. Une entreprise de bonne taille, assez dynamique et assez typique de ma clientèle, se targuant de posséder une équipe de développement à la pointe du progrès, la preuve, puisque maniant les balises Xaml avec la même dextérité que la truelle l'est par un vrai maçon diplômé.

Le trait n'est pas forcé, il n'y en a nul besoin. Ce fut au début un audit "classique" c'est à dire durant lequel j'ai vu du code "normal" donc assez médiocre. Je dis "normal" au sens de la loi normale statistique ce qui signifie que grâce à messieurs Laplace et Gauss il m'arrive de voir des choses épouvantables comme de pures merveilles, mais que ces deux cas représentent un pourcentage faible aux deux bouts de la cloche… Je ne savais pas encore que j'allais atteindre le bout de la cloche. Le mauvais. Bien entendu.

ESPECE DE SPAGHETTI !

Le code "normal" est médiocre généralement. C'est finalement une définition en soi. Le code exceptionnel, étant, par le même genre de raisonnement, plus rare. C'est finalement une lapalissade. Donc, en général, je vois du code médiocre, donc normal (dans l'autre sens c'est intéressant aussi, non ?).

Et j'appelle médiocre un code qui se caractérise par plusieurs aspects distinctifs très techniques que je détaille avec moult délicatesse dans mes rapports d'audit pour expliquer les choses en y mettant les formes mais qui se résument en général à quelques cas généraux qu'on pourrait caractériser très doctement. Je m'exprimerais ici de façon bien plus directe puisqu'on est "entre nous" :

- La "***putain***" c'est à dire le code sur lequel tout le monde est passé, sauf le train et vous, mais ça, ce n'est même pas sûr, puisque vous êtes là… C'est du code typique des boîtes où les gens sont mal payés et où les salariés défilent plus vite qu'une hirondelle dans un ciel de printemps…
- Les ***migrations de migrations de portage d'intégration*** (de milles sabords, version 24 bis modifiée E). En général du code qu'on trouve dans les

administrations. Avec des documentations de plusieurs centaines de pages, que personne n'a jamais lues bien entendu.

- Le code ***mille-feuille***. Savoureuse pâtisserie constituée de tant de couches qu'on ne peut les compter, comme les pattes du iule (autrement appelé mille-pattes). C'est un peu un mélange des deux précédents mais en plus structuré que le premier (beaucoup plus, et c'est la son problème) et en moins documenté que le second (beaucoup moins, et c'est aussi là son problème). C'est du code de SSII "in" avec de vrais morceaux de "nouvelles technos" dedans.
- Le ***code "start-up"***, celui-là est bourré de technos innovantes, hélas non maîtrisées, peu documentées et en bêta ne tournant que sur les versions US de l'environnement. Un code d'essayiste pur travaillant pour la beauté du discours qui va autour plus que pour l'efficacité, des gens qui devraient être en agence de pub plutôt que devant un environnement de développement.
- Le ***code à papa***, ou l'objet est utilisé comme du C procédural. C'est le code C# écrit par de vieux delphistes ou cobolistes reconvertis sur le tard par exemple.
- Le ***code d'ingénieur***, un des pires. Sortant de l'école et voulant montrer ses muscles en complexifiant tout et l'enrobant dans une prose technique pleine de sigles bizarres et de référence à des bouquins lus de lui seul et de ses potes de promo. Quand il arrêtera de sucer son pouce, il deviendra un bon développeur sévèrement burné comme disait Tapie. Mais en attendant son code c'est l'enfer…
- Et enfin, le célèbre, le magnifique, le ***code spaghetti***, marquant l'incompétence à maîtriser la complexité du sujet. Celui-là est typique des mauvaises équipes, tous contextes confondus.

Il y en a bien d'autres dans mon bestiaire, en 20 ans d'audit vous imaginez ce que j'ai pu voir (heureusement je rencontre aussi des équipes compétences et même parfois de l'excellent code, sinon ça serait désespérant mais c'est moin drôle à raconter !).

Et WPF et Silverlight (comme WinRT et Windows Phone d'ailleurs) dans tout ça ?

C'est là que je voulais en venir, mais il fallait bien passer par ce détour pour vous plonger dans l'ambiance trouble de cette descente aux enfers binaires. Sinon cela aurait été d'une platitude redoutable. Un peu de lecture ça change des extraits de code en Xaml. Justement. Parlons de lui et de ce dernier audit (qui n'est pas le dernier, confidentialité oblige comme je le disais plus haut). Disons que c'est assez récent pour être en Xaml mais pas suffisamment pour être en Silverlight 3.

Qu'avait ce code qui vaille ce billet un peu particulier ?

LE CODE QUI REND FOU

Il m'a rendu fou. Tout simplement ! Et en plus il m'a filé les chocottes !

WPF et Silverlight sont des technologies merveilleuses, Xaml a un pouvoir descriptif exceptionnel à tel point qu'il permet d'économiser beaucoup de code behind.

Malgré le génie des équipes MS ayant travaillé sur le Framework j'aurais malgré tout préféré que cette révolution se fasse à 100% dans le respect du paradigme objet et du fortement typé. Or ce ne fut pas totalement le cas, et si je comprends bien les contraintes techniques sous-jacentes qui ont interdit cet idéal, certains choix sont hélas autant de portes ouvertes sur des risques que je viens de palper de près. Et ça fait peur.

Il y a en premier lieu l'architecture. Il ne suffit pas de prendre Prism et ses Dll pour avoir un bon logiciel. Il faut comprendre et maîtriser la chose. Ce qui ne s'improvise pas. Mais il y a pire, car plus lié à la technologie elle-même.

Par exemple, prenez la syntaxe du Binding, en Xaml donc. Vous avez une balise, vous la complétez d'une propriété et là, au lieu de mettre une valeur, vous ouvrez des accolades américaines suivies du mot Binding, le tout entre guillemets, *une simple chaine de caractères*. Quant à ce qu'il y a après le mot Binding, je suis **convaincu qu'aucun aficionado de WPF ou de Silverlight n'est capable de me citer de tête toutes les combinaisons possibles** et astuces disponibles (j'ai d'ailleurs écrit un très long papier sur le sujet que je vous conseille). Parfois c'est {Binding} tout court, parfois c'est une longue phrase intégrant imbriquées d'autres accolades faisant référence à des ressources statiques ou dynamiques, des paramètres, des convertisseurs, etc… Une puissance énorme, un peu comme les expressions régulières : puissant mais pas très clair (et c'est un euphémisme). Pas clair, on peut s'y faire… mais pas clair et pas fortement typé ni même contrôlé, c'est là que la chute aux enfers commence…
Le code que j'ai audité était bourré d'element binding, de datacontext pointant des conteneurs de services avec des indirections dans tous les sens "pour la souplesse". Quand MVC et MVVM ne sont pas compris, mieux vaut tout mettre en procédural dans le code-behind de chaque fiche, c'est plus simple à débugger ! Le pire c'est que chacun dans l'équipe y allait de sa petite couche, de sa petite modification. Et je fais du "refactoring" par là, et je refactore par ici… Oui mais voilà, dans les balises Xaml ça commence à danser le cha-cha-cha toutes ses chaînes de caractères non contrôlées, tous ces paramètres de convertisseurs qui ont évolués sans qu'on ait mis à jour les références dans le Xaml, ces ressources statiques et d'autre dynamiques dans des dictionnaires chargés dynamiquement !

Le soft ne tenait plus que par un fil qu'un joyeux drille a du couper juste avant que je n'intervienne. Dommage. Je n'arrivais même pas à dépasser un ou deux écrans avant que ça

me pète à la figure. Même avec des outils très intelligents comme NDepend, comprendre le soft était virtuellement impossible.
Quant à savoir d'où vient "le" problème ! C'est le soft lui-même tout entier qui était "le" problème… Ainsi que ceux l'avaient écrit (et ceux qui les dirigent, car un mauvais soft est toujours la cause d'une mauvaise direction bien plus que de mauvais développeurs).

En fait, le Binding Xaml est une porte ouverte sur l'inconnu. Une feature d'une grande puissance, sans laquelle beaucoup du charme disparaîtrait, mais assez déraisonnable dans cette implémentation libre sous forme de chaînes de caractères non compilées. La porte sur le néant, l'incontrôlé, et pire : **l'incontrôlable**. Un trou noir syntaxique. La damnation du testeur envoyé moisir au purgatoire des pisseurs de ligne. Et l'enfer de l'auditeur.

LE BINDING AU PAYS DES X-FILES

Le Data binding Xaml est une jungle syntaxique pas très bien … balisée et totalement déconnectée de toute forme d'analyse à la compilation. du "Late Bugging" comme je m'amuse à appeler cette stratégie de type "late binding", principe de ligature tardive utilisé d'ailleurs en d'autres endroits et même sous d'autres frameworks. Pire que les Dynamic de C# 4 (pratiques mais dangereux), pire que F# (stimulant mais pas industrialisable), le Binding de Xaml est un gouffre à bugs, un chien fou s'il n'est pas tenu en laisse fermement.

En réalité un marteau ne pourra jamais être jeté aux fers (!) pour le meurtre de qui que ce soit. Un marteau est un outil, et même s'il a servi et servira encore dans de nombreux crimes, un outil est un objet sans âme, sans conscience et donc sans responsabilité. Même un fusil mitrailleur est un objet innocent, même un canon de 105 ou une bombe atomique sont plus innocents que l'agneau qui vient de naître et qui, comme tout être vivant, et à la mesure de son intelligence, portera le poids de la responsabilité de ses agissements. Un outil de développement restera donc à jamais hermétique à tout procès d'intention. **A celui qui s'en sert de le faire correctement**.
Il en va de même de Xaml, de son data Binding et de bien d'autres de ces facettes. La responsabilité incombera toujours à l'humain qui tient le marteau. A l'informaticien qui tient la souris. Au développeur qui tape un code affreux.

Mais certaines features de Xaml, certains choix conceptuels comme l'utilisation de chaînes de caractères non contrôlées et non parsées à la compilation sont à mon sens des erreurs. Si des projets comme celui que j'ai audité et dont je vous parle ici devenaient courants, nul doute que cela signerait l'arrêt de mort de WPF et de Silverlight. La faute aux mauvais développeurs ? Pas seulement. A ceux aussi qui ont décidé de programmer Xaml de cette façon trop ouverte, trop permissive. On voit bien comment Silverlight a été verrouillé dès le départ par Microsoft. Si le moindre virus, le moindre phishing avait été réalisé avec Silverlight 1 ou 2 c'en était fini des espoirs portés par cette technologie. Microsoft a été méfiant pour préserver l'avenir de la technologie et c'est une bonne chose. Ce que j'ai vu

dans le projet WPF dont je parle ici, c'est un peu de la même nature, mais à l'inverse : Microsoft n'a pas verrouillé Xaml comme cela a été fait avec Silverlight. Et si de tels détournements se généralisent c'est toute la technologie qui trinquera. D'ici un an environ, lorsque les projets lancés ces derniers temps seront finalisés, et qu'il faudra compter ceux qui n'aboutiront pas ou qui ne marcheront jamais bien, la note peut être salée pour Xaml. Microsoft a pris un sacré risque en faisant des choix de conception comme celui des balises non compilées (et dans lesquelles même Intellisence se prend les pieds dans le tapis).

Déjà sous Delphi je voyais souvent ce genre de code spaghetti avec des variables globales référencées n'importe où, des fiches utilisant des variables d'autres fiches jusqu'à créer des chaines de dépendances ingérables. J'ai vu des codes de ce type ne pouvoir jamais aboutir. Il m'est même arrivé une fois de réécrire en 15 jours proprement à partir de zéro un soft développés en 1 an par deux personnes sans le dire au client histoire que les 15 jours d'expertise qu'il m'avait payés servent à quelque chose… Je tairais ici le nom du client (une administration). J'étais plus jeune et je voulais me prouver des choses certainement, mais personne ne l'a jamais su jusqu'à ce billet (et encore vous en savez peu!). En tout cas ce projet là je l'ai sauvé. Mais être un pompier de l'ombre n'est pas ma vocation. Les gars étaient malgré tout étonnés ne pas vraiment s'y retrouver dans "leur" code. Amusante situation. Je n'avais rien gardé de "leur" code :-) Mais le soft marchait…

Mais avec Xaml, et une puissance décuplée, je viens de voir des horreurs du même genre alors que depuis des années, je commençais à trouver que C# incitait plutôt à faire du code "correct", le niveau étant globalement meilleur que celui que je voyais sur Delphi. Et patatras ! Xaml arrive avec ces chausses-trappes dans lesquels les développeurs s'engouffrent. La faute à Xaml ? Pas totalement, mais si tout était typé et contrôlé à la compilation certaines mauvaises utilisations ne seraient pas possibles. Après tout, le fortement typé en soi ça ne sert à rien si on suppose que tous les développeurs sont "bons" ! Mais tous les langages modernes tentent de l'être car on sait que l'humain est faillible. En créant une brèche incontrôlable dans un édifice aussi beau de Xaml, ses concepteurs ont fait un pari risqué. Celui que tous les développeurs sauraient maîtriser le potentiel sans tomber dans ses pièges. Un pari forcément perdu d'avance.

VISION D'HORREUR

Ce que j'ai vu est donc indescriptible. Un peu comme si j'essayais de décrire Cthulhu. Ceux qui ont essayés sont souvent morts avant de finir d'écrire son nom (ah Lovecraft…) !

Imaginez vous un logiciel de 250 fiches WPF environ utilisant de l'ADO.NET, des bouts de LINQ to SQL, et des nouveautés en Entity Framework, le tout à la sauce Prism / MVC (mais en ayant lu le manuel certainement à l'envers car Prism c'est très bien !) avec des tentatives d'inversion de contrôle (et des dérapages non contrôlés) farci d'un code Xaml bourré de Binding renvoyant vers on ne sait quoi (et hélas pas contrôlé à la compilation), le tout planqué dans 4 ou 5 couches DAL, BOL pré-BOL, post-BOL, et j'en passe, histoire de faire

court. La note s'élève a 6000 jours/homme (5 ans/homme environ). Pas un truc gigantesque mais qui commence à faire mal au budget malgré tout.

Agrémentez le tout de deux systèmes de versionning dont aucun n'avait vraiment une version complète du soft, des essais épars de tests unitaires avec MbUnit et VSTS. Vous obtiendrez le tableau. Une œuvre qui dépasse le classicisme habituel de ce point de vue, plus proche du Cubisme que de l'Hyperréalisme tout de même, avec au moins la bonne volonté de faire des petites choses (du mauvais testing prouve au moins qu'on a essayé de tester, ce qui est encore bien rare…). Mais l'enfer est pavé de bonnes intentions, c'est un peu ça le problème.

Le code était impossible à maintenir, les problèmes de performances impossibles à cerner sans y passer plus de temps qu'à toute refaire, **le code Spaghetti, avec un grand S avait frappé de son coup de poignard vengeur et lâche dans le dos**. Une **complexité non maîtrisée** qui tel l'horizon d'un trou noir aspirait irrémédiablement le bon code vers l'enfer central laissant le hasard faire le tri entre les chemins à prendre… Démêler l'écheveau n'était pas possible, pas plus que de formuler des guidelines sérieusement applicables dans un tel contexte ni aucun conseil pour se sortir d'une telle situation.
Vous imaginez peut-être l'horreur qu'a été l'écriture du rapport d'audit. Entre dire une vérité que personne n'était prêt à entendre et mentir sachant que peu importe ce que je dirais cela passerait mal et qu'au fond mieux valait ne pas trop déplaire… Mon éthique a tranché, j'ai dit la vérité. En l'enrobant. Des heures passées à réécrire deux fois, dix fois certains paragraphes. Ils ne s'en douteront jamais… Et au final un rapport qui sera peut-être classé au fond d'un tiroir car personne ne voulait vraiment savoir ce qui n'allait pas. La remise en question d'un tel échec dépasse de loin le cadre technique et peu de gens savent admettre leurs erreurs, surtout quand toute la chaîne hiérarchique de la base au sommet doit participer à ce questionnement.

LE SALAIRE DE LA PEUR

La peur dont je parlais plus haut, les chocottes que cet audit m'a données, c'est d'être confronté au fleuron de la technologie informatique, à des **choses en lesquelles je crois** car elles marquent un réel progrès et que (bien utilisées) elles **permettent justement un code limpide**. Cet audit a brisé ce fantasme en me rappelant qu'un marteau pouvait servir au meilleur, comme au crime. Xaml, WPF et Silverlight n'échapperaient pas à la règle et il faudra être vigilant. Surtout que l'avalanche de technologies et de patterns (WCF Ria Services, Entity Framework, MVVM, Prism, …) rendent presque impossible la maîtrise de tous ces sujets. Je suis payé pour ça. Au minimum 50% de mon temps est passé en autoformation pour apporter un conseil éclairé sur les technos à venir. Ce sont les 50% vendus qui financent cette formation et cette recherche permanente. Comment un développeur qui fait ces 48h (35 de base + le dépassement obligatoire non payé car il est aux "cadres") peut-il se former à tout cela en travaillant sur d'autres projets la journée ? C'est

impossible. Et cela créé une situation très dangereuse.**Technologie sophistiquée + manque de formation = code spaghetti !**

DAVID VINCENT AU PAYS DES TAGS

Il était donc urgent de vous en parler, d'attirer votre attention sur certaines dérives, car dans une moindre mesure, je sais, je l'ai vu chez d'autres, ces chausses-trappes savent s'ouvrir sous les pieds des développeurs les mieux formés et les mieux intentionnés ! Je les ai vues, et je dois convaincre un monde incrédule…

Ma mission, que j'ai acceptée (forcément avant de voir, on n'a pas vu…), était de faire un audit de pré-release. C'est à dire de venir faire le point sur l'état de la situation, les derniers fameux "boulons à serrer" en quelque sorte, et surtout de venir signer un satisfecit au DI, très fier de son bébé (Rosemary's baby plutôt que Dora de TF1 au final), afin qu'il puisse faire monter sa prime de fin d'année je suppose.

Hélas, déception et vilenie. Ce n'est pas avec le prix d'un audit (fort raisonnable, devis gratuit sur simple demande) qu'on achète ma conscience. Il m'a donc fallu trouver les mots et les formules diplomatiques pour rendre un audit policé et mesuré sur cet édifice in-maintenable, voué à l'échec et bon à mettre à la broyeuse. Ce n'est pas la partie la plus difficile, car tout rapport d'audit se doit d'être policé et tourné de façon neutre et technique. Même si parfois on a envie de crier "bande de nuls !", non, ce n'est pas bien, on ne le fait pas…

Que les futurs clients ne s'affolent pas trop, ce que je raconte aujourd'hui est malgré tout à classer dans les exceptions, le fameux bout de la cloche de la courbe Normale. Même si je ne suis que rarement content du code que je vois, on est, heureusement en moyenne, assez loin de l'horreur qui justifie le présent billet !

O TEMPORA, O MORES !

Parfois je rêve, j'imagine un monde où l'un de ces clients m'appellerait en me proposant un pont d'or pour lui avoir ouvert les yeux et l'avoir empêché de plonger plus encore tant qu'il était encore temps… Mais ce n'est qu'un rêve, bien entendu. Je suppose que je n'entendrais plus parler de ce client comme de quelques autres dont, les années passant et les vœux annuels restant sans réponse m'ont forcé à une résignation réaliste. Comme le chantait l'artiste, "il a dit la vérité, il doit être exécuté" !

Hélas, les temps ne sont pas à la prise de conscience ni à la bravoure. Nous vivons une époque de lâches où chacun ouvre son parapluie pour que les gouttes en atteignent d'autres. Forcément celui qui est au plus bas de la pyramide se prend tout sur la figure. Souvent c'est le développeur qui trinque pour une chaîne hiérarchique défaillante qui n'a

pas fait son métier et qui n'assume pas ses responsabilités. Un développeur n'est jamais coupable seul d'un mauvais code, c'est toute la chaine de commandement qui faillit.

*Il y a encore un rêve que je fais et que j'aimerais voir se réaliser avant ma (encore lointaine) retraite : que les entreprises fassent intervenir un conseil, un auditeur **avant** que l'irréparable ne soit commis, pas après pour constater les dégâts ! … ça serait tellement mieux et plus gratifiant pour moi et mes confrères ! Conseiller, orienter, former, en un mot aider, c'est tout de même plus chouette que de passer pour le père fouettard et l'inspecteur des travaux finis !*

FAUT-IL BRULER XAML ?

Certes non.

Mais ce que je veux qu'il vous reste de ce récit hallucinant autant que réel, c'est que Xaml est utilisé par certains comme un langage dans le langage. Un moyen de vider le code-behind de son C# pour remplir des balises ésotériques. Un langage dans le langage avec son Binding qui est, telles le sont les poupées Russes, encore un langage dans le langage.

A vouloir tout faire par binding, à tout dynamiser, à appliquer tout Prism et le chargement dynamique des modules, la découvertes des plugins avec MEF, le tout mis en forme façon MVVM, sans compter les ruses purement graphiques du templating sous Blend, des DataTemplate à double face pivotante en 2,5 D, à vouloir appliquer tout cela dans un seul soft **sans avoir pris les 2 ou 3 ans nécessaires à acquérir la parfaite maîtrise méthodologique pour faire marcher tout cela ensemble**, on peut facilement **arriver à une situation désastreuse**. Encore plus facilement que sous C, encore plus que sous Delphi, pire que sous Windows Forms. Tout cela n'était que de l'amateurisme face aux dérapages délirants que WPF et Silverlight peuvent permettre… Comment faire des tests unitaires parlants sur des balises de Binding imbriquées à un tel point que même Intellisense de VS s'y perd ?
L'extrémisme essayiste est un fléau dans notre métier. Il m'a toujours effrayé, mais il a toujours eu sa sanction "naturelle" : une technologie qui finit par être si mal utilisée créée une telle publicité négative qu'elle tend à disparaître. Je ne voudrais pas que cela arrive à WPF et à Silverlight qui apportent tant d'autres progrès !

IL FAUT SAUVER LE SOLDAT XAML !

Pitié pour Xaml, un bon geste pour WPF et Silverlight : ne laissez pas des **sagouins** faire du mal à toutes ces belles technologies que Microsoft nous offre depuis quelques années. Laissez leur une chance, celle que les développeurs deviennent plus matures et mieux formés aux méthodologies qui vont avec. Ne commettez pas le crime d'être complice de ce genre de chose. Intégrer les technologies les unes après les autres, ne sautez pas d'une bêta à l'autre sans avoir l'assurance que vous maîtrisez la précédente !

Mieux vaut une application Silverlight fonctionnant avec un Service Web classique et dont les champs sont connectés par code, qui marche et qui est maintenable, qu'une élucubration technologique improbable et non maîtrisée qui s'écroulera comme un château de carte au premier éternuement du premier bug venu…

UN PEU DE SERIEUX…

Il ne s'agit pas d'une invitation à la méfiance, encore moins au passéisme. Non, c'est une invitation au professionnalisme : *n'utilisez que ce que vous maîtrisez*.
Si vous maîtriser le top de la techno, allez-ci, franchement. Si vous avez un petit doute : faites joujou dans une machine virtuelle et produisez du code que vous comprendrez à 100% et surtout que d'autres pourront comprendre pour le maintenir, le faire vivre dans les années à venir.

Le code Kleenex coûte cher, il dévalorise notre métier et les outils que nous utilisons.
Le code spaghetti peut assassiner une technologie.
Codez bien. Formez-vous bien.
Et faites auditer vos projets avant, pas après.
Et Stay Tuned, j'aurais des trucs plus légers à vous dire la prochaine fois !

Patterns & Practices : les guides de bonnes pratiques à connaître par coeur !

Dans la jungle du Framework et de tous ses projets satellites qui sortent au rythme d'une rafale d'AK47 comment un pauvre développeur isolé peut-il intégrer et digérer en quelques heures (en plus de son travail quotidien) des centaines, voire des milliers d'années-homme de librairies, technologies, outils et langages produits par Microsoft ?

C'est une véritable question. Dans mon propre travail de conseil je m'oblige à une réserve de 30% de mon temps uniquement dédié à la veille technologique, c'est à dire qu'un tiers de mon temps n'est jamais vendu, je me le réserve pour apprendre, un luxe indispensable mais coûteux en chiffre d'affaire potentiel perdu. De plus c'est un mode de fonctionnement que seul un dirigeant d'entreprise ou un indépendant peut s'offrir. Et malgré ce privilège tous les jours je mesure l'étendue de mon ignorance sur certains aspects du Framework avec la désagréable impression que plus je rame plus la côte s'éloigne ... Je suis certain de n'être pas le seul à ressentir cette sensation !

Il y a ceux qui ont d'emblée choisi de se spécialiser. Ils connaissent tout ou presque de Windows Forms, de WPF, ou de Silverlight mais ignore tout des dizaines d'autres éléments du Framework. Impasse sur ASP.NET, Ajax, MVC, Entity Framework, WinRT, etc, etc. Chacun fait alors son petit marché n'ayant au final qu'une vue très restreinte sur le Framework, et, de fait, loupant souvent d'excellentes choses. Hélas une journée n'a que 24h, et pour avoir essayé par tous les moyens de contourner sans succès cette terrible réalité, je peux affirmer

que travailler, même beaucoup, même trop, n'est pas forcément la solution. Seule Zenon d'Elée arrivait dans son paradoxe à faire gagner une tortue face à Achille ! Quand l'adversaire est en supériorité, la force brute est inutile, il faut ruser...

Microsoft a conscience de ce problème. Qu'il s'agisse des petites vidéo "how do I", des multiples conférences qui se tiennent régulièrement, de la documentation très fournie de MSDN, de l'excellent magazine MSDN toujours riche d'articles de haute tenue technique, et de bien d'autres actions en faveur de la diffusion de la connaissance sur ses produits, Microsoft fait beaucoup pour nous aider à appréhender l'étendue de sa gamme.

Si cet effort louable est important, on peut toujours en réclamer plus. Par exemple une certaine décentralisation de toutes ces informations fait que peu de gens connaissent tous les "points d'entrée" utiles de ces informations. Gageons que Microsoft en a aussi conscience et que des efforts supplémentaires seront réalisés pour aider le développeur "à s'y retrouver dans les informations qui permettent de s'y retrouver" dans les produits...

PATTERNS & PRACTICES

Tout cela pour vous parler aujourd'hui des Patterns & Practices. Il s'agit d'un ensemble de recommandations et de code mis à disposition gratuitement sur CodePlex. Bien connaître ces "patrons et pratiques" peut vous aider à mieux tirer parti du Framework sans pour autant y consacrer vos nuits.
Pour obtenir la liste de tous les projets gravitant autour de ce concept de "patterns & practices" allez sur CodePlex et chercher cette expression. Vous pouvez aussi directement accéder à tous les projets pertinents par le tag du même nom (colonne de droite sur CodePlex).

LES PROJETS CLE

Il est bien entendu très difficile de créer un ordre de priorité dans tous les projets "patterns & practices", selon ce que vous développez, l'urgence de regarder tel ou tel projet sera plus ou moins grande. Mais je vais tenter une petite sélection de ceux qui, à mon sens, sont à connaître absolument.

APP ARCH GUIDE 2.0 KNOWLEDGE BASE

http://www.codeplex.com/AppArch
Ce projet initié par JD Meier, Jason Taylor et Prashant Bansode regroupe de nombreux documents et vidéos dont le but est d'expliquer "*How to put the legos together*", ce dont je parlais plus haut : savoir comment utiliser correctement toutes les briques de constructions du Framework pour en faire quelque chose.
The purpose of the Application Architecture Guide 2.0 is to improve your effectiveness building applications on the Microsoft platform. The primary audience is solution architects and developer leads. The guide will provide design-level guidance for the architecture and

design of applications built on the .NET Framework. It focuses on the most common types of applications, partitioning application functionality into layers, components, and services, and walks through their key design characteristics.
Ce projet est certainement le premier point d'entrée que vous devez connaître. Sa vision globale de l'architecture des applications vous aidera à prendre de meilleures décisions.

APPLICATION ARCHITECTURE GUIDE (LE LIVRE)

http://www.codeplex.com/AppArchGuide
Ce guide fournit des guides architecturaux pour la création d'applications .NET. Les principaux types d'applications sont étudiés et de nombreuses solutions sont proposées.

Le livre complète la "App Arch Guide 2.0 Knowledge base" décrite ci-dessus.

ENTERPRISE LIBRARY

http://www.codeplex.com/entlib
http://msdn.microsoft.com/fr-fr/library/cc467894(en-us).aspx
L'Enterprise Library est une collection de blocs applicatifs qui ont été conçus pour assister le développeur dans son travail. Il s'agit de code mais d'une certaine façon ce code constitue un guide des bonnes pratiques.

Le code source est fourni avec une documentation. EL peut être utilisé tel quel ou bien modifié ou étendu.

Constitué de plusieurs "blocs", EL est une vraie mine d'or. Gestion du caching, cryptographie, accès aux données, gestion des exceptions, logging, sécurité, des solutions pratiques, génériques et éprouvées sont apportées à chacun de ses sujets.

Un "must have" donc. (et surtout un "must understand" !).

WEB CLIENT SOFTWARE FACTORY

http://www.codeplex.com/websf
Le WCSF fournit un ensemble de directives pour les architectes et les développeurs d'applications Web. La factory inclue des exemples, du code réutilisable et un ensemble de guidlines pour automatiser les tâches clé du développement sous Visual Studio. Pour ceux qui connaissaient, WCSF remplace UIP (User Interface Process Application Block). WCSF supporte bien entendu les nouveautés du Framework comme ASP.NET, Ajax ou Workflow Foundation.

A connaître dès lors qu'on veut développer des applications Web...

COMPOSITE WPF AND SILVERLIGHT

http://www.codeplex.com/CompositeWPF

Ce bloc des patterns & practices fait partie du top 5 à connaître et maîtriser. Il intègre code et guidelines nécessaires à la mise en place d'architectures suivant les best practices pour les projets de type WPF et Silverlight. Tout est bon, il faut absolument le connaître !

A noter: ce projet est aussi connu sous le nom de "Prism" (info primordiale surtout pour faire le cador à la machine à café ☺).

SMART CLIENT GUIDANCE

http://www.codeplex.com/smartclient

http://msdn.microsoft.com/en-us/library/aa480482.aspx

Le Smart Client Software Factory (SCSF) est une autre guide essentiel. Guidelines et code forment un ensemble de composants réutilisables sous Windows Forms autant que WPF ou ASP.NET pour mettre en place une architecture de client intelligents composites.

Je n'ai pas eu encore le temps de plonger dans ce guide, mais du survol que j'en ai fait, il faut très certainement le regarder de plus près, les solutions proposées semblent tout aussi indispensables à connaître que les autres guides de la série "patterns & practices".

UNITY

http://www.codeplex.com/unity

Encore un guide que je n'ai pas eu le temps de lire... Les longues soirées d'hiver sont un mythe : il fait nuit plus tôt mais les journées ne comptent toujours pas plus de 24h, cette expression est donc une escroquerie ! :-)

The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection.

Dit comme ça, c'est un peu confus... pour tout savoir le mieux c'est de lire (le conseil vaut pour moi aussi) !

WCF SECURITY GUIDANCE

http://www.codeplex.com/WCFSecurity

La mise en place de services via WCF peut se faire de façon très naïve... ou de façon professionnelle, c'est à dire en gérant correctement la sécurité ! Ce guide fait le tour de la question et propose des guidelines autant que du code pour mieux sécuriser les applications communiquantes et gérer correctement les authentifications, les autorisations et toutes ces choses indispensables pour des applications mises en production.

WEB SERVICE SOFTWARE FACTORY

http://www.codeplex.com/servicefactory

Il s'agit d'une collection d'outils, de patterns, de code source et de guidelines destinés à vous aider à construire des Web services WCF et ASMX rapidement mais en garantissant la meilleure fiabilité possible.

Indispensable comme le reste...

GUIDANCE EXPLORER

http://www.codeplex.com/guidanceExplorer

Voici peut être un moyen de s'y retrouver un peu mieux parmi toutes les guidelines ! le Guidance Explorer, comme son nom l'indique, est un outil d'exploration des guidelines. Une fois installé il se met à jour via le Web.

Centralisant bon nombre de guides et simplifiant l'accès à l'information, Guidance Explorer est l'une des premières choses à installer à côté de Visual Studio !

MS HEALTH COMMON USER INTERFACE

http://www.codeplex.com/mscui

C'est un peu l'ovni de cette liste et même un vrai ovni à part entière dans tout le Framework et les guidelines. On se demande pourquoi Microsoft a investi dans cette branche très spécialisée plutôt qu'une autre. Le MSCUI est en effet un ensemble de guidelines, de code et de composants permettant de réaliser des applications orientées médical. Composants WCF, Silverlight ou autres, c'est un ensemble incroyable quantitativement et qualitativement. Ayant été l'un des pioniers de l'informatique médicale en France avec la gamme de logiciel Hippocrate je connais particulièrement bien la question et j'ai été bluffé par ce que propose MS avec MSCUI. Avec cet outil un bon développeur peut assez rapidement mettre en place des solutions tout à fait honorables capables de concurrencer les principaux logiciels médicaux du marché...

Si le médical n'est pas votre branche, MSCUI ne vous intéressera que peu dans la pratique, mais regardez tout de même comment cela est fait et comment les composants se présentent, ergonomiquement et fonctionnellement c'est un beau travail.

CONCLUSION

Si ce tour d'horizon n'est pas exhaustif il contient malgré tout le top des guidelines et outils à connaître pour développer sereinement des solutions basées sur des patterns éprouvées. Ne pas réinventer la roue, mettre rapidement en place la bonne architecture d'un projet c'est déjà s'assurer à 50% de sa réussite.

Si les soirées d'hiver ne comportent pas plus d'heures que celles d'été, la froidure incite moins à sortir et à flâner qu'en août, profitez-en pour rester au chaud en vous plongeant dans les Patterns & Practices !

N'oubliez pas ce vieux proverbe qui nous vient du fond des âges :

"L'hiver, qui bouquine Patterns & Practice,
l'été, produit du code qui glisse !"

Prism v4 (guidance and patterns for WPF and Silverlight)

Construire des applications modulaires offre de nombreux avantages : maintenabilité et évolutivité sont les premières qui viennent à l'esprit mais il en existe d'autres comme la meilleure séparation des tâches (travail parallèle d'équipes de développement sur des modules différents) par exemple.

Créer une architecture assurant la modularité d'une application n'est pas chose aisée. Bricoler "sa" solution dans "son" coin donne l'impression de gagner du temps (pas besoin d'apprendre un framework existant) mais montre souvent ses limites et ce, au pire moment, c'est à dire trop tard...

Microsoft ne fait pas que du soft pour micro... Depuis l'avènement de .NET il faut saluer les efforts importants qui sont fournis par MS pour fournir aussi de la matière grise. Labos de recherche, groupes de travail très indépendants, cette nouvelle orientation du management des équipes a permis l'éclosion d'un tas de bonnes idées. Tout ce travail est gracieusement délivré aux communautés de développeurs qui se donnent la peine de s'y intéresser...

Qui plus est, il ne s'agit pas d'élucubrations fumeuses. Les guides de bonnes pratiques, les conseils méthodologiques sont malgré tout le fruit d'un énorme travail collaboratif "au sommet" avec aux commandes des gens comme JD Meier, qui ne sont pas petites pointures !

Pour en revenir aux applications modulaires, il est essentiel de prendre connaissance de la **V4 de PRISM**, un recueil de codes, de documentations et de bonnes pratiques d'une **qualité exceptionnelle**.

Prism V4 c'est :

- Une librairie pour la création d'applications composites
- Une application de référence comme modèle d'implémentation (gestion de porte feuille boursier)
- Des "quick start" pour entrer dans le vif du sujet rapidement
- Des "how-to's" pour se former efficacement
- Une documentation à la fois claire et riche

Prism supporte les applications **WPF et Silverlight ainsi que Windows Phone 7** dans une même logique permettant de **partager encore plus de code entre les trois types d'application**.

Bref, je vous invite à vous pencher très sérieusement sur Prism si vous ne connaissez pas, pour tout projet d'une certaine envergure cela vous fera gagner beaucoup de temps, de productivité et le tout dans un cadre validé ne risquant pas de vous envoyer au mur.

Télécharger les éléments de Prism v4
Le site de Prism sur CodePlex
[Article mis à jour pour ce PDF, l'article original pointait la V2]

PEX : Tests unitaires intelligents pour Visual Studio

PEX est un produit des laboratoires de recherche de Microsoft. Vous allez me dire "*encore un 'truc' pour faire du testing !*" et je vous répondrai que si on avait déjà atteint la perfection ça se saurait et que donc, par force, il y a encore largement matière à recherche et à nouveaux produits en la matière !
Et PEX apporte beaucoup de nouvelles choses qui vont dans le sens de l'histoire, c'est à dire des tests unitaires **plus faciles à intégrer** au code, **plus complets**, **plus dynamiques** et surtout **plus "intelligents"**.
PEX offre ainsi une façon de travailler plus moderne en apportant une aide précieuse à la création de tests. Mieux, PEX utilise un Add-In pour VS 2008 et 2010 qui sur un simple clic droit permet de générer des tables pour tester une méthode. Par exemple, vous avez une méthode qui capitalise une chaîne passée en paramètre et qui retourne le résultat. Un clic droit dans le corps de la méthode et un appel à PEX et ce dernier génère des valeurs pour tester la méthode. Un null passé en paramètre par exemple... Ensuite il va dérouler les appels à la méthode avec les conditions générées et établir un rapport.

Mieux, à partir des conditions de test établies (modifiables) PEX peut générer une "vraie" méthode de test pour retrouver l'esprit plus habituel des projets de tests unitaires (on peut ainsi dérouler ultérieurement les mêmes tests pour contrôler l'éventuelle régression d'un code). Mais cela n'est pas obligatoire, on peut juste utiliser PEX "à la volée".

Plus fort et sans les mains, face à certaines conditions d'erreur, par exemple le passage d'un null en paramètre évoqué plus haut, PEX peut vous proposer de fixer le code. Dans un tel cas il ajoutera le code nécessaire (ici une exception de type *ArgementNullException* si la chaîne passée en paramètre est null) !
PEX est un travail en cours, comme tout ce qui émane des laboratoires de recherche de Microsoft. Il montre l'une des directions étudiées par Microsoft pour améliorer les tests unitaires dans de futures versions de Visual Studio. L'avantage est bien entendu qu'il ne s'agit pas d'une démo, mais d'un produit très efficace d'ores et déjà utilisable.

Le mieux c'est de de vous rendre sur le site de PEX, lire les PDF, télécharger le produit, voir les vidéos et tout ça... [Les choses ont évolué depuis ce billet, j'ai mis à jour le lien ci-dessus, je vous conseille d'y faire un tour, il y a même de quoi tester les PCL, « code digger »]

MEF - Managed Extensibility Framework - De la magie est des plugins !

UNE GESTION DE PLUGIN SIMPLIFIEE

Actuellement encore en preview mais très utilisable depuis la Preview 2 du mois d'octobre, MEF est un nouveau procédé de **gestion des plugins** pour le Framework .NET.
Projet Open Source se trouvant sur CodePlex (http://www.codeplex.com/MEF) MEF facilite l'implémentation des *addins* (ou *plugins*) en automatisant la liaison entre les propriétés du programme qui importe des valeurs et les addins qui exportent les valeurs. Sachant que *tout module peut être importateur et exportateur à la fois*, permettant des chaînes d'addins eux-mêmes supportant des addins…[A la date de création de ce PDF, 10/2013, MEF est intégré dans .NET et Silverlight et un package Nuget est disponible pour WinRT]

MEF ET LES AUTRES

Microsoft a intégré dans le Framework 3.5 une gestion des plugins qui se base sur l'espace de nom**System.Addin**. L'approche est différente de MEF et le choix n'est pas évident entre ces deux solutions. D'autant qu'il en existe une troisième ! En effet, Microsoft a aussi publié le **Composite Application Guidance for WPF**, spécifiquement dédié aux applications de ce type donc, dont la dernière version date de juin…
MEF est utilisable aussi sous WPF, même sous Silverlight mais je n'ai pas encore testé cet aspect-là.

COMMENT CHOISIR ?

Personnellement l'approche de MEF me convient très bien, c'est assez simple et cela répond aux besoins d'une gestion de plugins (ou addins). En ces temps d'avalanche de technologies toutes plus alléchantes les unes que les autres chez Microsoft il est vrai que je suis assez tenté par la simplicité de MEF qui évite de trop s'encombrer les neurones déjà bien saturés ! Simple et complet, je préfère donc MEF, mais je suis convaincu que dans certains cas la solution spécifique à WPF est mieux adaptée ou que System.Addin apporte certains petits plus (sécurité par exemple). J'avoue bien humblement que je n'ai pas encore trouvé le temps de tester à fond System.Addin ni la solution WPF. A vous de voir donc, et le mieux c'est de regarder de près en testant chaque approche. Ici je vais vous parler de MEF, pour les autres solutions suivez les liens suivants :

Composite Applicationn Guidance for WPF (http://msdn.microsoft.com/en-us/library/cc707819.aspx)

Pour System.Addin je vous conseille les 12 billets de Jason He qui sont plus parlant que l'aride documentation de l'espace de nom sur MSDN.
(http://blogs.msdn.com/zifengh/archive/2007/01/04/addin-model-in-paint-net-1-introduction.aspx)

MEF – LE PRINCIPE

Le but est de simplifier l'écriture d'applications dites extensibles. MEF automatise la découverte des modules (les plugins) ainsi que la composition des valeurs, c'est-à-dire un lien automatique entre les valeurs exportées et le module importateur. De prime abord ce n'est pas forcément très clair, mais le code qui va venir va vous éclairez (je l'espère en tout cas !). En première approximation disons que la composition dans MEF est une sorte de Databinding qui relie une propriété déclarée dans l'importateur à une ou plusieurs valeurs du ou des modules exportateurs (les plugins).

MEF – LES AVANTAGES

MEF est assez simple, je l'ai dit, et c'est un gros avantage (mais pas simpliste, nuance). Il est Open Source c'est un plus. Mais surtout MEF **évite de réinventer la poudre à chaque fois qu'on désire implémenter une gestion de plugins**. Et toute application moderne se doit d'être extensible ! Qu'il s'agisse d'applications à usage interne ou bien de logiciels d'éditeur, c'est souvent l'extensibilité et les plugins qui font le succès d'une application, ou aide grandement à celui-ci. Disposer d'une solution fiable pour résoudre ce problème d'architecture très courant est donc l'avantage principal de MEF.
Les extensions créées avec MEF peuvent être partagées par plusieurs applications, elles peuvent elles-mêmes utiliser des extensions et MEF saura les charger dans le bon ordre automatiquement.

MEF propose un ensemble de service de découverte simplifiant la localisation et le chargement des extensions. On trouve aussi un système de lazzy loading et un mécanisme de métadonnées riches permettant aux plugins d'informer l'hôte sur sa nature ou transmettre des données complémentaires.

UN EXEMPLE ! UN EXEMPLE !

Bon, je ne vais pas refaire la doc de MEF, qui n'existe pas d'ailleurs (enfin si mais c'est encore très partiel), et pour illustrer le propos je vais expliquer le fonctionnement de MEF au travers d'un exemple le plus simple possible (ce billet s'annonce déjà bien long !).

Installer MEF

Avant toute chose, et pour faire tourner l'exemple (et vous amuser avec MEF), il faut que vous installiez MEF. Rassurez-vous, c'est très simple, sur le site de MEF (indiqué en introduction) vous trouverez dans l'onglet *Releases* le dernier zip à télécharger. Il suffit de prendre le contenu du zip et de le copier quelque part sur votre disque. C'est tout. Le source est fourni ainsi que la lib compilée. Il suffit dans une application d'ajouter une référence à la DLL « System.ComponentModel.Composition.dll » se trouvant le répertoire /Bin du fichier téléchargé et l'affaire est jouée. Un Using deSystem.ComponentModel.Composition sera nécessaire dans l'application hôte ainsi que dans les applications fournissant un service (les

DLL des plugins). [On notera que désormais MEF est aussi intallable depuis VS et les Nuget Packages]

L'application exemple

Je vais faire très simple comme annoncé : prenons une application console. Cette application doit appliquer des calculs à des valeurs. Tous les algorithmes de calcul seront des plugins. Algorithme est un bien grand mot puisque dans cet exemple j'implémenterai l'addition et la multiplication. Dans la réalité les plugins découverts seraient par exemple ajoutés à un menu ou une toolbar. Ici nous nous contenterons de les lister à la console et de les appeler sur des valeurs codées en dur (dans une vraie calculette les valeurs seraient saisies par l'utilisateur).

Le contrat

Le plus intelligent pour une gestion de plugin est bien entendu d'avoir une ou plusieurs interfaces décrivant ce que sait faire un plugin. C'est le contrat (ou les contrats) entre l'hôte et ses plugins.

Ainsi nous allons ajouter à notre solution un projet de type DLL pour décrire cette interface. Cela se fait dans un projet séparé puisque l'interface doit être techniquement connu à la fois de l'hôte et des plugins et qu'il faut éviter à tout prix l'existence de dépendances entre ces deux niveaux de l'architecture. De plus l'interface peut ainsi être diffusée avec l'exécutable pour que des tiers puissent écrire des plugins.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MEFInterface
{
   public interface ICompute
   {
      double Compute(double x, double y);
      string OperationName { get; }
   }
}
```

Le code ci-dessus est très simple. Comme on le voit ce projet DLL ne fait aucune référence à MEF ni à notre application ni à rien d'autre d'ailleurs (en dehors du Framework). L'interface ICompute expose une méthode Compute() et une propriété de type chaîne OperationName. Compute réalise le calcul sur les deux valeurs passées en paramètre et OperationName retourne le nom de l'algorithme pour que l'hôte puisse éventuellement fabriquer un menu listant tous les plugins installés.

Les plugins

Cela peut sembler moins naturel de commencer par les plugins que par l’application hôte mais je pense que vous comprendrez mieux dans ce sens là. Donc comment implémenter un plugin ?

Nous ajoutons à notre solution un nouveau projet de type librairie de classes qui sera faite d’une seule classe implémentant l’interface que nous venons de voir. Prenons l’exemple de la DLL de l’addition, sachant que celle gérant la multiplication est identique (au calcul près) et que nous pourrions créer ainsi une ribambelle de projets implémentant autant d’algorithmes de calculs que nous en voulons.

Le projet ModuleAddition contient ainsi un fichier Addition.cs, fichier contenant le code suivant :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.Composition; // MEF
using MEFInterface; // interface des plugins

namespace ModuleAddition
{

   [Export(typeof(ICompute))] // exporter la classe vue sous l'angle de l'interface partagée
   public class Addition : ICompute
   {
      // Réalise l'addition de deux doubles.
      public double Compute(double x, double y)
      {
         return x + y;
      }

      public string OperationName { get { return "Addition"; } }

   }
}
```

Ce code est redoutablement simple et n’a pas grand-chose de spécial. Il implémente l’interface que nous avons définie (d’où le using à MEFInterface, nom de notre projet interface, rien à voir avec un module de MEF donc, et la référence ajoutée cette DLL).

Ce qui est spécifique à MEF se résume à deux choses :

D'une part le using deSystem.ComponentModel.Composition qui implique l'ajout dans les références de la DLL de MEF et d'autre part l'attribut **Export** qui décore la classe Addition (implémentant l'interface ICompute que nous avons créée).

L'attribut **Export** possède des variantes, ici l'usage que nous en faisons indique tout simplement à MEF que la classe en question est un fournisseur de service plugin et qu'il faut la voir non pas comme une classe mais comme l'interface ICompute. C'est un choix, il peut y en avoir d'autres, mais concernant une gestion de plugin cette approche m'a semblé préférable.
Concernant le code de la classe Addition on voit qu'elle implémente les éléments de notre interface donc la méthode Compute qui retournera ici l'addition des deux paramètres. Dans la classe Multiplication(l'autre plugin non visible ici) c'est la même chose, sauf que Compute calcule la multiplication. La propriétéOperationName de l'interface est implémentée en retournant le nom de l'algorithme de calcul exposé par le plugin. Ce nom sera utile à l'hôte pour créer un bouton, une entrée de menu, etc.
On notera que MEF supporte la notion de métadonnées. Il existe ainsi des attributs permettant de décorer une classe en ajoutant autant de couple clé / valeur qu'ont le souhaite. Le nom du plugin, sa version, le nom de l'auteur et bien d'autres données peuvent ainsi être transmis à l'hôte via ce mécanisme que je ne démontre pas ce billet.

Pour simplifier les tests notons que j'ai modifié les propriétés du projet de chaque plugin pour qu'en mode debug les DLL soient directement placées dans le répertoire de debug de l'application hôte. Jai choisi ici aussi la simplicité : les plugins doivent être dans le répertoire de l'exécutable. Bien entendu dans la réalité vous pouvez décider de créer un sous-répertoire à votre application pour les plugins, ce qui est même plus propre.

L'hôte

Le voici ! Application en mode console (quel mode merveilleux pour les tests !), son fonctionnement est rudimentaire : nous voulons que l'application découvre grâce à MEF tous les plugins installés et que pour chacun elle affiche le nom de l'opération et effectue un appel au calcul correspondant. Les valeurs servant à ces derniers sont figées dans le programme, pas question d'ajouter une gestion de saisie utilisateur dans cette démo.

Les using et références

Pour fonctionner le programme doit faire référence à la fois à la MEF et au projet interface (aux projets interfaces si nous supportions plusieurs types de plugins par exemple).

Ces références trouvent leur pendant dans les using :

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.Composition; // MEF
using MEFInterface;
using System.IO; // interface des addins
```

La section Main

Elle aura pour rôle de créer une instance de l'objet programme et d'appeler sa méthode Run (nom arbitraire bien entendu). Rien de spécial à comprendre donc.

```
static void Main(string[] args)
{
  new Program().Run();
}
```

La propriété importée

Nous avons vu que les plugins exportent une (ou plusieurs) valeur(s), dans notre exemple il s'agit d'instances de classes supportant l'interface ICompute. Du côté de l'hôte il faut « quelque chose » pour importer ces valeurs.

C'est le rôle de la propriété **Operations** dans notre code :

```
[Import] // attribut marquant une propriété importée depuis les addins découverts
      public IEnumerable<ICompute> Operations { get; set; }
```

Comme vous le voyez cette propriété est une liste. Cela s'explique par le fait que nous supportons plusieurs plugins à la fois. Dans le cas contraire (un seul plugin) la propriété aurait été directement du type de l'interface ICompute. Un seul plugin peut sembler étrange mais cela peut correspondre à une partie spécifique de votre application que vous désirez pouvoir personnaliser à tout moment en fournissant simplement une nouvelle DLL qui écrasera celle installée chez les utilisateurs. Un mode d'utilisation à ne pas négliger… Mais ici nous supportons plusieurs plugins à la fois et notre propriété est ainsi une liste. Autre spécificité liée à la MEF, la propriété est décorée par l'attribut **Import** qui indique à MEF qu'il devra faire le binding entre la propriété et les plugins supportant le type de celle-ci.

Le code du Run

L'application se déroule ici. Nous commençons par appeler une méthode Compose() dont le rôle sera justement de mettre en route toute la magie de MEF. Nous verrons cela plus bas. Ensuite nous ne faisons que boucler sur la collection représentée par notre propriété comme si nous y avions déjà placé du contenu et nous l'utilisons « normalement » c'est-à-dire comme si MEF n'existait pas. Rien de bien sorcier là non plus, un foreach énumère toutes les entrées (des instances vues comme des interfaces ICompute) et utilise les méthodes et les propriétés accessibles (le nom du plugin et son unique méthode Compute).

```
private void Run()
{
```

```
    Compose();
    Console.WriteLine("Operation Addins detected (test on x=5 and y=6) :");
    foreach (var operation in Operations)
    {
        Console.WriteLine(operation.OperationName+" : "+operation.Compute(5,6));
    }
    Console.ReadLine();
}
```

Comme vous le constatez, il n'y aurait pas MEF que nous aurions écrit le même code, sauf que nous trouverions quelque part, à la place de l'appel à Compose que nous allons voir maintenant, un code qui aurait « rempli » la propriété Operations en créant des instances de classes supportant ICompute.

Le code de Compose

Ecrire un bout de code qui créée des instances de classes supportant ICompute et remplir la listeOperations (la propriété liste de notre application), c'est ce que fait la méthode Compose. Mais c'est en réalité c'est MEF qui va le faire pour nous, et mieux encore, il va aller chercher tout ça dans des DLL de plugins.

Regardons le code de Compose :

```
private void Compose()
{
    // création du catalogue. Les addins sont dans le répertoire de l'exe
    var catalog = new
DirectoryPartCatalog(System.Reflection.Assembly.GetExecutingAssembly().Location);
    // créé le conteneur
    var container = new CompositionContainer(catalog.CreateResolver());
    // ajoute this pour qu'il puisse être lié via ses attributs 'import'
    container.AddPart(this);
    // réalise la composition (connexion de tous les exports à tous les imports)
    container.Compose();
}
```

4 lignes. 4 lignes seulement pour :

1. créer un catalogue de plugins d'après un répertoire disque,
2. créer un conteneur qui est le point de fusion où se rencontre les exportateurs et les importateurs,
3. ajouter l'instance de notre application au conteneur (puisqu'elle est consommatrice via sa propriété marquée Import), d) demander au conteneur d'effectuer le binding entre tous les exportateurs et tous les importateurs. Incroyable non ?

Et c'est tout !

Ce qui va se passer au lancement de l'application est assez simple : MEF va balayer tout le répertoire indiqué dans le catalogue (qui peut contenir plusieurs répertoires), chercher les DLL et tester pour chacune celles qui exportent des éléments compatibles avec les importations. Qu'il s'agisse ici de notre application (consommatrice via son Import) ou même des plugins entre eux qui peuvent aussi avoir des relations consommateur / fournisseur.
C'est là que réside la magie de MEF qui va ensuite faire le nécessaire pour renseigner automatiquement toutes les propriétés ayant l'attribut Import. Dans notre cas MEF va détecter que deux DLL sont compatibles avec l'Import de notre application. Il va créer des instances des classes supportant l'interfaceICompute, en faire une liste qui sera placée dans la propriété Operations automatiquement.
Il ne reste plus à notre application qu'à utiliser la propriété Operations comme n'importe quelle autre propriété, sauf qu'ici son contenu a été créé par MEF automatiquement.

CONCLUSION

Simple et un peu magique, c'est ça MEF.

MEF règle un problème récurrent d'architecture, l'extensibilité, de façon simple et élégante. Bien entendu MEF permet de faire plus choses que ce que j'en montre ici. A vous de le découvrir en le téléchargeant et en étudiant les exemples et le début de documentation fourni !

MVVM, Unity, Prism, Inversion of Control…

Dans la série, presque culte, que je diffuse régulièrement et dont le titre serait "comment ne pas passer pour un idiot à la machine à café" la belle brochette de termes que je vous propose aujourd'hui a de quoi plonger le non initié dans une profonde perplexité !

Tous ces termes sont plus ou moins liés entre eux, les uns renvoyant aux autres ou les autres utilisant les uns.

MVVM OU PLUTOT M-V-VM

"Ah oui ! avec les tirets on comprend tout de suite mieux !" (Jean Sérien, lecteur assidu).

Model-View-ViewModel. Ca bégaye un peu mais c'est pourtant ça.

En réalité vous connaissez certainement le principe qui vient de M-V-C, Model-View-Controler, Modèle-Vue-Contrôleur. Une stratégie de développement visant à découpler le plus possible les données de leurs représentations. Sinon je vous laisse vous faire une idée, depuis dix ans au moins le Web est plein de pages présentant cette approche. Le billet du

jour n'est pas d'aller dans les détails, juste de planter le décor avant d'aller plus loin dans d'autres billets.

Alors pourquoi MVVM plutôt que MVC ? C'est en tout cas le premier terme qui devient de plus en plus à la mode dans le monde WPF / Silverlight. Une réponse un peu courte je l'avoue… Mais même Prism (j'en parle plus loin) dans sa version 2 et qui, entre autres choses, présente cette stratégie ne l'évoque pas sous ce nom mais sous diverses autres formes. Autant dire que terminologiquement parlant, savoir ce que veut dire MVVM est un must de l'instant !

Donc M-V-VM est un modèle de développement, une façon de structurer l'écriture du code dont le but ultime est de séparer le **Modèle** (les données) des **Vues** (l'IHM ou l'UI), le tout en passant par un intermédiaire, le **contrôleur** qui s'appelle dans cette variante **ViewModel** (et pour lequel je ne trouve pas de traduction vraiment limpide).
La vraie question, légitime est : *ce changement d'appellation cache-t-il un changement de rôle* ? Les puristes vous diront bien entendu que oui, sinon ils n'auraient pas inventé un nouveau nom… Les pragmatiques dont je suis vous diront que lorsqu'on veut relancer un produit un peu trop connu mais toujours valable, on change son nom ce qui permet de communiquer à nouveau dessus plus facilement… Les méthodologistes sont des futés… Certains vont jusqu'à parler de parenté avec le pattern Presentation-Model, un peu comme d'autres vous expliquent que C# vient de C++ plutôt que de dire qu'il vient de Delphi. S'inventer des origines plus ou moins nobles (ou du moins qui le paraissent) est un jeu finalement très humain qui nous intéresse que peu ici.
Mais pour être totalement juste j'ajouterais qu'il y a en fait une véritable **évolution de point de vue** entre M-V-C, dont on parlait déjà du temps de Delphi Win32 et C++, et M-V-VM qu'on applique à WPF ou Silverlight. Par force le temps passe, les outils se sophistiquent et les pensées s'aiguisent. Donc, bien sûr, il y a quelques variations entre ce qu'on entendait par "contrôleur" dans MVC et ce qu'on entend par ViewModel dans MVVM. Mais, à mon sens, cela est mineur. D'ailleurs de nombreux forums proposent des threads à rallonge où des tas d'experts (autoproclamés) se déchirent sur ces nuances. Dépassons ces débats stériles.
Pourquoi MVVM est-il en train de s'imposer sous WPF / Silverlight ? Simplement parce que "out-of-the-box" Visual Studio et Blend ne permettent pas directement de vraiment séparer le travail entre les développeurs et les designers. Volonté pourtant clairement affichée de ces produits. Les outils eux-mêmes mélangent un peu des deux compétences (un peu plus de design dans Blend, un peu plus de développement dans VS).

Mais lorsqu'il faut écrire de vraies applications avec WPF ou Silverlight et qu'on a de vraies équipes à gérer, c'est à dire des informaticiens d'un côté et un ou plusieurs infographistes de l'autre, il faut bien convenir que sans une approche appropriée ça coince. Non pas que techniquement les outils (Blend, VS) ne sont pas à la hauteur, ni même que les langages

(Xaml, C#, VB) ne sont pas bien étudiés. Au contraire. La faille est ici humaine et méthodologique.

En réalité tous les outils du monde, aussi bien faits soient-ils, ne peuvent permettre aux informaticiens l'économie d'une méthodologie bien pensée… Ce que Microsoft propose avec WPF / Silverlight, Xaml et le Framework .NET est très nouveau et il faut du temps pour maîtriser ces outils qui évoluent beaucoup en peu de temps. Dégager des stratégies de développement pertinentes est plus long que de créer un nouveau compilateur.

Or, si on regarde une fenêtre WPF ou une page Silverlight, on voit immédiatement qu'il existe du code-behind joint au fichier Xaml selon un modèle vieux de 15 ans (le couple de fichiers DFM/PAS des TForm de Delphi 1 en 1995 par exemple). La tentation est alors bien grande de coder l'action d'un click sur un bouton directement dans ce code puisqu'il est là pour ça ! Si c'est autorisé, c'est utilisé. Et malgré une séparation forte du code métier (ce qui suppose d'avoir fait cet effort) on se retrouve au final avec une interface qui contient du code qui en appelle d'autres (le code métier ou les données) voire pire, qui contient des petits bouts de traitement ! Autant dire que certains codes Silverlight ou WPF que j'ai audités récemment ressemblent comme deux gouttes d'eau au code spaghetti Delphi que j'ai audité dans les 15 dernières années… La vie n'est qu'un éternellement recommencement disent certains. Je n'ai pas cette vision nihiliste de l'existence, mais en informatique, hélas, cela se confirme !

Vade Retro, Satana !

En dehors de l'imbroglio qui résulte de cette situation, le code final de l'application est donc difficilement maintenable, intestable de façon cloisonnée (donc obligeant à des tests depuis l'interface par des testeurs ou des outils simulant un humain qui clique sur les boutons) et, comble du comble, il est en réalité presque impossible de séparer proprement le travail des informaticiens et des designers ! L'enfer brûlant s'ouvre alors sous les pieds des infortunés développeurs qui croyaient pourtant avoir "tout fait comme il faut". Bad luck. On est loin du compte.

M-V-VM permet ainsi de régler plusieurs problèmes d'un coup : meilleure maintenabilité du code, vraie séparation entre code et interface, tests unitaires du code possibles sans même qu'il existe encore d'interface (ou totalement indépendamment de celle-ci), et enfin **vraie séparation entre infographistes et informaticiens**.
En pratique le fichier de code-behind **ne sert plus à rien et reste vide**. Les Vues (interface utilisateur) utilisent leur DataContext pour se lier à une instance du ViewModel qui lui sait utiliser le Model (les données). Grâce à cette notion de DataContext et au puissant Data Binding de Xaml on peut mettre en place une séparation bien nette avec, au final, peu de code en plus.

Reste le problème des événements de l'UI comme le simple clic sur un bouton. Comment le programmer sans code-behind ? Sous WPF la réponse est simple puisqu'il existe une gestion des commandes complète (le CommandManager et d'autres classes). L'interface ICommand est la base de ce mécanisme, les contrôles qui ont des propriétés de ce type peuvent donc être bindées au code des actions se trouvant alors placé dans le ViewModel, comme n'importe quelle autre propriété peut être liée via le binding.

Ce modèle est d'une grande élégance il faut l'avouer. J'aime plaisanter et je joke souvent dans mes billets, mais il ne faudrait pas croire que je passerais du temps à écrire un long billet sur un sujet futile ou une simple réinvention stérile de vieux procédés. **M-V-VM est "la" façon de coder proprement sous WPF et Silverlight**. Il n'y en a pas d'autres (d'aussi abouties pour l'instant en tout cas). Tout le reste ne peut que créer du code spaghetti. Cela peut choquer, vous paraître extrême dit comme ça, mais pourtant c'est la vérité…
C'est pourquoi je vous proposerais bientôt un autre billet abordant par l'exemple la stratégie M-V-VM.

Mais pour l'instant revenons deux secondes sur cette fameuse gestion des commandes de WPF. Et Silverlight ? Hélas malgré l'enthousiasme que vous me connaissez pour ce dernier et sa convergence avec WPF, il faut reconnaître que cette dernière n'est pas terminée. On le dit souvent, Silverlight est un sous ensemble de WPF, même si ce n'est plus tout à fait vrai. On avance aussi que le plugin est trop petit pour contenir tout ce qui fait WPF. C'est vrai. Mais comme je le disais dans un autre billet, avec une installation personnalisée du Framework ce dernier peut tenir dans 30 Mo. C'est bien plus gros que le plugin SL actuel. Mais j'ai joué pas plus tard qu'hier avec une application écrite en Adobe Air, et justement le download de Air fait environ 30 Mo. Pour un résultat tellement navrant que j'ai tout désintallé d'ailleurs. Alors d'ici quelques versions de SL, peut-être Microsoft (et les utilisateurs) accepteront-ils l'idée qu'un produit aussi bien ficelé puisse justifier les 10 ou 20 Mo de plus à télécharger (une fois) pour avoir enfin un équivalent absolu de WPF… Mais ce n'est pas encore le cas, force est de le constater.

Donc Silverlight pour le moment propose ICommand, l'interface, mais pas tout le reste qui fait la gestion des commandes sous WPF. De fait, dans Silverlight il faut ajouter un peu de code pour recréer cette dernière. C'est le but d'un petit framework comme "MVVM Light" qui se base sur les principes édictés dans Prism. Mais on peut inventer d'autres solutions. Donc pour faire du M-V-VM avec WPF, tout est dans la boîte, avec Silverlight il faut ajouter une simulation, même minimaliste, de la gestion des commandes afin que ces dernières puissent être intégrées au ViewModel et que la Vue puisse être liée aux actions du ViewModel par un binding sur des propriétés de type ICommand.

PRISM

Cela nous amène à évoquer Prism "patterns and practices : Composite WPF and Silverlight".

Il s'agit d'une réflexion méthodologique dont les fruits se présentent sous la forme de conseils et d'applications exemples. Prism couvre de nombreux champs et propose des solutions élégantes.

Cela ne serait pas fairplay d'en parler en quelques lignes, c'est un travail d'une grande qualité qui ne se résume pas en trois phrases. Je vous conseille vivement sa lecture, rien ne peut remplacer cet investissement personnel.

Mais puisqu'aujourd'hui le but est de planter le décor, il aurait été tout aussi impardonnable de "zapper" Prism, surtout que certaines solutions permettant d'implémenter M-V-VM sous Silverlight y sont présentées.

Je reviendrai certainement un jour sur Prism, c'est d'une telle richesse que j'ai vraiment envie de vous en parler plus. Prism est incontournable, mais ne peut se résumer dans un billet. Si je trouve le moyen d'un tel tour de force alors je vous en reparlerai en détails. Mais le meilleur conseil c'est encore que vous le lisiez vous-mêmes… [Nota : à la date de publication de ce PDF il existe aussi une version spéciale de Prism pour WinRT, voir la présentation en fin du présent document]

UNITY APPLICATION BLOCK

Unity est un framework Microsoft qu'on trouve dans la série "patterns & practices". Une mine d'or de la méthodologie sous .NET d'où provient aussi Prism dont je parlais ci-dessus. Il faut noter que MS fait d'immenses efforts pour ouvrir la réflexion sur les méthodologies tout en tentant le tour de force de rendre ces cogitations tangibles. Lorsqu'on connaît un peu les méthodologistes pour les avoir fréquentés, toujours perdus dans leurs hautes sphères et rebutant à se "salir" les mains avec du code, on imagine à quel point l'effort de Microsoft est louable et tellement remarquable que cela mérite d'insister un peu sur la valeur de patterns & practices.

Ainsi, Unity, tout comme Prism, ne se contente pas d'être un gros pavé d'explications qui seraient réservées à une certaine élite. Unity se sont bien entendu des explications mais aussi du code qui permet d'ajouter l'action à la réflexion. Du bon code même. Des librairies utilisables pour simplifier le développement d'applications qui se conforment aux best practices. Je n'irais pas jusqu'à dire que les travaux présentés dans patterns & practices sont d'abord aisés, les cogitations méthodologiques réclament toujours un minimum d'attention, mais la volonté assumée de joindre la pratique et l'usage à la réflexion rend tout cela accessible à qui se donne un peu de temps pour lire et expérimenter.

Pour ce qui est de Unity, ce bloc se concentre sur la conception d'un conteneur "léger" et extensible permettant de faire de l'**injection de dépendances** (dependency injection).

De quoi s'agit-il ?

Comme indiqué en introduction de ce billet, tout ce dont je vous parle aujourd'hui tourne autour des mêmes besoins, de concepts proches ou qui se font écho les uns aux autres. L'injection de dépendances est une réponse à une série de problèmes qui se posent lorsqu'on désire séparer proprement des blocs de code tout en évitant de construire une "usine à gaz".

Séparation de code, M-V-VM, Prism, ce sont autant de caméras placées autour d'une même scène. Des angles de vue différents, des manières d'entrer dans le sujet différentes, mais au final un but qui, s'il n'est pas commun, vise une même finalité. La conception d'applications souples, maintenables et extensibles.

Grâce à Unity vous disposez à la fois d'explications claires et d'une librairie permettant de gérer convenablement les injections de dépendances. Encore une terminologie "savante" pour des choses finalement pas si compliquées que cela. Mais pour en rajouter une couche disons que l'injection de dépendances n'est qu'une façon de gérer l' "**inversion de contrôle**".

Inversion of Control

"Pitié ! Arrêtez-le !" (Jean Peuplus, lecteur parisien)

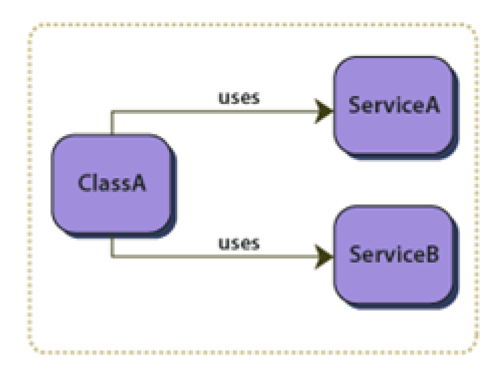
On peut comprendre ce lecteur. Une telle avalanche ne risque-t-elle pas de tout emporter sur son passage, au risque de rendre ce billet contre-productif ?

La question mérite d'être posée, mais je fais le pari que vous tiendrez jusqu'au bout (qui est proche je vous rassure).

Le problème à résoudre

Pour info : j'utilise ci-après des parties de la documentation de Prism version octobre 2009

Vous avez une classe dont le fonctionnement dépend de services ou de composants dont les implémentations réelles sont spécifiées lors du design de l'application.



Les problèmes suivants se posent

- Pour remplacer ou mettre à jour les dépendances vous devez faire des changements dans le code de la classe (classe A dans le schéma ci-dessus)
- Les implémentations des services doivent exister et être disponibles à la compilation de la classe
- La classe est difficile à tester de façon isolée car elle a des références “en dur” vers les services. Cela signifie que ces dépendances ne peuvent être remplacées par des mocks ou des stubs (maquette d’interface ou squelette non fonctionnel de code).
- La classe contient du code répétitif pour créer, localiser et gérer ses dépendances.

Les conditions qui forcent à utiliser le pattern d’inversion de contrôle

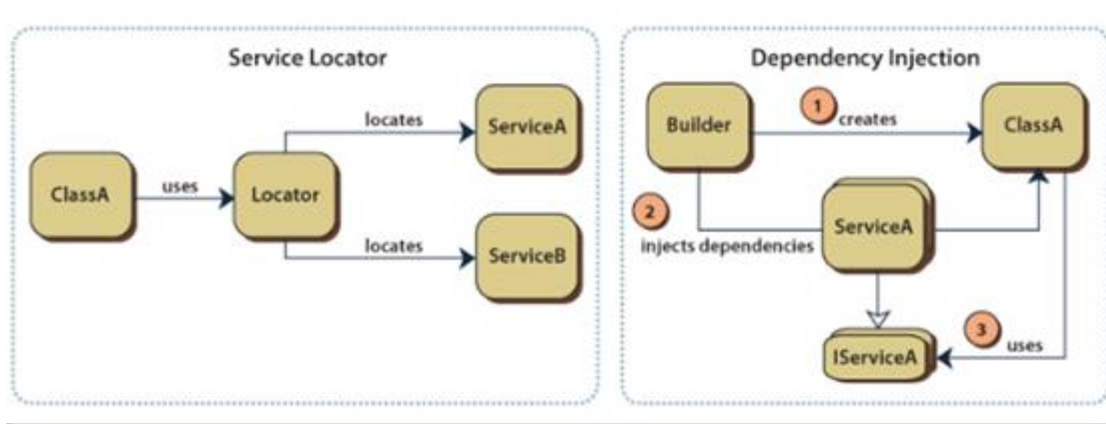
- Vous voulez découpler vos classes de leurs dépendances de telle façon à ce que ces dernières puissent être remplacées ou mises à jour avec le minimum de changement dans votre code, voire aucune modification.
- Vous désirez construire vos classes de telle façon à ce qu’elles puissent dépendre d’implémentations inconnues ou indisponibles lors de la compilation.
- Vous voulez pouvoir tester vos classes de façon isolée, sans faire usage des dépendances.
- Vous voulez découpler la responsabilité de la localisation et de la gestion des dépendances de vos classes.

La solution

Il faut déléguer la fonction qui sélectionne le type de l’implémentation concrète des classes de services (les dépendances) à un composant ou une source externe à votre code.

Les implémentations

Il y a plusieurs façons d’implémenter la pattern d’inversion de contrôle. Prism en utilise deux qui sont l’Injection de dépendances (Dependency Injection) et le Localisateur de service (Service Locator).



Injection de dépendances

L'injection de dépendances est un mécanisme qui permet d'implanter le principe de l'Inversion de contrôle. Il consiste à créer dynamiquement (injecter) les dépendances entre les différentes classes en s'appuyant généralement sur une description (fichier de configuration). Ainsi les dépendances entre composants logiciels ne sont plus exprimée dans le code de manière statique mais déterminées dynamiquement à l'exécution.

Le localisateur de services

Le localisateur de services est un mécanisme qui recense les dépendances (les services) et qui encapsule la logique permettant de les retrouver. Le localisateur de services n'instancie pas les services, il ne fait que les trouver, généralement à partir d'une clé (qui éviter les références aux classes réelles et permet ainsi de modifier les implémentations réelles). Le localisateur de services propose un procédé d'enregistrement qui liste les services disponibles ainsi qu'un service de recherche utilisé par les classes devant utiliser les dépendances et qui se chargent de les instancier.

Exemples

Même si les choses sont certainement plus claires maintenant (je l'espère en tout cas) il est certain que des exemples de code aideraient à visualiser tout cela. J'en ai bien conscience et j'ai prévu d'illustrer concrètement les principes expliqués dans ce billet. Toutefois en raison de la longueur de chaque exemple (nous parlons ici de techniques de découplement donc de nombreux fichiers et projets pour un seul exemple) il ne me semble pas raisonnable d'aller plus loin immédiatement. Laissons ce billet se terminer, et toutes ces notions tourner dans vos têtes avant de les appliquer !

CONCLUSION

Une terminologie ésotérique ne cache pas forcément des choses complexes. Le but de ce billet était de vous donner quelques clés pour mieux comprendre un ensemble de notions différentes mais participant toutes à une même finalité.

MVVMCross, Okra... Mvvm, WinRT et cross-plateforme Android/Apple

MVVM est un sujet que j'aime bien. Je vous ai présenté sous forme de livres PDF les frameworks MVVM Light et Jounce, bientôt je vous présenterai deux autres frameworks. Mais leur intérêt est tel que leur nom mérite d'être connu au plus vite.

OKRA

C'est le nouveau nom du projet Cocoon sur Codeplex (http://okra.codeplex.com/). Il s'agit d'un framework MVVM que j'ai déjà évoqué ici destiné uniquement au développement sous WinRT.
Comme tous les abonnés MSDN possèdent depuis le 15 aout à la fois la RTM de Windows 8 et celle de Visual Studio 2012, je suis certain qu'il est temps pour les plus avancés d'entre eux de télécharger Okra pour le prendre en main.

J'ai un papier en cours sur Okra, pour les moins pressés et aussi pour ceux qui seront contents d'avoir mes explications en plus du code brut et des docs en anglais. Ca vient, c'est planifié pour septembre [un an après c'est la première promesse de Dot.Blog non tenue, mais je n'y renonce pas !].

PRISM FOR WINDOWS STORE APPS

Absent du billet original je suis obligé d'en indiquer l'existence lors de cette révision (oct.2013). En dehors du fait que j'ai participé à son élaboration, Prism pour WinRT est un excellent framework MVVM pour les applications de type Windows Store. Il n'est pas cross-plateforme c'est son principal défaut et très différent de Prism pour WPF. Prism pour WinRT est présenté dans ce recueil PDF (voir le sommaire).

MVVMCROSS

Là on touche du lourd... Pas tant le framework lui-même qui est jeune et pas si énorme que ça, mais l'intérêt du sujet et de ce qui gravite autour.

Apple, Android, Windows Phone, WinRT

Je m'explique.

Comme je le disais il y a quelques temps, quel que soit ma fidélité à Windows et encore plus à Windows 8 force est de constater qu'en Europe, Android représente 68,1% des ventes de Smartphones et que côté tablettes, l'Ipad domine encore bien le marché.

Il faudrait être fou ou stupide pour ignorer cette présence et tous les besoins qui naissent autour de ces machines.

Personnellement je déteste Apple sans m'en cacher (la preuve !), et leur perte de vitesse inéluctable déjà sur les Smartphones m'amuse beaucoup. Je l'avais annoncée car cela ne pouvait se passer que de cette façon. Je ne pousserai donc personne à développer sous iOS. Mais en professionnel je sais que mon désamour d'Apple, même s'il a de solides raisons que je pourrais étayer, n'a pas à prendre le dessus sur la réalité de certains marchés où développer pour l'Ipad peut être une obligation.

Donc le développeur doit aujourd'hui pouvoir traiter toutes ces plateformes. Etre un expert en "tout" est impossible.

En revanche être bon en C#, en Xaml et savoir utiliser les bons outils, c'est à la portée de tous ceux qui n'ont pas peur de travailler un peu plus (j'en ai entendu quelques un dire "pour gagner moins".. tssss mauvais esprits va !).

Or, comment développer pour toutes ces plateformes si différentes sans tout recoder, sans apprendre des tas d'environnements plus ou moins pauvres (comparativement à ceux de MS) ?

MONOTOUCH ET MONODROID

Je suis devenu un fan de MonoDroid de Xamarin. Un produit pas cher qui permet de développer en C# sur Android (de façon native bien entendu).
Pour le monde Apple il existe un produit frère, MonoTouch, qui fonctionne de façon identique mais qui créé des applications iOS pour l'IPhone et l'Ipad.
Grâce à ces merveilleux produits, il est possible d'**utiliser Visual Studio pour développer sous Android ou iOS**. C'est vraiment une avancée incroyable pour tous les développeurs C# !
Vous allez me dire, et au fait c'est quoi le rapport entre tout ce discours et MVVMCross ?

MVVMCROSS : DEVELOPPEZ UNE FOIS POUR TOUTES LES PLATEFORMES

MVVMCross est une framework MVVM. Jusque là, un de plus me direz vous...

Oui, mais pas un framework MVVM comme les autres.

MVVMCross a été conçu pour permettre le **développement cross-plateforme entre toutes les plateformes en tirant partie justement de l'existence de MonoTouch et MonoDroid pour couvrir le monde Apple et celui d'Android**...
En gros, car je prépare aussi un papier sur ce framework où j'entrerai dans les détails, **avec MVVMCross vous écrivez une seule fois le code de vos Modèles et ViewModels et vous n'avez plus qu'à créer les Views pour chaque plateforme**.

Plus fort encore, MVVMCross ajoute, avec beaucoup d'intelligence, la notion de **Binding sous Android et iOS** ! De fait, une fois le code commun écrit (modèles et ViewModels) il ne reste plus qu'à faire des vues qui utilisent une extension (comme on peut le faire en xaml) pour indiquer les Bindings sur le ViewModel. On peut programmer les Vues sans une ligne de code C# dans de nombreux cas...
Aussi bien pour WinRT, Windows Phone, Android et iOS sur les smartphones et les tablettes.
Si vous ne trouvez pas ça génial, c'est que vous êtes blasés (ou que vous êtes pro Apple)

CONCLUSION

La nature a horreur du vide nous dit la sagesse populaire. La nature de l'homme certainement. Et face à l'éclatement des standards et des plateformes à prendre en compte, il fallait des réponses sérieuses. Les bricolages de "l'hybride" sont des horreurs et HTML 5 n'est pas un standard de développement natif (d'ailleurs depuis qu'il a forké on ne sait même plus de quel HTML 5 on parle...).

Microsoft nous offre dans les mois à venir sa conception du cross-plateforme : tout en WinRT du smartphone au PC. C'est une vision brillante et cohérente, mais tout en Microsoft...

Le monde est plus vaste et quel que soit notre passion pour les produits de Microsoft professionnellement nous ne pouvons ignorer les mastodontes que sont Apple et Android sur le marché. Ce serait de l'autisme.

Grâce à quelques "pionniers" nous disposons aujourd'hui des moyens de développer facilement des applications cross-plateformes.

Que cela soit MonoDroid et MonoTouch ou des idées géniales comme MVVMCross qui tire partie de l'existence des premiers, une voie se dessine enfin pour nous permettre d'envisager l'avenir de façon plus sereine sans faire l'impasse sur telle ou telle autre plateforme.

En fait, grâce à C#.

Alors ce n'est pas demain que vous me verrez faire un papier sur Objective-C ni sur JavaScript ou HTML 5 !

En revanche, bientôt vous en saurez plus sur Okra et MVVMCross.

MVVM, je m'y perds ! (une réponse rapide)

MVVM est un pattern, un simple pattern, pas une technologie nouvelle. Elle est utilisable dans de nombreux contextes, sous ce nom ou un autre et sous des formes plus ou moins identiques. Rien de nouveau donc. Et pourtant tout change. "Je m'y perds" est une réflexion que j'entends souvent. Encore aujourd'hui, un lecteur de Dot.Blog me faisait part de ce sentiment d'être un peu "embrouillé". Alors plutôt qu'une réponse en privé qui ne profiterait qu'à un seul, voici la réponse à ce lecteur, et bien sûr, à tous les autres qui partagent la même impression…

Renvoi implicite : je renvoie bien entendu le lecteur à tous les billets portant sur MVVM et notamment aux derniers gros articles qui abordent en profondeur MVVM. Ici il ne s'agit que de répondre globalement à un sentiment. Les réponses techniques se trouvent déjà en ligne dans le Blog.

MVVM ÇA CHANGE QUOI AU FINAL ?

Tout le problème consiste bien entendu en une question de « point de vue », techniquement MVVM ne fait qu'utiliser des choses « normales ».

C'est donc la façon de segmenter le travail qui change.

Dans une application WinForms « bien faite » on avait les couches suivantes : le **DAL** (data access layer, la couche d'objet en relation avec la base de données et objectivant les infos remontant de celle-ci et s'occupant de faire les requêtes, d'utiliser les procédures stockées, etc), le **BOL** (business objet layer, couche d'objets métier, c'est ici qu'apparaissent les concepts de l'application par exemple la notion de client, de facture… le BOL repose sur le DAL pour l'accès aux données) et l'interface Windows Forms (c'est un exemple). Dans cette dernière on gérait en code, accroché au visuel d'une fiche, toute la logique d'affichage en utilisant les objets du BOL qui eux utilisaient donc le DAL pour lire les données ou les persister.

C'est un schéma assez classique et qui connaissait des variantes, par exemple l'utilisation d'une couche de persistance objet entre le BOL et la base de données au lieu d'un DAL écrit à la main, ou même des générateurs de DAL.

Bref c'est un découpage déjà complexe dans certains cas et qui laissait se concentrer une partie de "l'intelligence" dans la fiche, dans son "code-behind", cela était même conçu pour.

On peut résumer le schéma comme cela : DAL<->BOL<->WinForm.

La Form ne dialoguait jamais directement avec le DAL (hmmm…) de même que le DAL n'affichait rien (re-hmmmm !). En tout cas les prémices d'une certaine "rigidité" étaient posées même s'ils n'étaient pas toujours parfaitement respectés. De même le code-behind devait se limiter à appeler des méthodes des objets du BOL, même si souvent on voyait des

méthodes de Click de bouton pleines de code fonctionnel. C'est ce qu'on appelle le code spaghetti dont j'ai parlé ici quelques fois.

LE DECOUPAGE A LA MVVM

Avec MVVM on fait un découpage assez proche mais différent :

Le **DAL** ou la couche de persistance objet *existe toujours*, ce sont les sources de données objectivées, sous MVVM on les appelle les **Modèles**. C'est juste une question de nom.
La couche **BOL** existe toujours, il faut bien des objets métiers où les concepts de l'application apparaissent (facture, article, liste d'impayés…). Sous MVVM *on intègre cette couche à la notion de « Modèle »*. Les objets du BOL ne sont finalement que des données (même si elles sont plus complexes que celles du DAL).
Donc DAL et BOL *existent toujours*, sous MVVM ce sont des données, donc des Modèles.
La WinForm, la fiche, la surface d'affichage, peu importe la technologie, porte un nom particulier sous MVVM : la **Vue**.
La petite nuance c'est que si sous WinForm on écrivait beaucoup de code dans le « code behind » de la fiche, sous MVVM c'est quasi interdit (sauf si cela ne concerne que l'affichage).

Du coup ce code il faut bien l'extraire et le placer quelque part… C'est là qu'apparait le Modèle de Vue, **ViewModel**.
Résumé : du schéma
DAL<->BOL<->WinForm on passe au schéma Modèle<->ViewModel<->Vue.

Sachant que le Modèle reprend en gros le groupe DAL<->BOL et que le ViewModel reprend à peu près ce qui se trouvait dans le code-behind des fiches sous WinForms.

La façon de programmer est en réalité la même, c'est juste la place du code et le nom du groupe auquel il appartient qui changent.
La gestion des données, lecture et écriture se fait dans des Modèles qui ne sont que des ensemble DAL/BOL (le DAL discutant avec un SGBD ou des services, peu importe).

L'affichage se fait toujours dans la fiche, sauf qu'on l'appelle la Vue.

Et le code de « bricolage » qui permet « d'adapter » les données des Modèles pour la Vue ou de persister les états de la Vue, au lieu d'être dans le code-behind de celle-ci, il se trouve à part dans un code appelé ViewModel.

Pour conclure : tout ce qui est affichage est mis en page dans des Vues. Tout ce qui concerne les données et les objets métier se trouvent dans des Modèles, et tout ce qui permet d'adapter ces données aux Vues se trouve dans des ViewModel. Ce qui fait le lien

entre données (Model) et affichage (View) c'est le modèle de vue (ViewModel) et ce lien entre View est ViewModel est basé sur le Data Binding XAML.

LES COMPLICATIONS…

Ce petit changement de point de vue qu'introduit MVVM a malgré tout des implications dans la façon d'écrire le code. On ne peut pas se contenter simplement par un couper/coller de déplacer le code du code-behind d'une WinForm pour le placer dans un ViewModel WPF ou Silverlight…

La première complication réside dans le fait que l'ancienne méthode utilisait un principe « étudié pour » pour la gestion des commandes : on place un bouton dans une WinForm on double-clic dessus et automatiquement VS créé un gestionnaire d'événement du Click dans le code-béhind de la fiche.

Avec MVVM, comme ce code est forcément dans le ViewModel, on « s'enquiquine » un peu en le déplaçant : il faut que le ViewModel expose des propriétés de type **ICommand** et que les éléments d'interface se lient par Data binding à ces *commandes transformées en propriétés*.
Hélas ni WPF ni Silverlight (et encore moins ASP.NET ou Windows Forms) n'ont été conçus pour cela. Il a des ajouts récents comme ICommand ou le **Command Manager** de WPF qui permettent de régler en partie le problème, seulement en partie. Et encore ce n'est pas évident. Par exemple la classe Button expose une propriété Command aujourd'hui. On peut donc la lier par data binding facilement avec une propriété de type ICommand du ViewModel. Mais comment gérer le changement de valeur d'un Slider par exemple ? Cela n'est pas pris en compte tout simplement…
Du coup, et principalement au départ pour la gestion de commande, il faut utiliser des toolkits qui simplifient un peu les choses. **MVVM Light** se charge de le faire par exemple. **Prism** aussi, en plus complexe, et d'autres toolkits d'ailleurs.

LA BARRIERE DE LA COMMUNICATION

"Avant" on s'autorisait un peu tout… et puis comme beaucoup de code était dans le code-behind des fiches, on avait finalement peu d'états d'âme. Le code spaghetti n'était jamais loin non plus… Et du code même très récent que je peux auditer reste écrit globalement de la même façon qu'il y a 15 ans en "client/serveur" avec des langages comme Delphi ou C++. Pour ceux qui travaillent encore aujourd'hui de cette façon il est certain que le pas à sauter vers la rigueur qu'impose MVVM peut s'avérer un numéro d'équilibriste !

Après avoir gérer la complication introduite dans la gestion des commandes on s'aperçoit assez vite qu'un ViewModel ne pouvant rien afficher (interdiction de même nature que celle qui interdit à la Vue de contenir du code fonctionnel), cela complique encore plus les choses

car là il n'y a vraiment rien de prévu, pas même un début de solution comme ICommand pour les commandes.

C'est ici que les toolkits deviennent indispensables pour s'éviter de réinventer la roue, car pour faire communiquer des ViewModels entre eux, ou des Modèles vers des Vue, ou dans le sens contraire des Vue vers les Modèles (même si ce n'est pas usuel cela peut être utile), pour tout cela il n'y a qu'une solution vraiment bonne : la gestion d'une **messagerie**. Il y a des émetteurs de message (tout objet de n'importe quelle partie en a le droit) et il y a des objets qui s'abonnent à des messages particuliers (idem, ces objets peuvent appartenir à n'importe quel bloc). Du coup, le découplage entre Modèle, ViewModel et View n'est pas violé et sans s'interdire certaines communications notamment transversales ou à « rebrousse-poil ».

ET LE RESTE…

Bien entendu, à tout cela s'ajoute l'asynchronisme, le problème des UI qui ne peuvent être mises à jour que par le Thread principal, la nécessité de gérer du parallélisme (avec PLINQ par exemple), l'obligation aujourd'hui de designer les interfaces, gérer les animations, la cohérence des états, etc… Tout cela complique un peu le tableau final et n'est pas directement lié à MVVM mais prend une « saveur » particulière lorsqu'on utilise cette pattern…

CONCLUSION

MVVM ne change rien, et MVVM change tout… La seule façon d'intégrer cette nouvelle façon de développer est encore de la pratiquer. Conseil d'une étonnante banalité s'il en est. Mais je ne vais pas jouer les philosophes et vous expliquer le sens de la vie : c'est en buchant qu'on devient bucheron et c'est en sciant que Léonard de Vinci ! (elle n'est vraiment pas terrible mais il fallait bien finir sur une note de gaité !).

MVVM est-il un vrai pattern ?

Il y a déjà quelques temps j'ouvrais le débat "Faut-il bruler MVVM ?" où j'expliquais qu'il y a peut-être matière à réflexion autour d'une réforme de cette pattern qui ne tient pas toutes ses promesses. Aujourd'hui je vais un cran plus loin en affirmant que MVVM n'est pas un pattern.

MASOCHISME ?

D'abord, vous risquez de penser que je suis masochiste. C'est vrai, pourquoi m'acharner à utiliser un pattern que je trouve si imparfaite ? Pourquoi ne pas lâcher l'affaire et passer à autre chose ? Il y a tellement de patterns du même type...

Alors je dois réaffirmer que MVVM contient de nombreuses bonnes idées qui sont des avancées difficiles à renier, que MVVM est récente [le billet original est ancien rappelez-

vous !] et conçue dans l'esprit XAML et que revenir en arrière en l'abandonnant totalement serait idiot.

Quant à utiliser "autre chose", je ne connais rien du même type qui soit parfait. MVP (créée en 96 par Mike Potel pour C++ et IBM Java), VP et toutes leurs cousines reposent sur des principes proches mais pour d'anciens systèmes et, à l'usage, s'avèrent aussi poser des problèmes. Je ne parle même pas de patterns comme MVC (issue de Smalltak dans les années 70 !) qui finissent divisés en sous-branches incompréhensibles (MVC a été "revu" par Martin Fowler et dérivé en "Supervising Controler" et en "Passive View", patterns bien difficiles à différencier).

Dans le précédent billet que je cite plus haut, je souhaitais ouvrir un débat, une discussion large sur les faiblesses de MVVM dans le but d'initier une "réforme", une évolution du pattern. C'est toujours dans ce même esprit que je m'exprime aujourd'hui.

MVVM est "la bonne idée", en tout cas le bon noyau fondateur, moderne, mais il lui manque des choses.

QU'CE QU'UN PATTERN ?

Une "design pattern" ou "patron de conception" est un concept de génie logiciel hérité de l'architecture. C'est en effet un architecte viennois (mais ayant vécu en Angleterre ce qu'on oublie souvent), Christopher Alexander qui dans les années 70 eut l'idée d'écrire un livre qui se composait de "recettes" applicables en architecture. Chaque pattern étant présenté sous une forme bien précise : son nom, le problème à traiter, la solution détaillée (éléments en jeu, leurs relations et responsabilités) et les conséquences... De la poigné de porte à une ville entière, Alexander définissait ainsi un catalogue de solution éprouvées ("*A Pattern Language : Towns, Buildings, Construction*").
En informatique l'idée sera reprise en 1995 par le célèbre "Gang of Four" ou GoF (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) dans leur livre "*Design Patterns – Elements of Reusable Object-Oriented Software*".

Un design pattern est un formalisme. On trouve d'autres segmentations que celles d'Alexander comme "Intention, Motivation, Solution, Conséquences". Mais l'idée reste la même, le formalisme de base reste cohérent.
Chaque "design pattern" décrit ainsi un problème spécifique mais qui se rencontre couramment dans l'écriture d'un logiciel. Après avoir décrit le problème et son contexte, les auteurs décrivent une solution fiable **aux conséquences connues** (par exemple sur la vitesse d'exécution, la consommation mémoire). Le but étant de décrire chaque problème et sa solution de telle façon "*que l'on puisse réutilisation cette solution des millions de fois, sans jamais le faire deux fois de la même manière*".

Les design patterns sont des condensés de savoir et d'expérience. Le GoF était constitué d'experts, des gens qui écrivaient de vrais logiciels et qui avaient atteint une maitrise certaine de leur art. Ce n'était pas des théoriciens fous, des méthodologistes universitaires enfermés dans leur tour d'ivoire.
Les designs patterns sont le fruit d'une réflexion poussée autour d'une expérience. Il ne s'agit pas surtout de "ballons d'essai théorique" lancés en l'air pour '"voir ce que cela va donner" ! Au lieu de partir d'une théorie, d'une méthode de chercheur et d'essayer d'en trouver des applications pratiques, avec les design patterns ont part de l'expérience, enrichie par plusieurs vécus différents, et on tente de **formaliser ce savoir accumulé**. C'est totalement différent. Dans un cas on est dans l'essayisme pur et dangereux, dans le second on avance en terre connue et balisée par le pattern.
Je dirais même que les design patterns, que ce sont ensuite réappropriés les théoriciens et autres méthodologistes, sont en quelque sorte la revanche des ingénieurs sur ces derniers. Les design patterns prouvent que dans notre métier, comme en architecture, il est bien préférable, pour créer de bons logiciels, de formaliser son expérience que de vouloir imposer des théories non éprouvées par les faits issues de chercheurs qui jamais n'ont écrits le moindre logiciel...

Donc voici ce qu'est un design pattern : un sirop, un extrait, un élixir d'expérience formalisée aux conséquences mesurées, assumées, et commentées.

LES CONSEQUENCES FONT LE PATTERN

En fait, on pourrait en arriver à définir les design patterns par la fin : c'est bien plus la mise en évidence claire des conséquences positives et négatives de leur application qui en fait des patterns réutilisables que l'exposé de la solution au problème !

Des solutions aux problèmes, interrogez 100 développeurs et vous obtiendrez 100 réponses différences.

Nous ne sommes pas à la recherche de solutions.
Nous sommes à la recherche d'expérience !
Et ce qui caractérise un design pattern est bien la partie "**conséquence**" de sa présentation. C'est ce qui **prouve qu'elle est issue de l'expérience** et qu'on peut prévoir à quoi s'attendre en l'appliquant.
Sans cela, un design pattern n'est qu'une solution parmi des centaines. Pire, qu'une "bonne idée" qu'il reste à tester avant d'en faire un pattern, donc dangereux à utiliser...

C'est l'expérience véhiculée par les conséquences clairement énoncées qui font toute la valeur d'un design pattern.
Bien entendu, l'élégance de la solution proposée compte aussi. Mais sans le retour d'expérience qui s'exprime par la connaissance des conséquences de son utilisation, cela

resterait qu'une solution de plus. Des solutions élégantes il est toujours possible d'en trouver plusieurs pour un même problème. Mais qui soient validées par l'expérience de leur mise en pratique, il y en a moins, et ce sont des design patterns dès que tout cela est formalisé sérieusement.

MVVM PEUT-ELLE ETRE CONSIDEREE COMME UN DESIGN PATTERN ?

Non.

Elle ne se présent pas comme un design pattern du GoF par exemple. Et notamment **il lui manque l'essentiel : les conséquences connues et prévisibles de son application**.
MVVM est juste une entrée de blog... Une bonne idée présentée par John Gossman le 8 octobre 2005 dans l'un de ses billet intitulé "*Tales Form The Smart Client*".

MVVM ouvre des voies, assène des lois, fait la publicité de tout ce qu'elle apporte de bon, mais jamais MVVM n'explique les conséquences de son utilisation. Jamais elle ne permet d'avoir cette assurance qu'un vrai design pattern apporte.

MVVM est une bonne idée lancée en l'air, mais **elle n'est pas présentée comme doit l'être un design pattern**, **elle n'en suit pas le formalisme** et **elle ne se base pas sur une longue expérience qu'on tente de formaliser**. Elle née presque en même temps que les techniques qu'elle utilise, comme une **guideline**. Mais pas comme un pattern riche d'un retour d'expérience.
De fait, MVVM c'est juste une "bonne idée" qui est lancée en l'air. **Chacun essayant de l'appliquer à sa façon**. Avec plus ou moins de bonheur.

Prism, Caliburn, MVVM Light, Jounce, et bien d'autres toolkits ou frameworks tournent plus ou moins directement autour de l'idée de MVVM. Aucun ne règle tous les problèmes et souvent en pose de nouveaux qui restent sans solution.

Appliquer MVVM oblige à choisir un toolkit, et avec lui ses inévitables faiblesses. Beaucoup de ces toolkits sont d'ailleurs l'œuvre d'un seul homme (MVVM Light, Jounce, Caliburn...) qui, aussi compétent soit-il, fabrique son toolkit personnel en fonction de sa propre expérience puis tente, en fonction des retours (les siens et de ses utilisateurs) de combler les manques. C'est ainsi que le concepteur de Caliburn a décidé de tout reprendre à zéro et de concevoir Caliburn.Micro, non pas comme une version minimaliste mais bien en remplacement de Caliburn. *On avance par petit sauts, on bricole chacun dans son coin, on recommence*. **C'est beau, cela s'appelle s'expérience et c'est justement ce qui manque à MVVM pour être un vrai pattern...**

Mais quand je prends un design pattern du GoF, je n'ai pas tous ces aléas... J'ai du solide, du prévisible, toujours valable presque 20 ans après la sortie de leur livre ! **C'est ça un design pattern : quelque chose de solide et de fixe, un moule sur lequel on peut compter**, tout en

ayant la liberté de réaliser autant de versions différentes selon les besoins. Mais jamais on ne remet en question le moule lui-même, le design pattern.

Je ne voudrais pas donner l'impression de "sacraliser" les design patterns du GoF. Rien n'est sacré, pas même le sacré. Je suis un libre penseur et aucune loi, ni divine ni humaine ne s'impose à moi sans que je m'octroie le droit légitime et absolu de l'examiner, de la soupeser, d'en vérifier la validité.

Les design patterns du GoF ne sont donc pas sacrés. Ils sont juste bien faits et je peux compter sur eux !

MVVM n'est pas de cette nature.

MVVM est juste une "bonne idée", ou plutôt une liste de "bonnes idées". Une direction intéressante à suivre, une réflexion empreinte de bon sens. Mais comme toutes les "bonnes idées", il ne s'agit pas d'un "package complet" utilisable "tel quel", **juste un guide qui doit forcer à raisonner, à s'interroger**.
MVVM est "l'idée d'un pattern" qu'il reste à créer.

QUELLES CONSEQUENCES A CE CHANGEMENT DE STATUT ?

En remettant MVVM à sa place, en la délogeant du piédestal du statut de design pattern, on ne tue pas les bonnes idées qu'il véhicule. Dans ce billet-ci non plus je ne brulerais pas MVVM !

En revanche je le remets à sa place : de bonnes idées qui nécessitent encore beaucoup de travail et d'expérience avant d'atteindre le niveau de ce que doit être un vrai design pattern.

C'est une invitation au travail, à la réflexion. Ce n'est pas un discours nihiliste, d'aucune manière.
MVVM mérite notre attention. Mais on ne peut le considérer comme un design pattern "fini". Juste "l'idée de ce que pourrait être un pattern de ce type".

Il pose trop de questions sans réponses, il laisse **le champ libre à trop d'interprétations personnelles**, à **trop de frameworks ou de toolkits** radicalement différents qui chacun ne **répond que partiellement aux attentes**.

Prenons un simple exemple : MVVM a été pensé par un fondateur de WPF. MVVM fait donc usage du data binding et de tout ce qui caractérisait à l'époque WPF. Null doute que si MVVM avait été énoncé hier, il prendrait en compte la blendabilité. Sur l'ensemble des toolkits cités dans ce billet, seul MVVM Light prend ce point essentiel en compte (avec Jounce mais d'une autre façon).

Mais pour y arriver MVVM Light utilise un *locator* avec des propriétés statiques. Impossible d'utiliser MVVM Light et sa blendabilité avec MEF par exemple...

Est-ce la faute de MVVM Light de ne pas offrir de solution à ce problème, est-ce la faute de MEF de ne rien prévoir pour la blendabilité ? Ou bien est-ce la faute au "pattern" MVVM ne n'être pas "fini", pas complet, de manquer de recul, de n'être pas capable de suivre les évolutions du monde qui l'a vu naitre ?

Si on considère que MVVM n'est qu'une bonne idée, qu'une voie à suivre et à étudier, alors on comprend mieux la situation : chacun tâtonne, essaye de donner "son" interprétation de ce qu'il comprend de l'essence de MVVM, mettant l'accent sur ce qui le gêne au quotidien, ignorant superbement les autres problèmes posés par l'application de ce "pattern". Prism pour WPF et Prism pour WinRT sont totalement différents, n'est-ce pas troublant, voire dérangeant ?

Il reste beaucoup de travail avant de faire de MVMV un design pattern.

Les conséquences de cet état fait sont évidentes : Appliquer MVVM comme un pattern est une erreur. Croire que tel ou tel "écart" en "viole" les lois est stupide car on ne viole pas une "bonne idée", on ne viole qu'un pattern. Une bonne idée, on "l'adapte", grâce à l'expérience. Cela fait une grande nuance.

ET ALORS ?

Bonne question... Je n'ai pas de solution à vous offrir, et ce n'était pas mon propos. Je voulais seulement rappeler que **MVVM n'est que le début du chemin qui mènera à un design pattern, mais qu'en l'état ce n'en est pas un.** Avec toutes les conséquences que cela entraine.

- La fin des discours parfois extrémistes sur tel ou tel "viol" du pattern.
- La fin du rêve qu'un '"absolu" serait atteint en appliquant le "pattern".
- La fin de l'illusion qu'appliquer MVVM répond à tous les problèmes qu'elle soulève.
- Le début d'une nouvelle histoire qui reste à écrire.
- Le début d'une nouvelle impulsion, pour se mettre au travail et trouver comment faire de cette bonne idée un vrai pattern...

CONCLUSION

Dans notre métier secoué régulièrement par les modes et les sigles étranges qui font passer pour des idiots ceux qui n'en connaissent pas la signification, MVVM a su créer son buzz. On pourrait dire qu'il n'y a pas de fumée sans feu. C'est souvent vrai (pas toujours), et si MVVM connait un certain succès c'est bien qu'il véhicule de bonnes idées, séduisantes. Je n'écris plus aucun logiciel sans mettre en œuvre MVVM depuis des années maintenant…

Mais MVVM ne serait rien sans les toolkits ou frameworks qui en offrent une vision personnelle. **C'est grâce à ces outils que MVVM gagne en expérience** ce qui lui permettra, un jour peut-être, d'atteindre le statut de "design pattern".

Il est ainsi hors de question de jeter la pierre à tel ou tel autre framework. Il est tout autant hors de propos de les renier et d'oublier les voies que tracent MVVM.

Il faut continuer à mettre œuvre MVVM, **il faut que chacun exploite plusieurs toolkits et frameworks pour en comprendre les forces et les faiblesses**.

Il n'y a que comme cela, **par l'accumulation de l'expérience**, qu'un jour, l'un d'entre nous sera capable de formaliser réellement MVVM, d'en faire un vrai design pattern aux conséquences maitrisées et sur laquelle pourront être bâtis des frameworks solides et non pas parcellaires.
Rendons une fois encore hommage à ceux qui produisent des frameworks ou des toolkits MVVM, ils font avancer les choses. Rendons un hommage tout aussi appuyé à ceux qui ont le courage de s'en servir pour créer de vraies applications qui dépassent le stade de la démonstration. Leur prise de risque est énorme et leur expérience accumulée est notre espoir de fonder un vrai pattern sur MVVM. Rendons peut-être aussi, et très humblement, un hommage à ceux qui réfléchissent à MVVM et qui prennent parfois beaucoup de temps à l'expliquer à la communauté pour faire avancer les choses...

Simple MVVM

MVVM, Model-View-ViewModel, est un pattern de développement que je vous ai présenté plusieurs fois (billet et un article récent de 70 pages) en raison de son importance dans le développement d'applications Silverlight principalement.

A ces occasions j'ai présenté plusieurs frameworks permettant de faciliter le développement en suivant le pattern MVVM. Je viens d'en découvrir un autre, encore plus simple que MVVM Light, c'est "Simple MVVM" un projet CodePlex.

La gestion des commandes n'est pas prise en compte mais comme je le montre dans le long article évoqué en introduction en utilisant Silverlight 4 (en bêta pour l'instant mais bientôt releasé) on peut facilement gérer l'interface *ICommand* dans une vaste majorité de cas sans utiliser de librairie annexe.

Simple MVVM est vraiment simple. C'est un peu "MVVM pour les nuls". Mais justement, c'est en partant d'exemples simples, de librairies hyper light qu'on peut mieux cerner une technologie et choisir ensuite des frameworks plus lourds et plus complets. Je vous conseille donc d'y jeter un œil.

Dans tous les cas je suis pour les librairies les plus light possible. Les gros “zinzins”, même très bien faits, pose *toujours* un problème de maintenabilité (si vous, personnellement, vous avez investi du temps pour apprendre telle ou telle grosse librairie, c’est bien, mais que ce passera-t-il s’il faut qu’un autre informaticien maintienne votre code au pied levé ? Combien coûtera sa propre formation sur la dite librairie ? Alors que souvent ces dernières ont pour objectif de simplifier le travail et donc de couter moins cher “à la longue”. C’est faux en réalité, et c’est donc un contre-sens que de les utiliser, aussi puissantes ou savantes soient-elles, sauf cas exceptionnels…).

Ce qui se conçoit bien se programme clairement – Et le code pour le faire vient aisément… (Paraphrase libre du célèbre proverbe de Nicolas Boileau).

Silverlight MVVM : les commandes

Silverlight 4 ajoute la gestion des commandes à certains objets comme les boutons ce qui simplifie la mise en œuvre du pattern MVVM. Le lien entre Interface et ViewModel s’en trouve amélioré même si cela semble toujours un peu nébuleux pour le débutant. Il est vrai que programmer de la sorte impose de raisonner autrement. Nous allons le voir au travers d’un exemple.

LE BUT

Je ne vais pas me lancer dans un cours sur MVVM d’autant que j’ai déjà écrit un très long article sur la question il y a peu de temps sans compter les billets où j’y fais référence directement ou non. Le but de ce billet est plutôt de partir d’un cas particulier pour voir la mise en œuvre d’un besoin courant sous MVVM, **les commandes**. Après les visions globales, notamment celle de l’article évoqué, pénétrer par une petite porte et découvrir le labyrinthe est une approche complémentaire, parallèle, pouvant certainement éclairer d’une autre façon le sujet.

STRUCTURE SIMPLIFIEE D’UNE APPLICATION MVVM

Plutôt que des redites je renvoie ainsi le lecteur vers les billets et articles suivants pour qu’il puisse faire le point sur MVVM :

- Billet: Article: M-V-VM avec Silverlight
- Article à télécharger (70 pages) : M-V-VM avec Silverlight, de la théorie à la pratique.
- Billet : MVVM, Unity, Prism, Inversion of Control…
- Billet : Simple MVVM

Un résumé

Pour résumer ici, une application suivant la pattern Model-View-ViewModel est une application qui sépare très fortement l’interface et son design du code applicatif. Cette séparation s’effectue en écrivant d’une part une interface (Xaml), la Vue, et d’autre par un

ViewModel (Modèle de Vue) qui est connecté à l'interface via le DataContext de l'élément le plus haut(le contrôle de base) de cette dernière.

Toutes les données nécessaires au fonctionnement de l'application sont placées dans un ou plusieurs "modèles". Seuls les Modèles de Vue (ViewModel) ont accès aux modèles. Les Vues n'effectuent aucun traitement, elles ne font que consommer des données retournées par le Modèle de Vue qui leur est associé.

Le pattern MVVM accepte qu'une Vue dialogue directement avec un Modèle (les données) s'il n'y a aucun traitement (par exemple une simple fiche listant des données). C'est un cas extrême car même dans une telle situation il a beaucoup de chances qu'il soit nécessaire d'avoir au moins un bouton de sélection, le déclenchement de l'affichage d'un détail, ou autre action qui nécessitera en réalité la présente d'un Modèle de Vue.

De même, en MVVM il n'est pas "interdit" de placer du code dans la Vue. Bien au contraire. Mais ce code ne doit être que du code de présentation (gestion des objets visuels, animations, etc).

Donc, la structure d'une application MVVM ce sont des Vues (des interfaces Xaml) reliées par leur propriété DataContext à des ViewModels (Modèles de Vue) se chargeant de tout le travail. Ces derniers puisent leurs données depuis Internet le plus souvent (web services, Ria services…) mais il peut aussi s'agir de données locales (images, sons, flux webcam…). Lorsque les données sont suffisamment complexes, les ViewModels ne les gèrent pas directement, ils les consomment depuis des Modèles qui implémentent les accès aux sources d'information.

INFORMATIONS MAIS AUSSI COMMANDES

A la limite, si MVVM ne concernait que les informations à afficher, cela serait certainement plus simple à comprendre. Faire un data binding n'est pas une chose nouvelle, cela existe depuis les Windows Forms dans les premières versions de .NET. Relier la valeur d'une propriété à une source de données est trivial. Cette source de données étant ce qu'on veut (une liste, voire une simple instance d'un objet).

Le "volet information" de MVVM reprend ainsi des recettes éprouvées. Seul le mécanisme de séparation entre la vue et son Modèle de Vue peut troubler un peu au départ.

Là où les choses deviennent un peu plus difficile à comprendre, c'est lorsqu'on prend conscience qu'une interface n'est pas qu'un tableau noir utilisant la craie provenant d'un ViewModel pour créer son affichage. Un tableau noir est passif, il reçoit l'information et l'affiche aux yeux de tous, mais sans aucune interactivité.

Plutôt qu'un tableau noir d'école, la Vue est un Tablet PC, un IPad : elle affiche des données mais l'utilisateur peut interagir avec ces dernières. L'interface contient ainsi des éléments permettant **d'envoyer des commandes à l'application**.

Une commande est, *in fine*, une action réalisée par du code. Comme le code applicatif ne peut être placé que dans le ViewModel, c'est donc lui qui contiendra les méthodes réalisant ces actions.

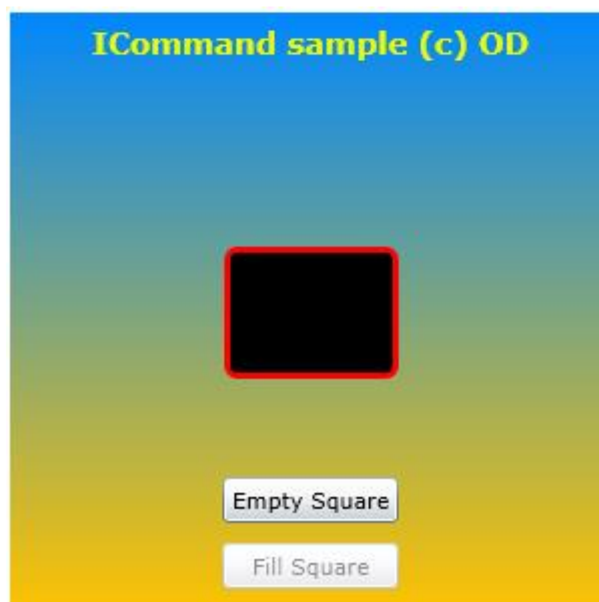
Si certaines actions peuvent être "muettes", la grande majorité implique un changement d'état de *l'automate application*. Tout changement d'état doit être visualisé pour que l'utilisateur sache "où il en est". Cela peut se matérialiser par un simple changement de couleur d'une Led passant du vert au rouge pour indiquer une erreur comme par l'affichage de nouvelles données, la recréation d'un graphique, voire un changement de page ou de site Web.
Les commandes et leurs traitements impliquent donc qu'il existe un moyen :

- de déclencher le code utile se trouvant dans le ViewModel depuis l'interface;
- de récupérer les changements d'état du ViewModel pour mettre l'interface en accord visuellement avec ces derniers.

UN EXEMPLE LIVE

Le principe est simple mais la chaîne d'action pour y arriver en respectant MVVM n'est pas forcément évidente : Un rectangle ayant un bord rouge et, au départ, aucun remplissage.

Deux boutons, l'un permettant de remplir le rectangle (en noir) et l'autre permettant de le vider (intérieur transparent). Selon l'état, le bouton qui n'a pas de sens fonctionnel est disabled.



Testez par vous-mêmes (ça devrait être rapide) on se retrouve au paragraphe suivant !

L'INTERFACE ICOMMAND

La solution de la gestion des commandes passe par la définition d'une interface, **ICommand**, qui représente une méthode pouvant être exécutée. Comme ICommand est un type comme un autre, il est possible de définir des variables de ce type. Je peux écrire “ICommand maCommande;” définissant ainsi une variable “maCommande” de type ICommand.

Si c'est une variable, elle peut être exposée comme n'importe quel autre champ via une propriété du ViewModel. Finalement il n'y a aucune différence avec un string ou un entier.

Et puisque le ViewModel peut exposer des commandes sous la forme de propriétés de type ICommand, la Vue qui est data bound au ViewModel peut consommer ces variables comme n'importe quelle autre.

Cela signifie que les objets d'interface pouvant déclencher une commande, comme le Bouton par exemple, exposent une propriété “Commande” de type ICommand.

C'est justement ce que Silverlight fait depuis sa version 4.

ICommand

Pour être complet disons que l'interface ICommand définit peu de choses :

```
1: public interface ICommand
2: {
3:     // Events
4:     event EventHandler CanExecuteChanged;
5:
6:     // Methods
7:     bool CanExecute(object parameter);
8:     void Execute(object parameter);
9: }
10:
11:
```

- **CanExecute**, qui permet de savoir si la commande peut être exécutée (selon l'état de l'application)
- **CanExecuteChanged**, un événement qui permet de savoir quand l'état de la commande change (ce qui permet par exemple de mettre un bouton Enabled/Disabled automatiquement)
- **Execute**, qui exécute l'action représentée par la commande.

Rien de bien complexe dans tout ça donc. Reste, pour chaque commande, à coder le nécessaire. On notera que sous WPF l'ensemble est un peu plus sophistiqué avec un CommandManager qui se charge entre autre de vérifier le CanExecute pour mettre l'interface sans cesse en accord avec l'état de l'application. Cela n'existe pas sous Silverlight et nous verrons comment, malgré tout, arriver à un résultat identique (dans l'exemple cela concerne l'état disabled des boutons).

LE CIRCUIT D'UNE COMMANDE

La commande est ainsi définie dans le ViewModel comme une propriété de type ICommand. Dans le constructeur du ViewModel (généralement) le développeur initialise la valeur de ces propriétés un peu particulières. S'il s'agissait d'un entier on écrirait "maCommande = 12;" par exemple pour l'initialiser à "12".

Mais comme ICommand est un type décrivant un code à exécuter, un delegate (ou plus exactement un System.Action<> ou un Predicate pour le CanExecute), ce n'est pas une valeur comme "12" que le programme fixera pour la variable maCommande mais tout simplement une Action, donc un bout de code. Ce bout de code peut être une méthode vers laquelle la commande pointera, un delegate, ou bien une expression lambda, ce qui est très souvent utilisé.

Nous disposons maintenant d'une propriété "maCommande" de type ICommand qui "pointe" vers une Action (un code), l'initialisation de la variable se faisant dans le constructeur du ViewModel. La propriété est bien entendu en mode read only c'est à dire qu'elle dispose d'un getter public mais pas d'un setter (ou bien ce dernier est private, ou la propriété utilise un backing field retourné par le getter et éventuellement modifié en interne).

Comme "maCommande" est publique, elle s'appellera plutôt "MaCommande". Et comme il est préférable de différencier les commandes des autres propriétés (c'est plus lisible) on ajoute le suffixe '"Command" en général. "MaCommande" cela ne veut rien dire, alors devenons plus concret. Reprenons notre exemple live avec un Rectangle qui peut être soit plein soit vide. On exposera deux commandes "FillRectangleCommand" et "EmptyRectangleCommand".

Une commande étant souvent visualisée par un bouton ou autre élément de ce type, on a pour habitude aussi de déclarer pour chacune une propriété string qui retourne le texte à afficher (texte qui pourra de plus être facilement localisé dans le ViewModel).

Ici nous aurons ainsi 4 propriétés, 2 commandes et 2 textes de commande :

- FillRectangleCommand, EmptyRectangleCommand

- FillRectangleCommandText, EmptyRectangleCommandText

Le suffixe “Text” se rajoute à “Command”. Il est ainsi très facile de voir qui est la commande et qui est le texte du nom de la commande. Cela sera bien pratique lors de l’établissement du data binding.

```
1: public RelayCommand EmptySquareCommand { get; private set; }
2: public string EmptySquareCommandText { get { return "Empty Square"; } }
3: public RelayCommand FillSquareCommand { get; private set; }
4: public string FillSquareCommandText { get { return "Fill Square"; } }
```

Vous noterez que les commandes sont déclarées de type RelayCommand et non ICommand. Cela est lié au fait que l’exemple utilise la librairie MVVM Light. RelayCommand supporte ICommand bien entendu et pour ce qui nous intéresse ici on pourra considérer que cela ne fait aucune différence.

Bref. Nous disposons d’un ViewModel exposant des commandes et des textes. Ce ViewModel est lié à la Vue par le DataContext de cette dernière.

On notera au passage que le lien Vue –> Modèle de Vue ne s’effectue pas directement sous MVVM Light, ce qui est une bonne chose et créé un nouveau niveau de découplage. Il existe un ServiceLocator ayant pour propriétés l’ensemble des ViewModels existants. La Vue se lie ainsi à son ViewModel en passant par la propriété ad hoc du Service Locator. L’implémentation est triviale (vous le verrez dans le code source du projet). De fait, le code Xaml de l’application est celui-ci :

```
1: <UserControl x:Class="SLmvvm01.MainPage"
8:          . . .
9:          DataContext="{Binding Main, Source={StaticResource Locator}}">
```

Le Service Locator s’appelle Locator, la fiche de l’exemple fait pointer son DataContext sur la propriété Main de ce Locator, propriété qui retourne l’instance du ViewModel associé à cette page.

Il est donc maintenant possible d’établir des data binding entre les éléments de la Vue, des boutons par exemple, et les propriétés du ViewModel.

Le titre de l’application est ainsi fourni par le ViewModel et n’est pas une chaîne fixée dans l’interface. Son code Xaml est le suivant (en supprimant le code de présentation):

```
1: <TextBlock Text="{Binding Title}" ... />
```

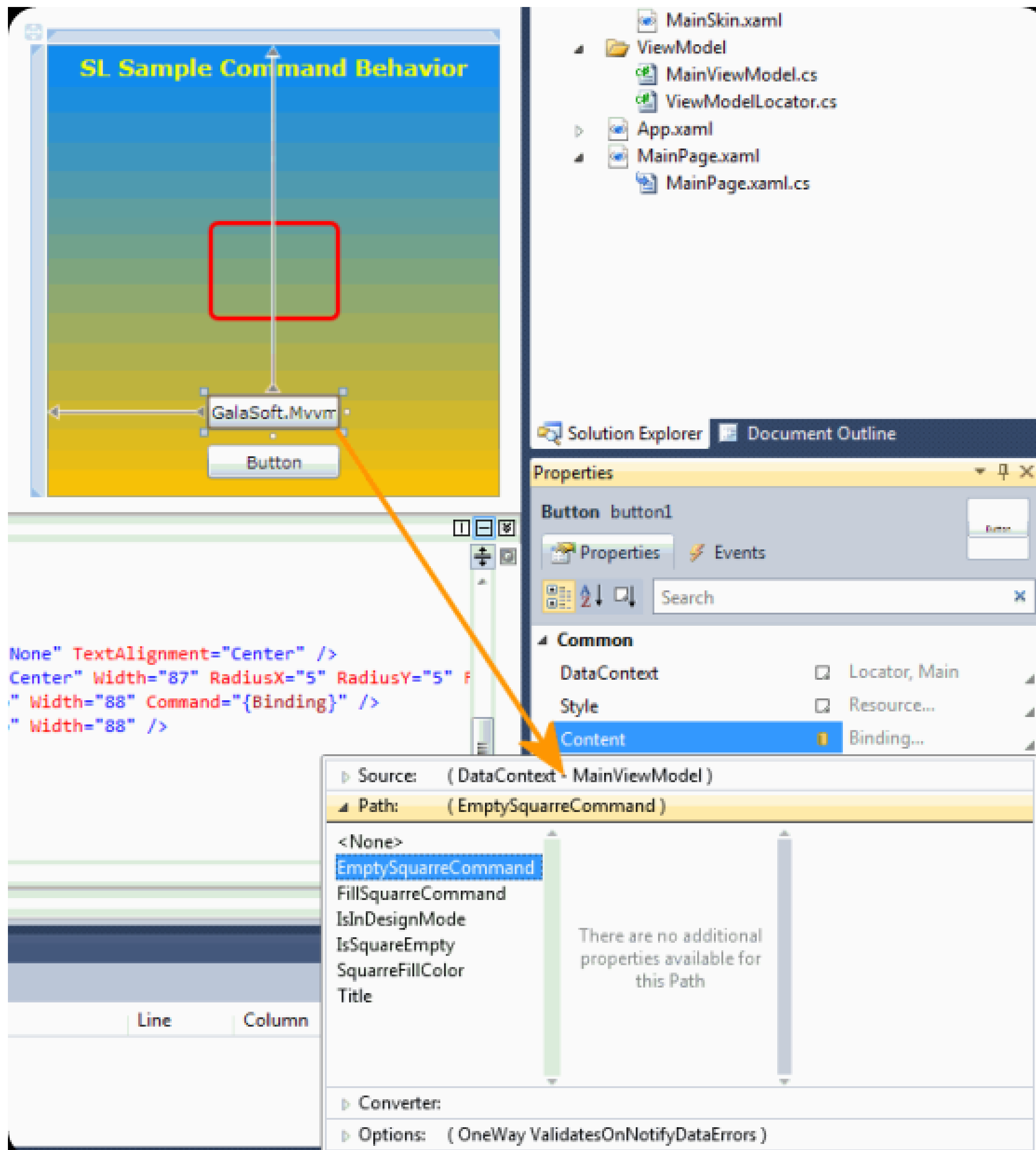
“Title” est une propriété du ViewModel associé à la Vue.

Pour les boutons il y a deux bindings, un pour le texte, l’autre pour l’action :

```
1: <Button Content="{Binding Path=EmptySquareCommandText}"
2:        Command="{Binding Path=EmptySquareCommand}" />
3: <Button Content="{Binding Path=FillSquareCommandText}"
4:        Command="{Binding Path=FillSquareCommand}" />
```

Le circuit commence à se former : la Vue expose un bouton, celui-ci voit sa propriété Command reliée par data binding à la commande xxx exposée par le ViewModel. Donc un clic sur le bouton exécutera le code se trouvant dans le ViewModel. Parfait !

Ci-dessous on voit comment, sous Visual Studio, le binding entre l’un des boutons et sa commande dans le ViewModel s’effectue facilement (notammant grâce au Service Locator qui instancie le ViewModel) :



Oui mais il manque le circuit retour, *le nécessaire feedback* : l'action change l'état de l'application et **l'interface doit refléter ce changement** !
Comment marche ce circuit de feedback ?

FEEDBACK DES COMMANDES

Il y a plusieurs options, mais les deux plus utilisées sont les suivantes :

- le data binding lui-même
- un système de messages

Le data binding peut largement suffire à assurer ce feedback entre le ViewModel et sa Vue. Poursuivons l'exemple du rectangle qui peut être vide ou plein. Il suffit que sa propriété "Fill" soit data bound à une propriété "SquareFillColor" de type SolidColorBrush du

ViewModel et l’affaire est jouée ! Enfin presque… Il ne faut pas oublier dans le code des commandes de lever un PropertyChanged de SquareFillColor.

Dès lors, quand on clic sur le bouton “remplir rectangle”, cela invoque la commande correspondante se trouvant dans le ViewModel. Cette commande modifie la propriété SquareFillColor et, d’une façon ou d’une autre (directement ou indirectement) déclenche l’événement PropertyChanged de cette dernière. Comme l’objet Rectangle de la Vue a sa propriété Fill reliée par data binding à SquareFillColor du ViewModel, tout changement de cette dernière (repéré par le ProperyChanged) mettra automatiquement à jour l’affichage… Magique!

```
1: <Rectangle Height="66" Stroke="Red" StrokeThickness="3" Width="87"
2:          RadiusX="5" RadiusY="5"
3:          Fill="{Binding Path=SquareFillColor}" />
```

On voit ci-dessus comment la propriété Fill du Rectangle est liée à la propriété SquareFillColor du DataContext du UserControl (pointant, via la propriété Main du Locator, le ViewModel associé). Tout déclenchement du PropertyChanged de SquareFillColor entraînera la mise à jour de la propriété Fill du rectangle. Ce n’est pas du MVVM ici, juste du Binding…

Dans certains cas le changement d’état qu’entraîne l’action nécessite un traitement différent. Un simple data binding entre le Fill d’un rectangle et une propriété Brush n’est pas suffisant. On met alors en place un procédé de messagerie : au lieu d’un PropertyChanged, c’est un message particulier (par exemple “L’état de l’application vient de passer à xxx”) qui va être diffusé. Toutes les instances de toutes les classes participant au fonctionnement de l’application pourront s’abonner à cette messagerie, voire à ce message particulier et réagir en conséquence (mise à jour d’affichage si c’est une Vue, mise à jour de l’état d’un autre objet, etc) ce qui peut bien entendu déclencher d’autres PropertyChanged ou l’émission de d’autres messages… Les messages peuvent même transiter d’un ViewModel à un autre, être diffusés ou traités par les Modèles, etc. La gestion d’une messagerie apporte une grande souplesse et permet de régler de nombreux problèmes que nous ne voyons pas l’exemple du jour (par exemple l’affichage d’un message : déclenché par une action utilisateur, une commande, le ViewModel peut avoir à afficher un dialogue, ce qu’il ne peut pas – ne doit pas – faire, il faut ainsi qu’il demande cet affichage à un objet d’interface mais aussi qu’il puisse receueillir le résultat… un cas que je traiterais dans un prochain article sur MVVM Light).

Un tel système de message n’est pas implémenté directement sous Silverlight. Mais il existe des solutions (voir les billets cités en introduction) sous la forme de librairie plus ou moins grosses.

Personnellement, et après les avoir toutes essayées (les principales en tout cas) j'en suis arrivé à la conclusion que la seule qui soit vraiment abordable pour une majorité de développeur est MVVM Light de Laurent Bugnion. Quand je dis "abordable" cela ne veut pas dire que je considère les développeurs comme des idiots. Non, mais je sais à quel point la charge de travail qui pèse sur eux limite de façon drastique le temps qu'ils peuvent investir dans la formation à une librairie. Prism est géniale, très riche et répond à d'autres problématiques, mais Prism réclame beaucoup d'investissement. Et quand on maîtrise peu ou mal un framework quel qu'il soit, on fait fatalement des bêtises et on perd du temps.

Je suis donc partisan des petits frameworks qui font peu de choses et qui s'apprennent vite. On les maîtrise de bout en bout et on s'en sert bien. C'est une vision réaliste du monde donc, et non un manque de confiance en vos compétences !

MVVM LIGHT

MVVM Light est une librairie d'un seul homme. Elle est donc simple et se concentre sur une chose : MVVM. Laurent Bugnion, un MVP aussi, est un développeur compétent plein de bonnes idées et surtout adepte, comme je le suis, des choses simples qui s'apprennent vite. Son framework est donc totalement dédié et uniquement dédié aux mécanismes permettant d'écrire facilement des applications MVVM sous WPF, Silverlight et Phone 7/8. Des templates sont fournis ainsi que des snippets pour aller encore plus vite.

Bien entendu, Laurent est un peu charrette comme nous tous… Il fait beaucoup de choses et si la librairie est bien écrite et évolue très régulièrement, c'est comme toujours la doc qui trinque un peu… Mais le code source est simple, lisible, et c'est comme cela qu'on apprend le mieux à quoi il sert : en l'étudiant ! Il existe aussi des billets ou articles épars mais il est vrai que pour un grand débutant l'approche sera un peu rude (surtout que tout est en anglais).

Je prépare d'ailleurs un article sur MVVM Light pour rendre tout cela plus abordable.

Pour l'instant vous pouvez bien entendu charger et installer MVVM Light (ce qui sera nécessaire pour faire tourner l'exemple) : http://www.galasoft.ch/mvvm/getstarted

L'INITIALISATION DES COMMANDES

On en a parlé, mais on ne l'a pas vu ! Les commandes sont des propriétés de type RelayCommand (ICommand) qu'on initialise généralement dans le constructeur du ViewModel. Cela peut se faire de plusieurs façons mais les expressions Lambda simplifient et allègent le code. Voici l'initialisation des deux commandes de l'exemple :

```
1: EmptySquareCommand = new RelayCommand((Action)(() => IsSquareEmpty = true),
2:             () => !isSquareEmpty);
```

```
3: FillSquareCommand = new RelayCommand((Action)(() => IsSquareEmpty = false),
4:            () => isSquareEmpty);
```

Les commandes se bornent à basculer la propriété IsSquareEmpty, un booléen qui est défini comme suit :

```
1: private bool isSquareEmpty = true;
2:  public bool IsSquareEmpty
3:  {
4:      get { return isSquareEmpty; }
5:      set
6:      {
7:          isSquareEmpty = value;
8:          RaisePropertyChanged("IsSquareEmpty");
9:      }
10:  }
```

On notera bien entendu l'appel à RaisePropertyChanged, une méthode de MVVM Light qui propage le changement d'une propriété comme PropertyChanged.

Nous disposons aussi d'une autre propriété, celle qui est liée au Fill du Rectangle :

```
1: public SolidColorBrush SquareFillColor
2: {
3:     get { return isSquareEmpty ? null : new SolidColorBrush(Colors.Black); }
4: }
```

Normalement, IsSquareEmpty, dans son setter, devrait appeler le RaiseNotifyChanged de SquareFillColor. En effet, c'est cette dernière propriété qui est liée au Fill du Rectangle, pas le booléen que nous utilisons en interne dans le code… Même si ce booléen comporte bien un RaisePropertyChanged cela ne peut avertir l'interface que SquareFillColor vient de changer.

Dans un tel cas les PropertyChanged doivent se suivre, mais une petite astuce de MVVM Light nous en empêche. L'enfer étant pavé de bonnes intentions, MVVM Light, en mode Debug, vérifie que la chaîne de caractères passée dans le RaisePropertyChanged correspond bien au nom de la propriété en cours. C'est une sécurité vraiment géniale qui efface un peu l'aberration de la présence d'une chaîne de caractères non controlée dans un si beau langage fortement typé qu'est C#… Hélas cette astuce fait que si nous plaçons le second

RaisePropertyChanged sous celui du booléen, MVVM Light lèvera une exception car le nom (le second) ne correspond pas à celui de la propriété en cours. Fâcheux dans ce cas précis.

Pour s'en sortir il faut définir dans le ViewModel un gestionnaire de son propre PropertyChanged, et dans le code de ce gestionnaire il faut vérifier si c'est la propriété booléenne qui vient de changer. Dans l'affirmative on effectue le RaisePropertyChanged manquant (sans erreur car nous ne somme pas dans le code de définition d'une property).

LE COMMANDMANAGER MANQUANT

Comme je l'i dit plus haut, WPF offre un CommandManager qui s'occupe d'exécuter les CanExecute de toutes les commandes pour s'assurer que les éléments d'interface connectés aux commandes passent bien en disabled / enabled selon le cas.

Or Silverlight n'offre pas cette classe. Il en existe des substituts sur CodePlex, mais ici nous ferons sans autre code tiers.

Il convient donc, quand nous savons qu'une action modifie la possibilité d'exécuter ou non une commande d'appeler pour la ou les commandes concernées leur méthode RaiseCanExecuteChanged, une méthode fournie justement par RelayCommand et qui n'existe pas dans l'interface ICommand.

Ainsi, lorsque IsSquareEmpty est modifiée nous prenons en charge dans un gestionnaire d'événement PropertyChanged du ViewModel l'ensemble des cas de figure :

```
1: void MainViewModel_PropertyChanged(object sender,
2:      System.ComponentModel.PropertyChangedEventArgs e)
3: {
4:    if (e.PropertyName == "IsSquareEmpty")
5:    {
6:       RaisePropertyChanged("SquareFillColor");
7:       EmptySquareCommand.RaiseCanExecuteChanged();
8:       FillSquareCommand.RaiseCanExecuteChanged();
9:    }
10: }
```

Et voilà ! Lorsque la propriété IsSquareEmpty change, notre gestionnaire effectue les tâches suivantes :

- RaisePropertyChanged de la propriété SquareFillColor, ce qui entraîne la mise à jour de l'affichage dans la Vue (grâce au binding entre le Fill du rectangle et cette propriété)
- RaiseCanExecuteChanged est invoqué pour les deux commandes qui dépendent de l'état du booléen. Le code exécuté est celui défini par les expressions Lambda

initialisée dans le constructeur du ViewModel. L'interface réagit aux événements CanExecuteChanged de chaque commande et les boutons se placent en mode enabled / disabled tous seuls.

- Voir MVVM non plus de façon globale mais au travers d'un exemple simple et d'une problématique courante, la gestion des commandes, m'a semblé être une approche complémentaire aux articles et billets que j'ai déjà écrits sur la question.

Article: M-V-VM avec Silverlight

Model-View-ViewModel, je vous en parlais il y a très peu de temps (MVVM, Unity, Prism, Inversion of Control…) et je vous avais promis un exemple pour rendre tout cela plus concret. C'est fait ! Et même mieux, un article de **70 pages** l'accompagne !

Vous saurez tout (ou presque) sur cette design pattern absolument incontournable pour développer sérieusement sous Silverlight.

Après des explications sur le pattern elle-même l'article vous présente une application exemple entièrement réalisée avec ce qu'il y a "out of the box". J'ai fait le choix de n'utiliser aucun Framework existant (Prism, Cinch, Silverlight.FX, MVVM Light…) pour vous montrer que M-V-VM peut entièrement être mis en œuvre "à la main" sans aide extérieure.

Cela ne veut pas dire que tous ces Frameworks (dont l'article parle aussi) ne sont pas intéressants, au contraire ! Mais comment choisir une librairie facilitant M-V-VM si vous ne savez pas comment mettre en œuvre ce pattern et si vous ne connaissez pas les difficultés qu'elle soulève autant que ses avantages ?

Cet article vous permettra de faire le point sur M-V-VM et de pouvoir ensuite choisir le Framework qu'il vous plaira en toute connaissance de cause ou bien cela vous aidera à développer votre propre solution. Après tout, l'application exemple fonctionne parfaitement sans aucun de ces Frameworks….

Le code source du projet est fourni. En raison de l'énorme avantage de la gestion des commandes introduites dans Silverlight 4 l'article utilise cette version qui sera bientôt disponible. Tout est expliqué pour savoir comment faire fonctionner le code exemple à l'aide de VS ou Blend.

L'article peut être lu sans faire tourner le code si vous ne souhaitez pas installer la beta de SL4, et la première partie théorique s'applique aussi bien à M-V-VM sous SL3.

Bonne lecture !

(PS: n'oubliez pas que depuiis août 2012 les articles sont regroupés sur la page publications).

Téléchargement ici : M-V-VM avec Silverlight, de la théorie à la pratique.

M-V-VM et Silverlight – De la théorie à la pratique [avec Silverlight 4, VS 2010 et Blend 4]

Version 1.0 Janvier 2010

REFERENCES

J'ai pioché dans les sites suivants pour y récupérer certains schémas. Les articles sont tous intéressants et vous pouvez bien entendu les lire pour compléter le présent document.

Model-View-Controller http://fr.wikipedia.org/wiki/Mod%C3%A8le-Vue-Contr%C3%B4leur
Model-View-Presenter http://blog.fossmo.net/post/Model-View-Presenter-explained.aspx
Supervising Controller http://msdn.microsoft.com/en-us/library/cc707873.aspx
Passive View http://msdn.microsoft.com/en-us/library/cc304760.aspx
View-Presenter http://jab.developpez.com/tutoriels/dotnet/mvppattern/
Model-View-ViewModel http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx
TechEd 2007 http://www.e-naxos.com/Blog/?tag=/teched

(Beaucoup d'autres références se trouvent réparties dans l'article)

CODE SOURCE

Cet article est accompagné du code source complet de l'application exemple.

Décompressez le fichier Zip dans un répertoire de votre choix. Attention le projet nécessite Silverlight 4, Visual Studio 2010 et Blend 4. Le code source peut néanmoins être ouvert et lu avec tout éditeur de code.

PREAMBULE

M-V-VM est un design pattern née du monde WPF / Silverlight pour WPF et Silverlight. Bien qu'il puise certaines de ses racines dans des patterns plus anciens il ajoute des éléments essentiels. Son but est de proposer un modèle de programmation pour WPF et Silverlight facilitant les tests, introduisant un découplage fort entre les tiers, et insistant sur la nécessité de séparer totalement l'implémentation de l'IHM du reste du code. En outre, M-V-VM permet de résoudre enfin le problème de la séparation du travail entre graphistes et informaticiens ce qui, faute d'une stratégie claire, n'apparaît pas si évident que cela.

> *La chaîne de production d'une application WPF/Silverlight/WP/WinRT, notamment au travers de Expression Blend, fait apparaître un rôle décrit comme celui « d'intégrateur », entre le designer et les développeur. L'industrie a horreur de ce qui est rare, car cher. Ce poste mi-infographiste mi-informaticien n'a que peu de chance d'exister un jour car le profil qu'il impose est rare par essence. M-V-VM permet de rendre Xaml utilisables par des graphistes et des informaticiens sans passer par ce rôle improbable d'intégrateur. M-V-VM prend donc ici une importance cruciale : elle rend la chaine de production de logiciels WPF / Silverlight « industrialisable ». De ce point de vue M-V-VM a une portée qui dépasse de loin son seul intérêt technique et méthodologique.*

Après une présentation de M-V-VM nous verrons comment la mettre en œuvre avec la réalisation d'une application exemple sous Silverlight 4+.

Il est possible d'utiliser Silverlight 2 ou 3 pour construire des applications M-V-VM mais il faut utiliser des librairies externes. J'ai fait le choix ici de ne pas suivre ces dernières pour vous montrer que le principe est applicable « out of the box ». Toutefois cela nécessite quelques améliorations introduites depuis Silverlight 4.

INTRODUCTION A M-V-VM

De quoi s'agit-il ?

M-V-VM est un design pattern, c'est-à-dire la formalisation d'un savoir réutilisable à l'infini. Une solution générique à un problème qui l'est tout autant.

M-V-VM signifie « Model – View –ViewModel » (Modèle – Vue – ModèleDeVue[1])

En fait vous connaissez certainement ce qui se cache derrière M-V-VM, au moins de nom, puisque cette pattern est très proche du pattern M-V-C, Model – View – Controller (Modèle – Vue – Contrôleur). Cette stratégie a pour principale finalité une meilleure séparation entre la couche métier, l'accès aux données et l'interface utilisateur.

[1] Traduction d'après Gossman, créateur de la pattern M-V-VM. Voir plus loin dans cet article.

Mais parler d'équivalence entre M-V-VM et M-V-C n'est qu'un raccourci pratique qui n'est valable uniquement qu'en première approximation pour aider à comprendre de quoi il s'agit. Si on veut mieux appréhender ce qu'est M-V-VM il est nécessaire de s'attacher aux détails mais aussi à l'histoire de la branche de patterns de même type dont elle est issue.

Je vous propose ainsi dans un premier temps une visite guidée des diverses patterns de la famille de M-V-VM, ce petit parcours vous aidera à mieux cerner le cheminement de pensée qui se cache derrière.

Rappel du modèle M-V-C

Classiquement, M-V-C se représente par un schéma tel que montré figure 1 illustrant la collaboration qui s'instaure entre les trois parties :

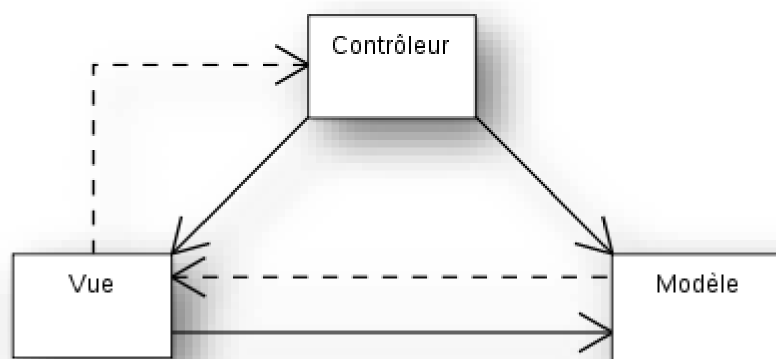


Figure 1 - Le modèle M-V-C

Le **Modèle** prend en charge tous les traitements. Il expose des propriétés et des méthodes permettant d'agir sur les données de l'application mais ne propose aucune représentation visuelle de celles-ci.

La **Vue** est l'interface utilisateur. C'est elle qui expose les données du Modèle. Elle a aussi pour rôle de transmettre les actions de l'utilisateur au Contrôleur (clic sur une entrée de menu, un bouton…). La Vue n'effectue aucun traitement.

Le rôle du **contrôleur** se limite à la gestion des actions, les événements parvenant soit de la vue, soit du Modèle, avec pour mission centrale d'assurer la synchronisation entre ces deux blocs. Par exemple si le clic d'un bouton (événement de l'interface utilisateur donc provenant de la Vue) doit mettre à jour une liste (affichée par la Vue), le cycle sera celui montré par le diagramme de séquence figure 2.

Nota : sur le schéma figure 1 les traits pleins indiquent une utilisation, par exemple la Vue utilise le Modèle (elle le « connaît » et y accède). De même le Contrôleur « connaît » la Vue et le Modèle. Les traits discontinus indiquent le chemin des événements. La Vue transmet les événements utilisateur au Contrôleur, le modèle transmet les événements concernant les données à la Vue pour qu'elle se rafraichisse. Le modèle M-V-C, bien motivé par une séparation de l'IHM et du code, laisse entrevoir de nombreuses interdépendances entre les blocs. Il existe des améliorations plus sophistiquées de ce pattern visant justement à diminuer ces dépendances.

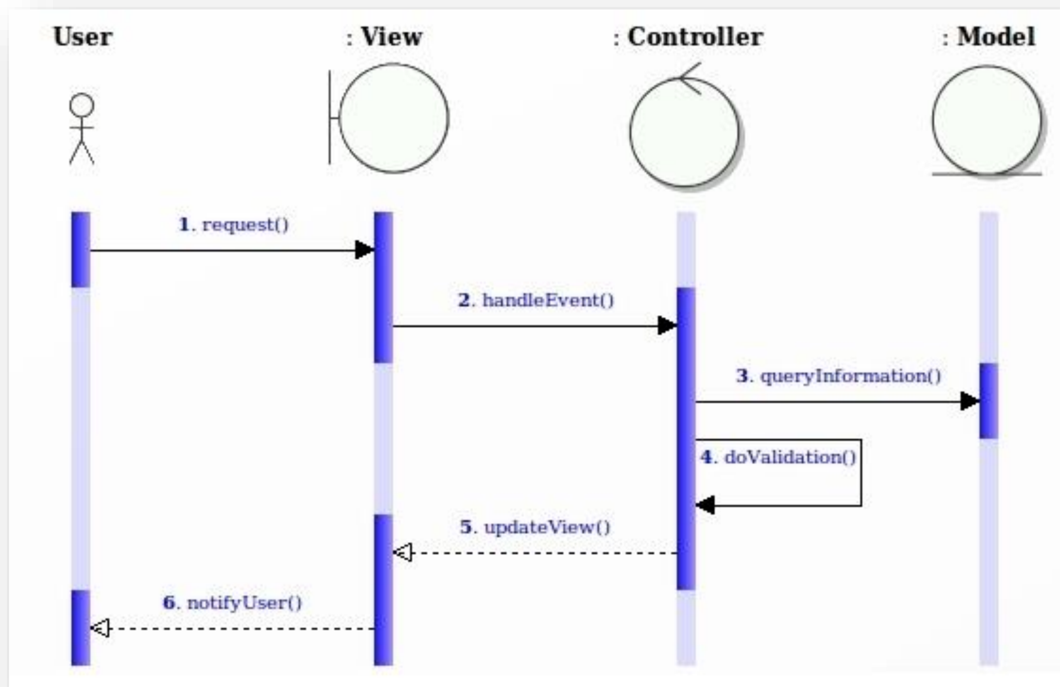


Figure 2 - MVC - Séquence d'une action utilisateur

L'utilisateur émet une requête (ex : clic du bouton) ce qui est d'abord géré par la vue (puisqu'elle possède les objets d'interface). La vue notifie le contrôleur qui va à son tour notifier le modèle (accès aux données). Le contrôleur peut profiter du retour des données pour effectuer des validations puis il avertit la vue qu'elle doit se rafraichir. Elle-même transfère cette notification aux composants liés aux données qui peuvent ainsi refléter les changements. Enfin, l'utilisateur est notifié par la vue de la fin de son action. Cette dernière notification peut s'accompagner d'un message, d'un changement visuel, ou tout simplement exister « de fait » par la mise à jour des conteneurs de données (ce que l'utilisateur détecte et peut donc être considéré comme une notification).

M-V-C connaît des variantes d'interprétation, mais les rôles présentés ici font à peu près consensus.

M-V-VM vs M-V-C

Dans M-V-VM on peut avoir l'impression qu'on a juste remplacé le Contrôleur par le « ViewModel » (Le « Modèle de Vue ») dont nous verrons plus loin la réelle signification. Il s'agirait alors d'une sorte de « relookage » d'un concept ancien (mais toujours d'actualité) pour mieux pouvoir communiquer à nouveau sur celui-ci. Cela arrive souvent dans notre profession…

Il faut chasser de son esprit cette impression. Entre M-V-C et M-V-VM il s'est passé du temps. Les outils ont évolué, les besoins sur le terrain aussi, et les méthodologistes ont continué à faire travailler leurs neurones ! L'évolution entre les deux patterns est une *maturation* ayant entrainé une *modification de point de vue*.

Ainsi dans les détails les différences sont malgré tout plus nombreuses qu'on le croit de prime abord. Certes, en première approximation on peut dire que les deux design patterns sont très proches. Mais uniquement si on en reste aux grandes lignes.

En réalité bien plus que l'échange du Contrôleur pour le Modèle de Vue, Ce sont aussi et surtout les rôles respectifs de la chaîne M-V-C qui ont été redéfinis. D'ailleurs, le pattern M-V-VM hérite finalement plutôt du pattern Model-View-Presenter du point de vue des rôles, sans être totalement équivalente à celle-ci non plus. On voit ici le jeu des influences croisées qui ne simplifient pas la détermination des filiations.

Model-View-Presenter

M-V-P définit le Modèle comme étant une interface définissant les données à afficher, la Vue ne change guère de sens (IHM) et le Presenter est un intermédiaire entre la Vue et le Modèle. Il va chercher les données dans ce dernier, les mets en forme pour la Vue et il persiste les données modifiées par la Vue en utilisant le Modèle.

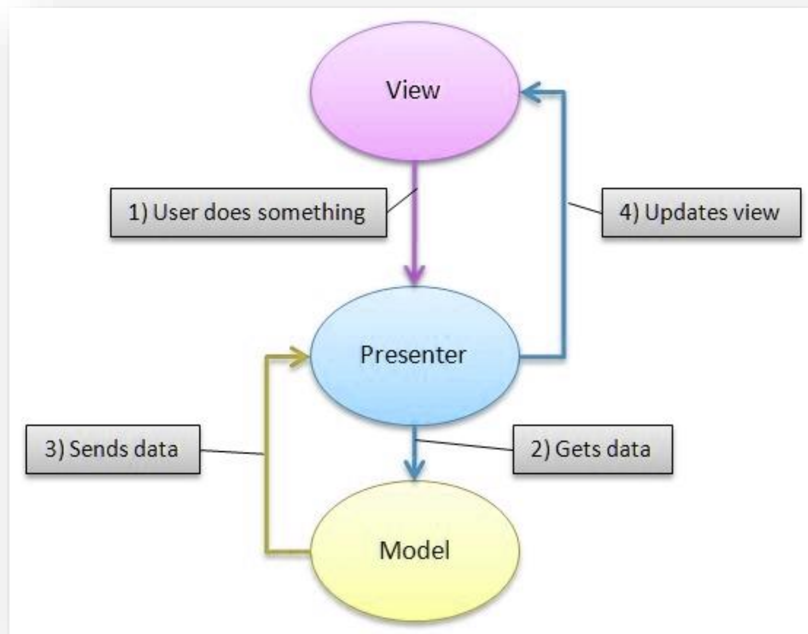


Figure 3 - Schéma de la pattern Model-View-Presenter

> *On notera que le pattern M-V-P née en 1990 a ensuite été découpé en 2006 en deux patterns par Martin Fowler[2] : « Supervising controller » et « Passive view ». Certains disent depuis que M-V-P n'existe donc plus, mais d'autres continuent à l'utiliser…*

Même si du point de historique cela ne semble pas tout à fait exact, du point de vue pratique on peut dire que M-V-P peut être vue comme un hybride entre M-V-C et un autre modèle, V-P, View-Presenter.

View-Presenter

Ce pattern ne définit que deux blocs principaux comme l'illustre la figure 4.

[2] Martin Fowler est un auteur et un consultant renommé. Son site : http://martinfowler.com/

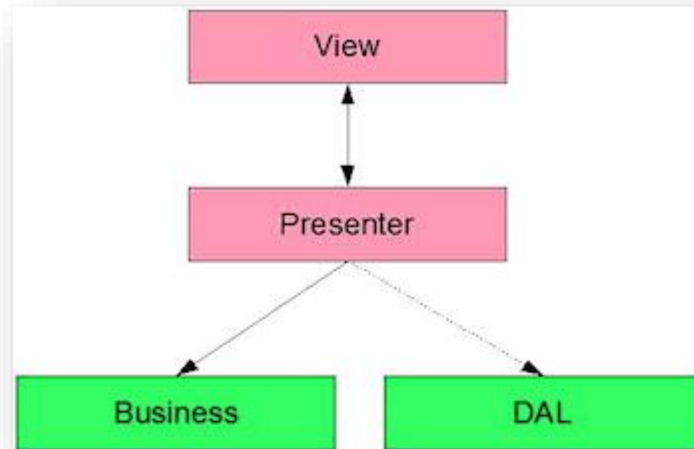


Figure 4 - Le modèle View-Presenter

Sans trop entrer dans les détails on voit que dans ce pattern il n'existe pas de Contrôleur. Le Presenter à la charge de répondre aux sollicitations de la Vue mais aussi de la piloter, ce qui explique le lien bidirectionnel entre ces deux blocs. Comme cela créé un couplage assez fort, le pattern prévoit qu'en réalité la Vue n'est accessible pour le Presenter que depuis une interface IViewer, déclarée dans le Presenter (la Vue implémentant alors cette interface). C'est ce qui est illustré par la figure 5.

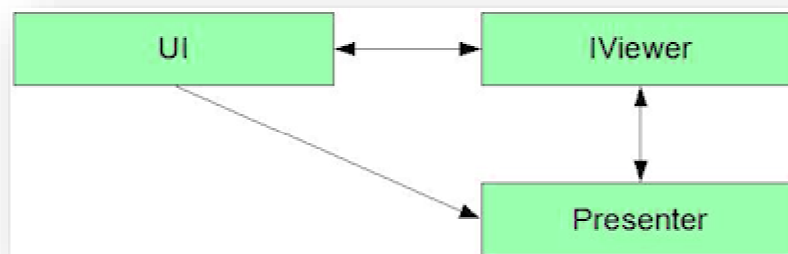


Figure 5 - L'interface IViewer dans View-Presenter

Ce pattern est une autre façon de concevoir la séparation entre l'IHM et le code, elle fait jouer un rôle nouveau au Presenter, assez différent de celui du Contrôleur de M-V-C. La nécessité de « bricoler » ce pattern en ajoutant une interface pour assurer un réel découplage ne me plaît pas beaucoup et explique peut-être aussi pourquoi elle n'a pas l'air d'être très populaire. Mais elle fait avancer les choses conceptuellement. Et M-V-VM lui doit conceptuellement certainement beaucoup.

Et M-V-VM ?

C'est le 8 octobre 2005 que John Gossman[3], l'un des architectes de WPF et Silverlight a parlé de cette pattern dans son blog, le billet s'intitulait « Tales from the Smart Client » (qu'on pourrait traduire par « Contes du Client Riche »).

Gossman lui-même ouvre son billet en présentant M-V-VM comme une variation de M-V-C taillée pour les plateformes de développement d'interfaces utilisateurs modernes où les Vues sont de la responsabilité d'un *designer* (infographiste) plutôt que d'un développeur classique.

Si M-V-C, V-P, M-V-P et consorts parlaient uniquement de données, de vues et de ce que nous appelons dans notre métier « la plomberie », *M-V-VM est la première pattern à aborder l'aspect purement graphique des interfaces en faisant intervenir un tiers jusqu'à lors absent : le designer (ou infographiste).*

Cette variation est très importante. En effet, il était possible d'utiliser M-V-C (et les autres patterns) avec Delphi ou C++ sous Win32, ou même avec ASP.NET plus récemment. Et même s'il était sage de faire appel à un spécialiste du design pour concevoir de belles interfaces à l'époque de ces technologies, ce rôle était souvent considéré comme accessoire, passé sous silence, voire dans 99% totalement zappé !

Avec WPF et Silverlight Microsoft a introduit quelque chose de nouveau dans la chaîne de création d'un logiciel (ou plutôt *quelqu'un* de nouveau), le designer.

> *Je me rappelle de la session d'ouverture des TechEd à Barcelone en novembre 2007[4]. S. Somasegar, Corporate Vice President de Microsoft, avait alors basé toute son introduction sur une anecdote lui permettant d'affirmer que désormais le meilleur ami du développeur était le designer. Les TechEd ne sont pas un petit événement, et la keynote d'ouverture est essentielle, c'est là que Microsoft force le trait sur tout ce qui compte pour l'éditeur. C'est la keynote qui est la plus décryptée, la plus prisée, c'est celle où chacun essaye de lire entre les lignes pour tenter de deviner l'avenir au travers des annonces qui sont faites. Là le message était clair et bien assené ! Ce dont je vous parle aujourd'hui n'est donc pas qu'une approche méthodologique, c'est aussi un axe de développement privilégié par Microsoft depuis la sortie de WPF et pour les années à venir, quelles que soient les patterns utilisées pour atteindre l'objectif final…*

---

[3] Voir les référence en introduction de l'article

[4] Voir dans les références en début d'article ma série de billets sur les TechEd 2007 et notamment mon résumé de la keynote d'ouverture.

Mais revenons à Gossman. Dès l'introduction de son billet il insiste sur une chose : M-V-VM repose sur quelque chose de plus que M-V-C : M-V-VM se fonde sur Avalon (qui deviendra WPF à sa sortie) et sur un mécanisme unique et nouveau de data binding propre à Xaml.

Quel que soit son héritage, M-V-VM ajoute donc une dimension nouvelle : son adaptation à WPF (et à Silverlight) !

On commence à mieux comprendre pourquoi je passe autant de temps à vous en parler !

Selon Gossman, le **Modèle** se définit comme dans M-V-C, ce sont les données et la logique métier. Il est complètement indépendant de l'IHM. Le Modèle est écrit en code ou peut même n'être que de simples données dans une base de données relationnelles voire même un simple fichier XML. Le dénuement est donc poussé à l'extrême laissant à chacun toute latitude de modeler le Modèle à sa convenance !

La **Vue** est un ensemble d'éléments visuels. C'est l'IHM, décrite en Xaml. Dans les premières explications de Gossman elle prend en charge certaines tâches qui sont à la charge du Contrôleur dans M-V-C (en réalité certaines actions traitant purement l'IHM peuvent être codées dans les vues, comme la gestion d'un drag'n drop. L'action finale ne peut en revanche être réalisée que par le ViewModel). La vue est encodée de façon déclaratives avec un outil spécialisé (on entend les mots Xaml et Blend sans qu'ils ne soient encore prononcés…).

Pour Gossman, dans la version la plus simple, la Vue est bindée[5] directement au Modèle. Mais en réalité seules des applications très simples peuvent se contenter d'un tel raccourci. Il faut faire intervenir un troisième bloc : le **ViewModel** (Modèle de Vue).

Il arrive en effet souvent que les données du Modèle ne puissent pas être liées directement à la Vue, malgré la présence des convertisseurs sous Xaml. L'IHM peut aussi avoir besoin d'effectuer des traitements sur les données ce qui, raisonnablement, ne peut être placé ni dans la Vue (horrible retour au code spaghetti !) ni dans les données (les procédures stockées ont des limites de puissance et de portabilité).

Au final on a besoin d'un endroit *ad hoc* pour stocker les états de l'IHM et faire des traitements sur les données (en entrée ou en sortie).

[5] Le « binding » consiste à lier la propriété d'un objet à celle d'un autre objet de façon déclarative dans Xaml. On pourrait parler de ligature, de lien, mais le binding de Xaml est suffisamment particulier pour lui réservé un terme à part. L'anglicisme « binder » (prononcer *baïndé*) me semble plus chargé de sens que lier, ligaturer ou rabouter… Certains penseront autrement, ils en sont libres, cela va de soi.

C'est là que nait le **ViewModel**. Il est responsable de toutes ces tâches qui ne peuvent être placées ni dans la Vue ni dans le Modèle. Mais attention : ce n'est pas un fourre-tout pour autant, son rôle est clairement défini !

Toujours selon Gossman, le terme « ViewModel » signifie « *Model of a View* », le modèle d'une vue (que je simplifie dans cet article en **« Modèle de Vue** »). Le ViewModel peut être compris comme **une abstraction de la Vue**, mais il peut aussi être vu **comme une spécialisation du Modèle** que la Vue peut utiliser pour son data binding. Dans ce dernier rôle, le ViewModel contient des transformations de données qui convertissent les types du Modèle dans d'autres, plus pratiques pour la Vue, et il contient aussi des **Commandes** que la Vue peut utiliser pour interagir avec le Modèle.

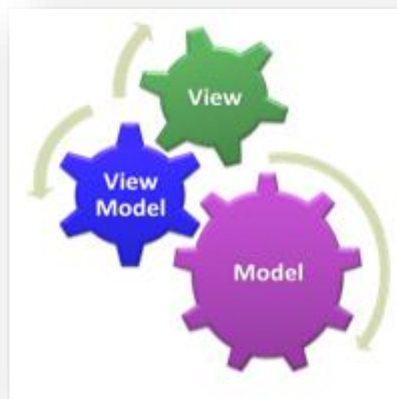


Figure 6 - Les rôles dans M-V-VM

Un autre ingrédient vient d'être ajouté au pattern : **la notion de commande**.

Les commandes sont un point essentiel du pattern M-V-VM. Toutes les autres patterns ignorent ce problème ou bien certaines de leurs variantes se basent sur des mécanismes plus ou moins sophistiqués (ou au contraire totalement désuets comparés à l'effort conceptuel de ces mêmes patterns). M-V-VM est le premier pattern de cette famille à prendre en compte l'impératif d'un système de commande pouvant utiliser le binding Xaml pour arriver à une grande efficacité tout en restant un modèle simple à appliquer.

> *On appelle ici « commande » toutes les actions que l'utilisateur peut réaliser depuis l'interface, c'est-à-dire les clics sur des objets, la saisie de texte, les « gestures » d'un stylet ou même les mouvements de doigts sur un écran tactile. En programmation classique toutes ces actions donnent naissance à des événements déclenchés par les objets concernés. Les gestionnaires d'événements sont alors placés dans le code-behind de l'interface. Lorsqu'on dit que le Modèle de Vue (ViewModel) gère les commandes cela signifie que ces dernières sont représentées par des points d'entrée (des variables de type ICommand) dans le Modèle de Vue. De fait la Vue n'utilise plus les Events des objets mais le binding pour se lier aux commandes définies dans le Modèle de Vue. Nous reviendrons plus loin sur cet aspect essentiel.*

Bien entendu M-V-VM est né de l'esprit d'un architecte de WPF et c'est avec ce qu'il voyait dans Avalon et l'immense potentiel de cette technologie que Gossman a pensé M-V-VM.

En général les design patterns sont toujours très éloignés de l'implémentation. Elles se parent d'explications et de dessins UML, et si des exemples de code sont fournis ils sont parfois même écrits en pseudo-code pour éviter tout « mariage » avec un langage ou une technologie plutôt qu'une autre. *Les design patterns sont universelles par principe*. Parfois elles sont plus aisées à mettre en œuvre avec certains langages que d'autres, mais cela n'est que marginal et anecdotique.

M-V-VM est un design pattern qui est née en même temps que la technologie qui la rend réellement utilisable, dans la tête d'un des pères de cette technologie. C'est quelque chose d'assez rare dans le monde des design patterns. M-V-VM est certainement applicable à d'autres plateformes de développement, mais en s'appuyant sur le binding Xaml et le système de commande de WPF elle créée un filtre sélectif interdisant son utilisation « tel quel » en dehors de WPF et Silverlight. Un design pattern « local », donc non « universelle » ? Est-ce encore un vrai design pattern dans ce cas ? Laissons ce débat aux spécialistes !

C'est aussi pourquoi M-V-VM est connu dans le monde .NET et plus particulièrement sous WPF et Silverlight. Les autres plateformes utilisant toujours encore les autres patterns présentées plus haut, notamment M-V-C (dans des variantes plus ou moins éloignées de l'original il faut bien l'avouer).

La figure 7 suivante montre les interactions entres les blocs de M-V-VM. On note que la vue peut contenir un peu de code-behind notamment pour gérer la logique propre à l'interface. Ce n'est pas une entorse à la séparation du code, c'est une conséquence logique de celle-ci. Tout ce qui concerne l'affichage et l'IHM en général est du côté de la Vue.

Le ViewModel code la « logique de présentation », il est lié à la Vue par data binding (dans le sens Vue -> ViewModel), les commandes utilisent le même mécanisme de binding. Le ViewModel émet des notifications à la Vue (principalement pour la prévenir de la disponibilité des données ou de changements intervenus dans celles-ci hors de l'IHM). Ces notifications sont le plus souvent « cachées » car gérées par le data binding sans intervention du développeur. Quant au Modèle il dialogue avec le ViewModel pour fournir les données. Il englobe la logique métier (Business logic) ce qui en fait en réalité un bloc assez gros pouvant être représenté par une chaine complète allant du Modèle à la base de données ou à des services distants via une couche BOL[6] et une couche DAL[7].

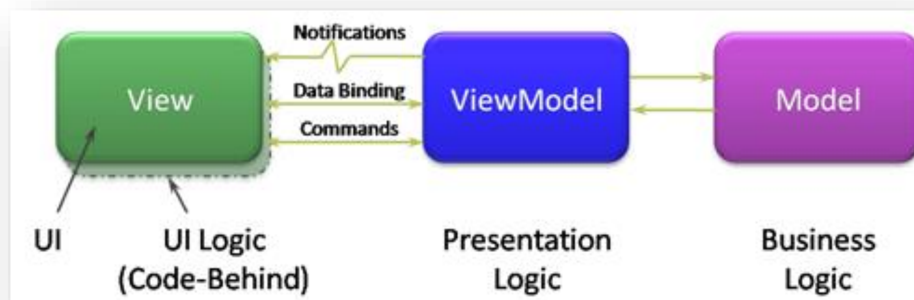


Figure 7 - Interactions dans M-V-VM

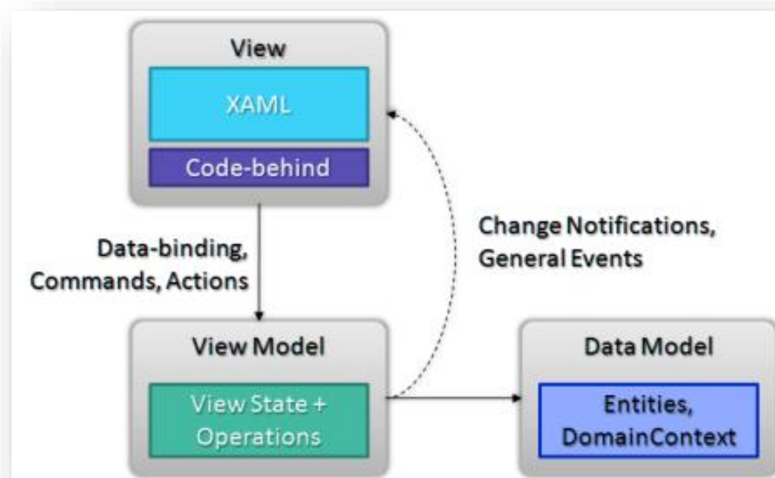


Figure 8 - Les tiers en jeu sous M-V-VM

[6] Business Object Layer : couche d'objets métier.
[7] Data Access Layer : couche d'accès aux données.

La figure 8 ci-dessus montre d'une autre façon les interactions entre les différents blocs de M-V-VM. On comprend bien que la vue est avant tout du Xaml complété si nécessaire d'un peu de code-behind et qu'elle se lie par data binding au ViewModel. Ce dernier stocke les états de la vue (autre aspect important du pattern) et concentre les opérations.

Le Model, ou Data Model, gère les entités, généralement des objectivations des données originelles.

Pour résumer

Modèle-Vue-Modèle de Vue (Model-View-ViewModel) est un design pattern proche de M-V-C mais s'en démarquant par de nombreux aspects. On lui trouve des liens avec d'autres patterns qui l'ont certainement influencé comme M-V-P ou V-P.

Parmi les différences essentielles on trouve la notion de commande et l'utilisation du data binding spécifique à Xaml. Ces nuances ont un impact sur les rôles de chaque bloc et la façon de les programmer.

M-V-VM fait intervenir la place du designer dans ses réflexions afin de mieux envisager sa collaboration active avec les développeurs. Elle offre ainsi un cadre méthodologique qui clarifie les modalités de cette collaboration dans les faits. De ce point de vue elle est presque un acte fondateur, le top départ que certains attendaient ne sachant pas trop comment s'y prendre pour intégrer le travail du graphiste dans une organisation et une hiérarchie déjà bien rodée.

La figure suivante offre encore une vue sur M-V-VM mais cette fois ci en entrant un peu plus dans les détails et en répondant à des questions telles que de savoir où se placent les Data Templates, les définitions de style, les commandes…

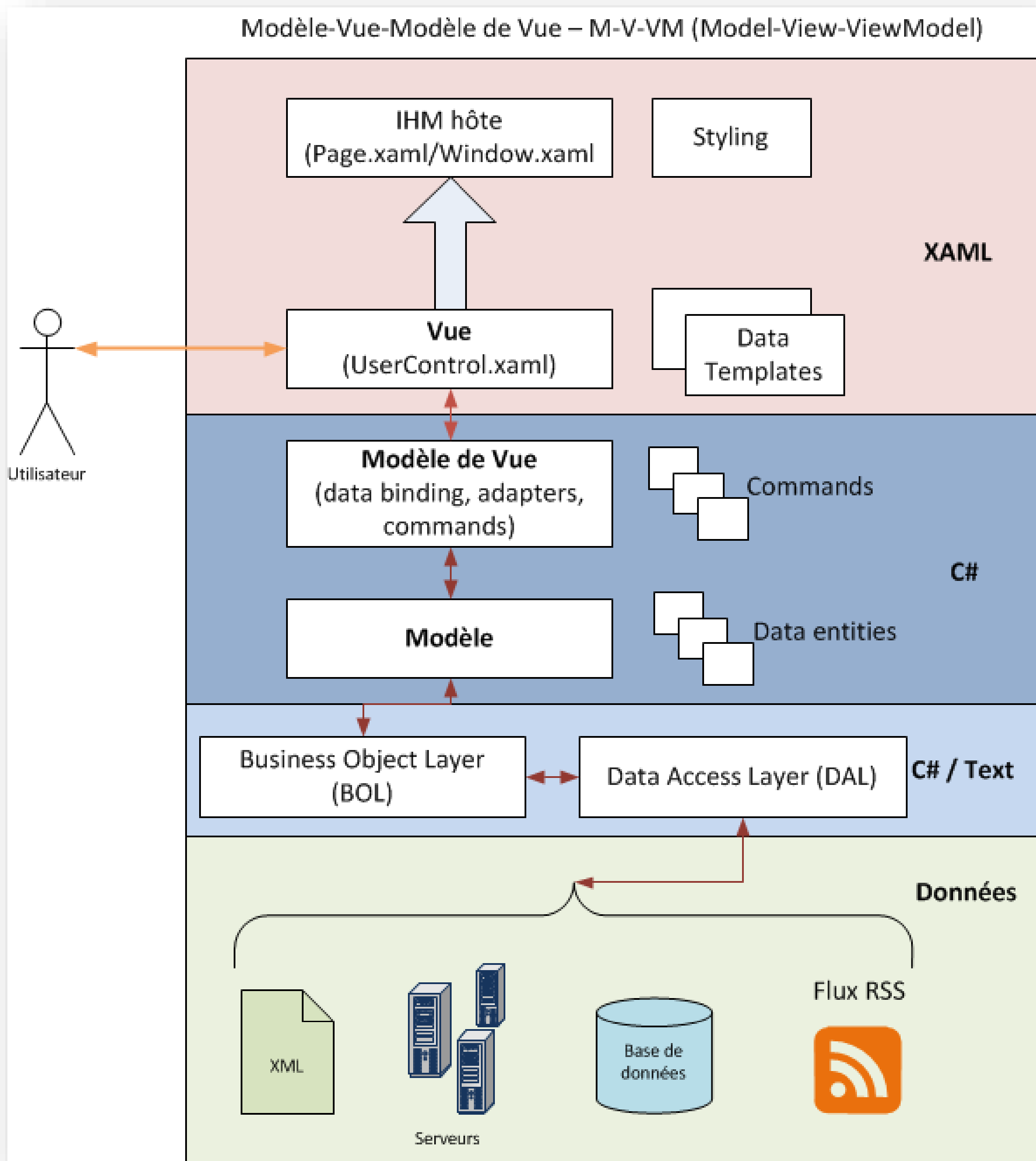


Figure 9 - M-V-VM

Le rôle de chaque bloc (illustré figure 9) se résume ainsi :

- La **Vue** (View) définit le visuel, elle est constituée principalement d'un fichier Xaml et se présente sous la forme d'un UserControl dont le fichier de code-behind est vide (ou presque). Le DataContext de la vue est initialisé pour pointer une instance d'un Modèle de Vue. La Vue définit aussi les Data Templates dans le sens où il s'agit d'éléments de présentation des données (visuel). En général la Vue est hébergée au sein d'une application hôte qui propose le Styling global (définition des dictionnaires

de styles).

- Le **Modèle de Vue** (ViewModel) est une abstraction de la vue, il contient tout le code nécessaire au fonctionnement de l'interface : stockage des états, formatage des données, transcodage, persistance des données, mais aussi toutes les commandes de l'interface (ex : réponses aux clics). Le Modèle de Vue est totalement indépendant de l'interface visuelle.

- Le **Modèle** représente les données de l'application. Dans la version la plus simple le Modèle peut être un fichier XML. Dans la réalité le Modèle contiendra des requêtes à des services de données (au sens large). De fait, le Modèle utilise généralement des couches intermédiaires comme un BOL et un DAL, mais il peut aussi bien dialoguer directement avec un SGBD. Ce sont d'autres design patterns et d'autres bonnes pratiques qui imposeront la présence d'éventuelles couches supplémentaires.

Avantages de M-V-VM

L'utilisation de M-V-VM a plusieurs avantages, le premier c'est d'être taillé sur mesure pour WPF et Siverlight. Plus techniquement :

- Le pattern offre une totale isolation entre l'interface et le code fonctionnel
- Chaque bloc est testable de façon isolée

De cela découle des gains bien concrets. Par exemple la séparation nette et totale entre l'interface définie en Xaml et le code application permet enfin de faire réellement collaborer les designers et les développeurs. Le travail des uns et des autres pouvant se faire en toute indépendance, sans obliger la présence de ces deux acteurs au même endroit.

> *Le rôle de l'intégrateur, utilisateur de Expression Blend, ayant à la fois des compétences d'infographiste et de développeur pour être capable de « souder » la partie graphique au code, était un profil improbable. Pourtant les outils forçaient le constat de l'obligation d'un tel rôle dans une équipe. Nous savons tous que l'industrie n'aime pas dépenser de l'argent… Qu'il faille un designer est déjà pour certain un déchirement. Alors, inventer un profil d'intégrateur, rare donc très cher, cela représentait un frein certain à l'utilisation de WPF. M-V-VM simplifie les rôles et fait disparaître l'obligation d'un profil d'intégrateur. En ce sens promouvoir M-V-VM est au moins aussi important que de plébisciter WPF ou Silverlight. C'est l'outil qui permet enfin d'industrialiser les développements sous ces plateformes. C'est pour cela que j'insiste « un peu » (beaucoup ?) sur cet aspect…*

L'indépendance de l'IHM offre aussi l'avantage de pouvoir modifier celle-ci à tout moment sans jamais risquer de créer le moindre bogue dans l'application. Dans le pire des cas le

binding d'une commande sera oubliée, la réponse graphique à un état ne sera pas conforme aux attentes, mais jamais le code ne sera perturbé et le bogue sera très visible et facilement repérable.

On peut aussi imaginer des applications offrant des vues totalement différentes sur les mêmes données (donc utilisant le même ViewModel), la sélection pouvant être dynamique à l'exécution.

Le fait que le Modèle de Vue intègre aussi la gestion des commandes est un atout formidable pour les tests unitaires : en effet, il devient possible d'écrire des tests complets d'une vue sans que celle-ci n'existe graphiquement ! Il suffit pour le programme de test d'invoquer les commandes du ViewModel et de lire les données attendues fournies par ce dernier. Cet aspect est un des points fort de M-V-VM alors même qu'on sait à quel point les tests unitaires jouent et joueront un rôle de plus en plus considérables dans la conception des logiciels.

Enfin, M-V-VM est suffisamment simple (mais si !) pour être mis en œuvre sans entraîner trop de code supplémentaire et suffisamment souple pour accepter des petites adaptations selon les projets.

En guise de conclusion je dirais que M-V-VM n'est pas une option… **C'est « la » façon de développer proprement des applications professionnelles**, s'en écarter est forcément une erreur. C'est un peu brutal je l'avoue, mais c'est fait exprès ☺ Plus vous serez familiarisé avec M-V-VM, plus vous en arriverez à cette même conclusion quand vous regarderez du code qui aura été écrit sans suivre cette pattern…

UTILISER M-V-VM

Une fois les concepts de M-V-VM expliqués, il reste à savoir comment on met en œuvre ces bonnes paroles dans une application…

Première chose à comprendre, faire propre, appliquer des méthodes éprouvées, tester, tout cela coûte plus cher en temps de développement que faire du code spaghetti sans aucun test.

Cela étant posé, M-V-VM ne coûte pas grand-chose à mettre en place une fois qu'on en a compris le principe. Ce n'est pas une méthodologie bien lourde et bien grasse (de type RUP par exemple), **c'est juste un design pattern qui facilite les choses**.

Dans un billet récent sur Dot.Blog (MVVM, Unity, Prism, Inversion of Control…) j'explique rapidement ce qu'est M-V-VM ce qui m'amène à introduire d'autres personnages dans l'histoire comme Prism, Unity Application Block, MVVM Light et à parler d'autres patterns

comme l'Inversion de Contrôle. Faisons ici un rapide point et voyons pourquoi ces acteurs peuvent jouer un rôle dans l'implémentation de M-V-VM.

Prism pour Xaml

Son nom exact est « patterns & practices : Composite WPF and Silverlight ». Il ne faut pas confondre cette version de Prism avec la très récente implémentation de même nom pour WinRT mais qui n'a rien à voir techniquement. Réflexion méthodologique, Prism est aussi une librairie de code aidant à la mise en œuvre des bonnes pratiques qui sont exposées. Vous pouvez télécharger Prism ici : http://msdn.microsoft.com/en-us/library/ff648465.aspx

Pourquoi parler de Prism ? Simplement parce que ce travail considérable explique comment développer des applications modulaires sous WPF et Silverlight et que parmi l'ensemble des bonnes pratiques qui sont présentées (illustrées par une application exemple aidant à comprendre le comment à partir du pourquoi) se trouve, par force, la séparation de l'IHM et du code fonctionnel.

> *Rappelons que Prism est issu de « patterns & practices » un laboratoire de recherche en méthodologie mis en place par Microsoft et que l'ensemble des travaux publiés par ce laboratoire sont en accès libre et ne coûte donc rien… On a aussi une garantie de validité du contenu, une pérennité des solutions proposées et une bonne synchronisation avec les avancées des nouvelles versions de WPF et Silverlight.*

Prism ne parle directement de M-V-VM, ce qui est étonnant, mais aborde le sujet en proposant une approche dérivée de M-V-C et surtout de M-V-P. Cette dernière est très proche de M-V-VM comme nous l'avons vu plus haut et les solutions proposées par Prism sont parfaitement acceptables, bien entendu. Il faut d'ailleurs noter que dans la version 2 de Prism sont apparus des termes tel que ViewModel et qu'un lien est donné vers un article MSDN « WPF Apps With The Model-View-ViewModel Design Pattern ». Dire que je vous conseille sa lecture tombe sous le sens.

Alors pourquoi insister sur Prism ? Parce que Prism expose des concepts essentiels et propose des solutions élégantes et que s'intéresser à M-V-VM sans connaître Prism serait certainement une erreur.

Prism propose (et explique) des implémentations pour gérer **l'Inversion de Contrôle** ou la **gestion des commandes**. Des aspects essentiels pour développer des applications modulaires, testable et dont les blocs sont bien indépendants les uns des autres. C'est-à-dire autant de choses qui rejoignent directement les préoccupations de M-V-VM.

Inversion de Contrôle

L'inversion de Contrôle permet d'éviter les dépendances entre les blocs d'une application. C'est une pattern complémentaire à M-V-VM qui s'occupe plus particulièrement de la séparation entre IHM et code métier. Les deux approches peuvent être utilisées simultanément pour en tirer les avantages cumulés.

Pratiquement, l'Inversion de Contrôle règle des problèmes bien concrets et vous allez tout de suite comprendre pourquoi cela rejoint les préoccupations de M-V-VM :

Prenons un code classique, avec une classe A qui utilise deux services (au sens large), Service A et Service B. Le schéma (rudimentaire) est donc le suivant :

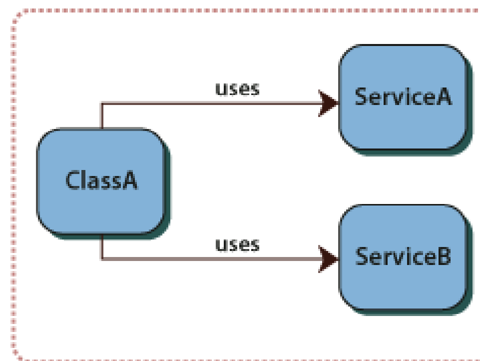


Figure 10 - Schéma classique des dépendances

Tout développeur un peu expérimenté connaît les désavantages de cette construction :

- Pour remplacer ou mettre à jour les dépendances de la classe A (les services) il faut changer le code source de cette dernière.
- Les implémentations concrètes des services utilisés doivent être disponibles à la compilation.
- La classe A est difficile à tester car elle code en dur des appels aux services. Il n'est par exemple pas possible de substituer à ces derniers des implémentations vides, des mocks[8] ou des stubs[9] pour effectuer des tests rigoureux.
- La classe A contient du code répétitif pour créer, localiser et gérer les dépendances.

Une telle situation arrive dès qu'on ajoute une référence à un projet et dès qu'un code pose un « using » sur des espaces de noms et se met à utiliser les classes de la librairie importée.

[8] Maquettes d'IHM dédiés aux démos ou aux tests.

[9] Squelettes non fonctionnels de code servant aux tests.

On peut généraliser ce problème à tout type de référence, même à des services Web, aux RIA .NET services, etc.

Pour régler les problèmes posés par les dépendances il existe un pattern générique : l'Inversion de Contrôle. Il se décline en d'autres patterns, deux sont implémentées dans Prism. Il s'agit du **Localisateur de Services** (Service Locator) et de **l'Injection de Dépendances** (Dependency Injection).

Injection de dépendances et Localisateur de Services

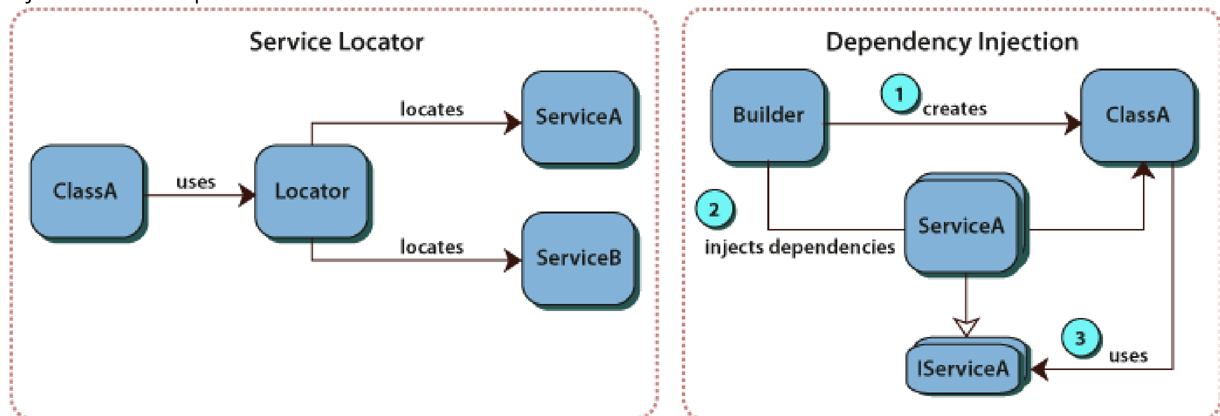


Figure 11 - Localisateur de Services et Injection de Dépendances

La figure ci-dessus montre les deux approches permettant de régler le problème des dépendances.

Ils sont très différents mais permettent d'atteindre le même objectif.

> *Nota : Comme vous pouvez l'imaginer, créer un dialogue propre et bien séparé entre les blocs de M-V-VM peut nécessiter de mettre en place d'autres patterns. L'inversion de contrôle fait partie de celles-ci. Dans l'exemple que je vais commenter plus loin j'utiliserai la solution du Service Locator pour créer une indirection entre les Vues et les Modèles de Vue afin d'éviter le couplage entre ces blocs. Mais nous verrons cela plus loin. C'est juste pour que vous sachiez que tout cela va servir à quelque chose et qu'il ne s'agit pas uniquement que du plaisir de s'instruire !*

Le Localisateur de Service

Comme on le voit figure 11, le Localisateur de Services s'interpose entre la classe A et les services A et B qu'elle consomme. Son rôle est très simple et explique son nom : Les services sont recensés (peu importe le mécanisme utilisé) par le Localisateur. Ce recensement permet au Localisateur de connaître les implémentations concrètes des services et d'en maintenir un dictionnaire. La clé peut être une simple chaine de caractère par exemple ce qui renforce encore le découplage. La classe A, lorsqu'elle doit utiliser l'un des services, demande au Localisateur de le lui fournir. C'est vraiment très simple.

Le Localisateur de Services évite que la classe A connaisse directement les implémentations réelles des services. Ces derniers n'ont pas besoin d'exister physiquement à la compilation de la classe A. On peut fabriquer des services de test mimant les services A et B (mais qui s'enregistre auprès du Localisateur avec la même clé) ce qui permet d'isoler le comportement de la classe A pour la tester parfaitement.

Il faut noter que dans ce pattern le Localisateur n'est qu'une sorte d'index, de « pages jaunes » des services. Il les recense et sait passer une référence aux classes qui les consomment. Charge à ces dernières de gérer les instances des services qui sont, le plus souvent (mais ce n'est pas une obligation), implémentées en suivant un autre pattern, celui du singleton.

L'objectif est donc atteint. La mise en œuvre est assez légère et Prism offre au travers de sa librairie une réponse toute faite, prête à être utilisée dans vos applications.

> *Nota : Ce pattern connaît des variantes. Parmi celles-ci notons que certains auteurs n'utilisent pas les interfaces mais des propriétés qui retournent les instances des services. Pour les tests il suffit de changer le Locator et de le faire pointer sur d'autres classes. De même le locateur peut jouer le rôle de créateur de points d'accès uniques (pattern Singleton), il peut créer un cache des instances créées pour les fournir plus rapidement ultérieurement. Etc.*

L'injection de dépendances

Ici les choses sont un peu plus sophistiquées. Les services ne sont pas disponibles directement, on les utilise uniquement via une interface qu'il publie. Par exemple le Service A implémente IServiceA. Seule cette interface est connue de la classe A qui consomme le service. Comme dans le pattern précédent il existe une sorte de catalogue des services, souvent sous la forme d'un fichier XML, qui associe à une clé le nom physique de chaque service.

Lorsque la classe A a besoin du service A elle appelle le « builder » qui lui va utiliser le catalogue pour charger dynamiquement le service A et renvoyer son interface à la classe A.

Pour cette dernière l'implémentation réelle du service est totalement cachée. Du point de vue de l'application le chargement dynamique est aussi un avantage (performance, modularité…).

En suivant ce pattern on arrive aux mêmes avantages puisqu'il est ici aussi possible de changer les implémentations des services sans avoir à modifier le code des classes les consommant. On peut donc fournir aussi, et très facilement, des services de test permettant d'isoler totalement le comportement des classes à tester.

Utiliser l'Inversion de Contrôle avec M-V-VM

Vous n'êtes pas obligé d'utiliser Prism ni l'Inversion de Contrôle pour mettre en œuvre M-V-VM. L'exemple que nous développerons plus loin le fait partiellement pour montrer comment cela est possible.

En revanche, pour des applications réelles ayant de nombreux modules (et M-V-VM en fait apparaître beaucoup : les vues, les modèles de vue, les modèles, qui s'ajoutent aux autres modules comme ceux du BOL ou du DAL) il semble indispensable de suivre l'une ou l'autre des patterns d'Inversion de Contrôle.

Vous pouvez bien entendu développer vos propres solutions basées sur ces patterns. Prism n'est qu'un exemple (utilisable en production) de mise en œuvre et de nombreux aspects que je n'aborde pas ici sont pris en compte dans la librairie de Prism. Cela en fait une bonne solution prête à l'emploi. Libre à vous de vous servir ou non !

Mixer l'Inversion de Contrôle avec M-V-VM permet d'atteindre une modularité maximale. Les deux patterns vont dans une même direction en mettant l'accent sur des aspects différents. Les associer est une stratégie habile !

La gestion des commandes (Commanding)

Si j'insiste ici sur Prism c'est qu'il prend en charge bien d'autres choses que l'Inversion de Contrôle et qu'il s'agit d'un travail considérable qui mérite qu'on s'y attarde, d'autant que les sujets traités nous concernent directement pour cet article. Prenons le problème de la gestion des commandes que nous avons déjà évoqué ici.

Pour rappel, il s'agit d'implémenter dans les Modèles de Vue (ViewModel) toutes les commandes (actions) qui répondent aux sollicitations de l'utilisateur. De fait cela impose de ne plus coder les événements de type « click » d'un Button dans le code-behind de la Vue. Ce code est dans le Modèle de Vue associé. Pour assurer le lien entre l'événement original, par exemple le Click, il faut utiliser autre chose, quelque chose se présentant sous forme utilisable par le binding Xaml puisque la vue est reliée au Modèle de Vue par cette méthode (plus précisément la Vue est « databindée » au Modèle de Vue via sa propriété DataContext).

WPF intègre une gestion des commandes qui repose sur une interface, ICommand et de nombreux services s'occupant d'activer les commandes ou de gérer l'état actif / inactif de chacun (le Enabled à true ou false dans un Button par exemple).

Jusqu'à la version 4, Silverlight n'implémentait pas tout ce système, seule l'interface ICommand était présente, mais seule elle ne servait pas à grand-chose.

Prism étant un travail qui depuis longtemps s'est donné comme impératif d'être utilisable à la fois sous WPF et Silverlight (puis Windows Phone et WinRT) une solution globale a été apportée à cette disparité entre ces Frameworks (seul Prism pour WinRT diffère totalement en raison de la nature très particulière de Windows 8+ et de Modern UI / WinRT).

Silverlight 4+ implémente une partie de la gestion de commande et n'oblige plus à se reposer sur un Framework externe, comme Prism ou d'autres, pour gérer les commandes dans le Modèle de Vue. Toutefois Prism conserve tout son intérêt car ce qu'il propose est totalement portable entre WPF et Silverlight et s'intègre dans une logique plus vaste.

Avec Silverlight 3 il était indispensable d'utiliser une librairie comme Prism pour gérer les commandes. De nombreuses applications déjà développées ou en cours de développement peuvent encore utiliser SL3 (les migrations en cours de développement sont toujours risquées). L'utilisation de Prism est un moyen de s'assurer que ces applications évolueront correctement.

Les autres librairies à connaître

Outre Prism dont j'ai largement parlé plus haut, il existe plusieurs tentatives d'écriture de librairies permettant de mettre en œuvre M-V-VM sous Silverlight (et WPF) et, notamment, de combler les lacunes de SL 2 et 3 à propos de la chaîne de commande. Comme d'habitude, soit ces librairies sont trop légères et ne règlent pas tout, soit elles sont plus ambitieuses, comme Prism, mais réclament alors un investissement important pour les maitriser.

C'est pourquoi j'ai décidé aujourd'hui de **vous montrer par l'exemple** que grâce à Silverlight 4+ il est possible **de couvrir tous les besoins principaux** d'une application M-V-VM **uniquement avec les outils de base de la plateforme** et un peu de code.

Mais cela ne doit pas vous interdire de regarder du côté de certains framework, parfois très complets, et qui, dans tous les cas, pourrons vous inspirer pour trouver vos propres solutions. Certains pourront même vous séduire, n'en conseiller aucun ne signifie pas que je n'en trouve aucun digne d'être utilisé !

Silverlight.FX

Cette librairie peut encore se trouver ici : https://github.com/nikhilk/silverlightfx

Il s'agit d'une des premières tentatives, elle n'est plus mise à jour et n'a d'intérêt qu'historique (même si son code est toujours intéressant à étudier). Son but était d'autoriser la conception d'application SL2 et SL3 riches en fournissant des blocs et des contrôles qui favorisent une meilleure architecture.

MVVM Light Toolkit

Créer par Laurent Bugnion, il s'agit d'un toolkit minimaliste et léger permettant de mettre en œuvre M-V-VM sous Silverlight 3+.

On le trouve ici : http://www.galasoft.ch/mvvm/getstarted/

Caliburn

On peut télécharger Caliburn ici : http://caliburn.codeplex.com/

Caliburn se veut être une aide au développement d'application Silverlight et WPF. Il implémente plusieurs patterns orientés UI. Elles sont directement inspirées de M-V-C-, P-M (M-V-VM).

Caliburn était un framework très complexe, tellement que même son concepteur a senti le besoin de sortir « Caliburn.Micro » pour simplifier son utilisation. Petit à petit Caliburn est tombé en désuétude et c'est aujourd'hui Caliburn.Micro qu'il faut utiliser si on choisit ce framework.

L'intérêt de Caliburn aujourd'hui n'est donc plus qu'historique, tout comme Silverlight.FX, mais son code mérite encore d'être étudié.

Caliburn.Micro

C'est la version moderne et allégée de Caliburn. Reposant sur les mêmes principes il ne faut pas se faire abuser par le qualificatif « micro »… Même micro, Caliburn reste un framework complexe et un peu lourd. Toutefois certains développeurs le trouvent à leur goût et c'est une librairie bien développée. Le choix des frameworks est une excellente chose et prouve la vivacité de MVVM !

On peut télécharger Caliburn.Micro ici : http://caliburnmicro.codeplex.com/

Cinch

Cinch est un projet intéressant qui reste conçu pour WPF plus que pour Silverlight. On peut l'adapter au moins en partie et sa philosophie ne manque pas d'intérêt.

Toutefois il fut très peu utilisé et n'est présenté ici que pour l'intérêt de la diversité des impémentations de MVVM, chaque framework ayant sa « vision » des choses, ses choix d'implémentation et sa propre « zone de couverture » du pattern souvent très différente des autres librairie.

On le trouve (avec son code source) sur CodeProject en plusieurs articles (liens dans l'article d'introduction):

http://www.codeproject.com/KB/WPF/Cinch.aspx

Jounce
Jounce a été l'un de mes frameworks préféré pour Silverlight. Se reposant sur MEF, lui aussi offre une vision très personnelle sur le pattern MVVM. Son code et les solutions proposées sont très originaux et méritent toujours d'être étudiés.

Le fait qu'il ne cible que Silverlight lui a certainement interdit une plus large diffusion, ce qui est dommage.

On trouve Jounce ici : http://jounce.codeplex.com/

(Dot.Blog a aussi abordé ce framework en profondeur, voir le sommaire de ce PDF).

MVVMCross
Le plus original de tous les frameworks MVVM assurément. D'abord par son implémentation et sa façon de résoudre les problèmes posés par MVVM et ensuite par son unicité : MVVMCross est Cross-plateforme et fonctionne aussi bien en mode console que sous WPF, Silverlight, Windows Phone et WinRT mais aussi Android et iOS ! Pour ces deux derniers OS il utilise la puissance de MonoDroid et MonoTouch (Xamarin).

Il est présenté en détail notamment dans une série de 12 vidéos d'une durée totale d'environ 8H. Voir le sommaire de ce PDF pour la présentation de ces vidéos.

On retrouve aussi dans le présent document des billets présentant ce framework étonnant.

Il peut être téléchargé ici : https://github.com/slodge/MvvmCross

Préparation de la mise en œuvre
Le grand moment est venu… Préparez-vous car pour simple qu'est l'application de démonstration elle n'en est pas moins riche de multiples astuces et patterns qu'il ne faut pas louper !

Le projet
Le but du jeu n'est pas de créer l'application du siècle mais de trouver un prétexte permettant de vous montrer l'ensemble des problématiques soulevées par la mise en œuvre de M-V-VM et, bien entendu, les solutions correspondantes…

Je suis donc parti d'un petit fichier XML qui expose un diaporama. Ce fichier sera la source de données car je ne voulais pas compliquer à outrance en ajoutant des Web services. Le contenu décrit ainsi un diaporama classique : l'Url de l'image, son titre, et un rating (vous savez, les petites étoiles pour dire si on aime ou pas).

Partant de cette source de données j'ai conçu une application qui comprend une About box (whouaaa ! ☺) et une fiche permettant de visionner le diaporama.

Je vous entends dire « bon, en un quart d'heure maxi c'est bouclé ! ». Un quart d'heure d'informaticien déjà à la base ça fait bien deux jours, sans les pauses ni les RTT, tout le monde le sait… De plus cela ne prend pas en compte la foule des petits détails qu'il va falloir régler. C'est pour cette raison que l'estimation d'un quart d'heure de l'informaticien est vendu 15 jours et que, encore, il livre en retard ☺

Regardons le résultat pour savoir où nous allons :

L'écran d'accueil

Il s'agit d'une page découpée en trois parties horizontales :

- Un bandeau de commande (boutons)
- Un espace centrale qui accueille les fiches
- Un bandeau de bas de page indiquant le nom de la fiche en cours

C'est dans l'espace centrale que les vues sont incrustées. Au lancement de l'application c'est la fiche About box qui est visualisée.

L'écran principal est géré comme toutes les autres vues, via un ViewModel. Les problèmes qu'il pose sont un peu différents puisqu'il contient une surface d'accueil pour les vues. Nous étudierons ces différences.

La Vue AboutBox

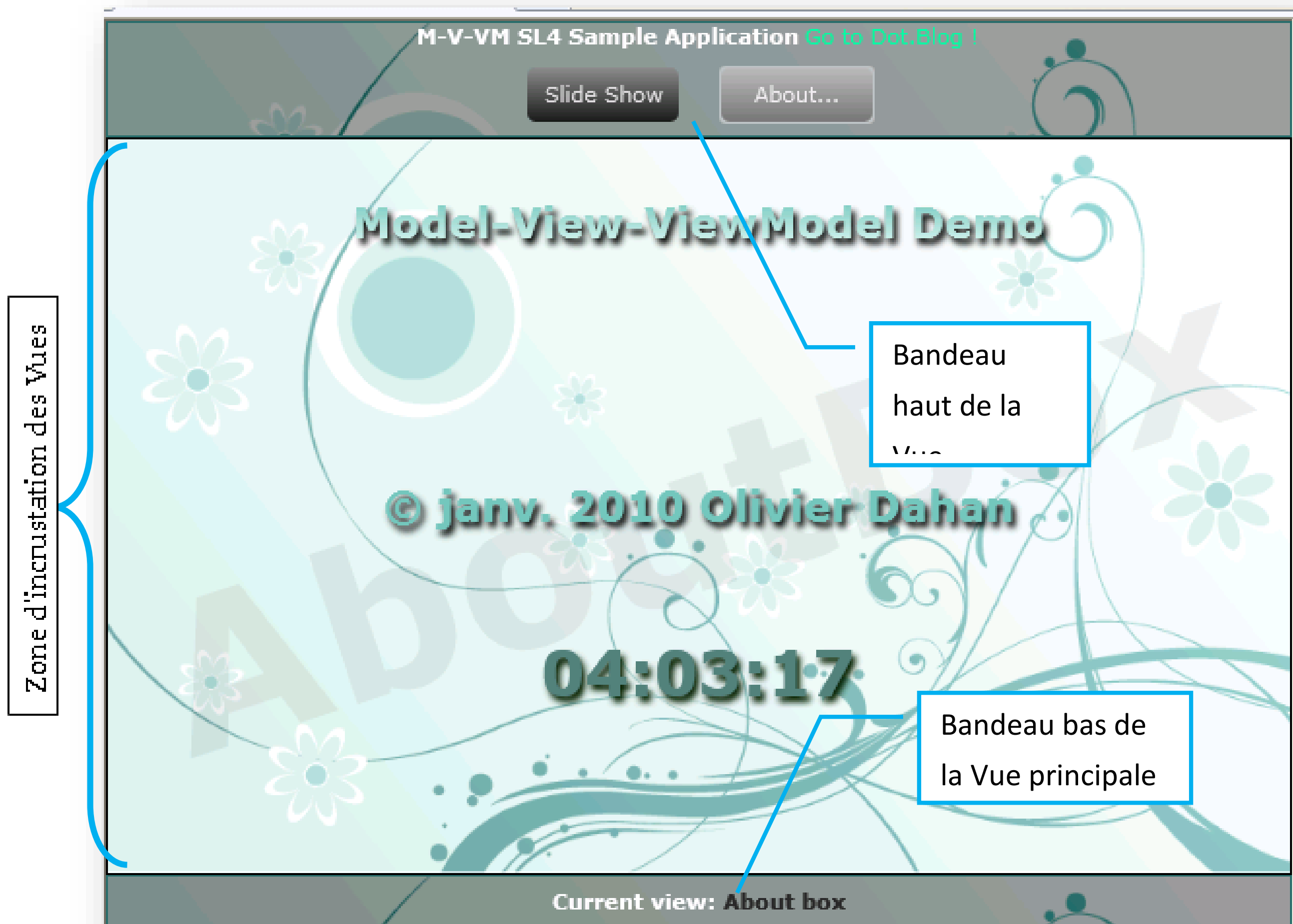


Figure 12 - L'écran principal de l'application exemple

C'est la vue affichée par défaut par l'application. Elle se compose d'un fond, d'un titre et de deux informations puisées du ViewModel : les copyrights et l'heure qui défile.

La vue AboutBox est la plus simple de toute, c'est elle que nous étudierons en premier.

> *Nota : Le projet fourni propose un troisième bouton de commande permettant de modifier les données du slide à partir d'une fiche navigable de type DataForm. Cela étoffe un peu la démonstration mais la Vue d'édition ne sera pas abordée dans cet article car elle n'ajoute rien par rapport au sujet abordé.*

Le diaporama

La fiche est ici aussi très simple, le design n'étant pas le sujet de cet article. On trouve ainsi un bandeau inférieur avec trois boutons de commande, et au centre la zone d'affichage de l'image en cours. Cette zone est surmontée par des indications (numéro d'image en cours,

nombre total d'images) et elle est fermée en bas par le titre de la photo ainsi que par l'affichage du rating.

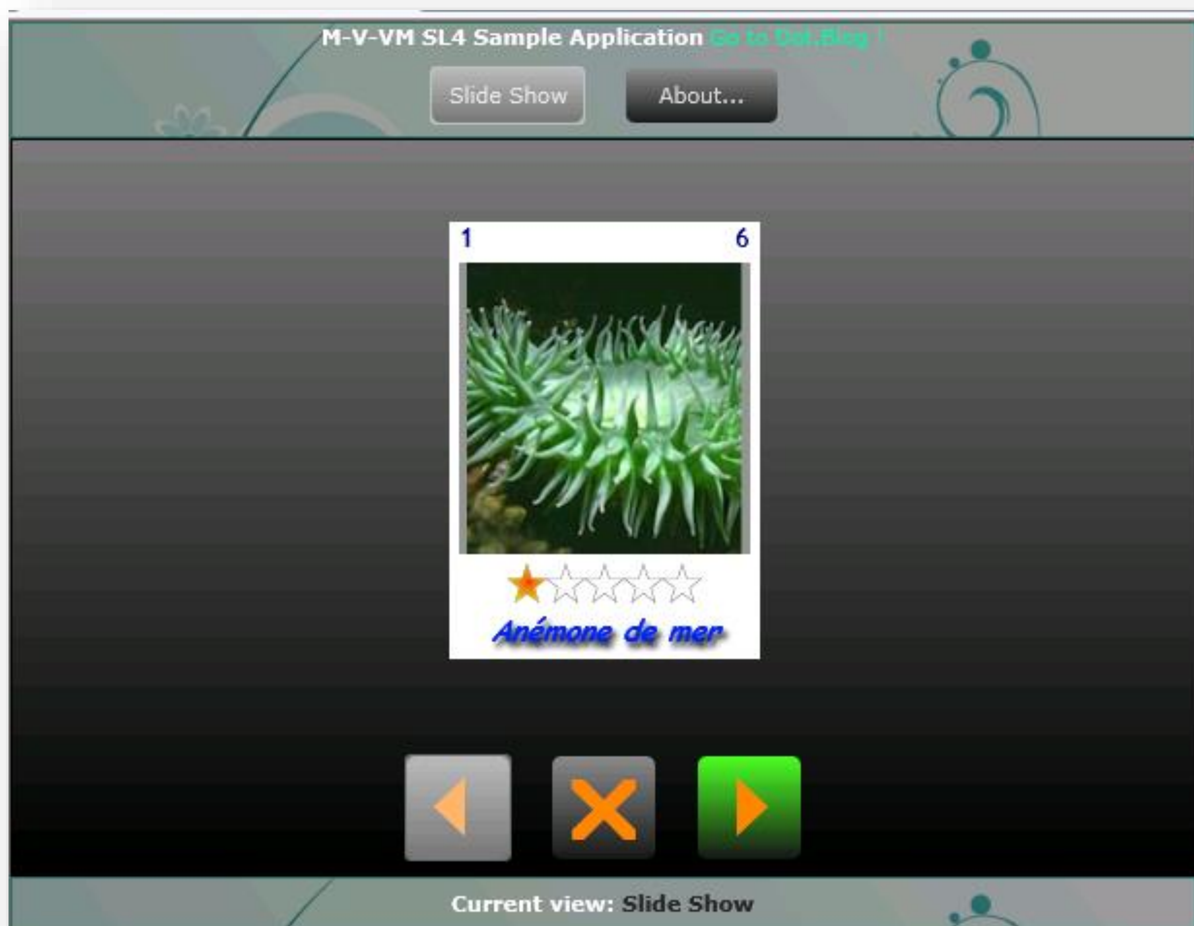


Figure 13 - L'écran diaporama

La vue diaporama est plus complexe. Elle ajoute deux problèmes nouveaux : la saisie (puisqu'il est possible de modifier le rating) et l'affichage d'un message de confirmation lorsqu'on clique sur la croix (remise à zéro du rating de la photo en cours).

L'équipement nécessaire

Comme pour la haute montagne, se lancer à l'assaut d'un tel design pattern impose de ne pas être équipé comme un touriste !

> *Nota : Si vous lisez cet article après la sortie officielle de tous ces outils, vous n'aurez bien entendu rien de spécial à faire et vous pouvez sauter la section suivante.*

Etant donné que j'ai choisi d'utiliser Silverlight 4 (notamment pour son implémentation de la gestion des commandes), cela signifie en pratique :

- Une machine virtuelle XP Pro (J'en tout un stock de VM en XP mais en Windows 7 c'est encore mieux).
- L'installation de Visual Studio 2010 et du Framework 4.0 (versions minimum)
- L'installation de Silverlight 4 (ou 5)
- L'installation de Blend 4 (ou 5)

Tout cela étant en version bêta au moment de l'écriture de cet article (d'où l'intérêt de la machine virtuelle, à l'heure de la mise en page sous forme de PDF tout cela peut se faire en SL5 avec VS2012 et le Blend fourni avec).

Organisation disque

Au fil du développement la solution ressemble à cela :

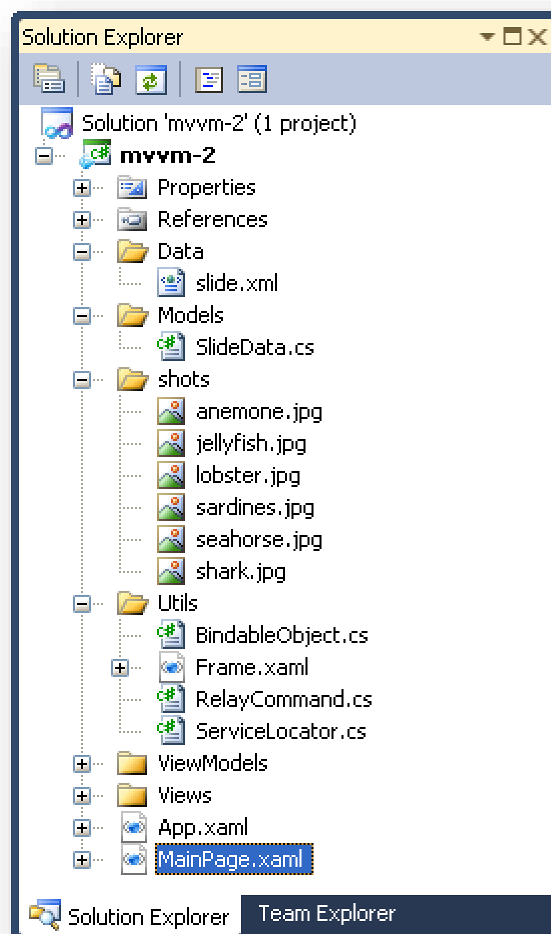


Figure 14 - organisation disque du projet

La solution contient un seul projet de type Silverlight

- La solution s'appelle mvvm-2.

> *Comme on le constate, il n'y a pas de projet de tests. Il s'agit d'un choix visant à conserver une taille raisonnable à cet article. La mise en place de tests est dans la réalité quelque chose d'indispensable, bien entendu.*

Les sous répertoires sont les suivants :

- Data, qui contient le fichier de données XML
- Models, qui contient le seul Modèle de l'application
- Shots, qui contient les images du diaporama
- Utils, qui contient du code utilitaire
- ViewModels, qui contient tous les ViewModels de l'application
- Views, qui contient toutes les Vues de l'application
- Racine : elle contient les éléments classiques comme App.xaml mais aussi la fiche principale MainPage.xaml.

Plan de route

Plutôt que de passer en revue chaque morceau de code, je vous propose de progresser dans l'ordre dans lequel j'ai réalisé l'application de démonstration.

Cela pourra paraître un peu brouillon, mais je pense au final que l'approche est plus pédagogique, vous pourrez ainsi suivre le fil de la mise en œuvre dans le respect de la séquence réelle.

Here we go !

MISE EN ŒUVRE

Tout est prêt pour commencer le travail. La première chose consiste à créer un projet Silverlight. La démonstration utilise un projet simple, sans projet Web, inutile ici.

La source de données

Pour la démonstration je suis parti de l'idée d'un diaporama qui serait représenté par un fichier XML décrivant chaque photo, fichier accompagné des images.

Le fichier XML

Voici le fichier exemple complet :

Slide.xml

```xml
<slide>
		<shot>
			<url>anemone.jpg</url>
			<title>Anémone de mer</title>
			 <rank>1</rank>
		</shot>
		<shot>
			 <url>jellyfish.jpg</url>
			<title>Méduse</title>
			<rank>3</rank>
		 </shot>
		<shot>
			<url>lobster.jpg</url>
			 <title>Langouste</title>
			<rank>5</rank>
		</shot>
		<shot>
			 <url>sardines.jpg</url>
			<title>Sardines</title>
			<rank>4</rank>
		</shot>
		<shot>
			<url>seahorse.jpg</url>
			 <title>Cheval de mer</title>
			<rank>3</rank>
		</shot>
		<shot>
			<url>shark.jpg</url>
			<title>Requin blanc</title>
			<rank>2</rank>
		</shot>
</slide>
```

Ci-dessous une autre vue du même fichier

	url	title	rank
1	anemone.jpg	Anémone de mer	1
2	jellyfish.jpg	Méduse	3
3	lobster.jpg	Langouste	5
4	sardines.jpg	Sardines	4
5	seahorse.jpg	Cheval de mer	3
6	shark.jpg	Requin blanc	2

Figure 15 - Structure du fichier de données XML

Le champ "url" contient l'adresse de l'image. Dans notre application nous stockerons les images dans un sous-répertoire « **Shots** » et c'est le code du ViewModel qui aura en charge de créer le chemin d'accès réel.

> *L'Url pourrait contenir des références à des images stockées sur un serveur dans une version réelle de l'application. Ici les images ont été intégrées au projet et pour ne pas le faire trop grossir il s'agit de Jpeg de petite taille (environ 200x200).*

De même, le fichier XML sera contenu dans le sous-répertoire « **Data** » du projet et intégré aux XAP.

> *Dans la réalité le visionneur de diaporama chargerait dynamiquement le fichier XML depuis un site serveur (qui stockerait aussi les images à afficher). Pour être encore plus exact, il y aurait un mécanisme de découverte des différents fichiers XML présents sur le serveur afin de permettre le visionnage de plusieurs diaporamas différents. Ce qui imposerait le développement d'un service Web puisqu'une application Silverlight ne peut pas réaliser une telle découverte à distance.*

Pour notre démonstration j'ai simplifié à l'extrême ses aspects secondaires par rapport au sujet abordé.

Le Modèle

Voici la première brique essentielle de notre construction M-V-VM : le Modèle. C'est lui qui a en charge la gestion des données, leur accès et leur persistance. Comme expliqué dans la première partie de cet article il peut se résumer à un simple fichier XML. Cette solution n'est que très rarement acceptable dans la réalité. De fait comme tout projet nécessite l'implémentation d'un Modèle (voire de plusieurs) nous allons en mettre un en place.

Il existe ainsi un sous-répertoire « **Models** ». Dans celui-ci nous créons un fichier de code « **SlideData.cs** ».

> *Nota : A partir de maintenant il sera certainement plus simple de suivre cet article si vous l'affichez sur un écran et si vous ouvrez le projet sous Blend 4 ou VS 2010 sur un second écran.*

Le code sera assez simple puisque nous allons créer deux classes :

- L'une, statique, qui retournera une collection de « shot »
- L'autre classique qui définira justement la classe « shot »

La classe statique SlideData expose une propriété statique Slide. C'est une collection de type ObservableCollection dont l'avantage principal est de gérer les événements de changement de propriétés ou d'éléments dans la liste. On utilise ce type de façon préférentielle dès lors que la collection est liée à une interface visuelle par exemple.

SlideData, la source de données

Voici le code :

```
/// <summary>
  /// Retrives data from XML file and returns them as an observable collection
  /// </summary>
  public static class SlideData
  {
    /// <summary>
    /// Returns data from XML file.
    /// </summary>
    public static ObservableCollection<Shot> Slide
    {
      get
      {
        var q = from c in XElement.Load("Data/slide.xml").Elements("shot")
            select
              new Shot
              {
                Url = "/mvvm-2;Component/shots/" + c.Element("url").Value,
                Rating = Convert.ToInt32(c.Element("rank").Value),
                Title = c.Element("title").Value
              };
        return new ObservableCollection<Shot>(q);
      }
    }
  }
```

La méthode unique de la classe utilise une requête Linq to XML pour charger le fichier XML et générer une collection d'instances de type Shot. Je renvoie le lecteur vers mes nombreux billets[10] et articles à propos de Linq pour tout éclaircissement sur cette syntaxe.

La classe Shot

A la base une telle classe est extrêmement simple à mettre en œuvre. On pourrait même se contenter d'une structure possédant les trois champs nécessaires.

Cela serait valable fonctionnellement pour aller vite mais n'apporterait rien dans cette présentation de M-V-VM.

De fait, la classe Shot sera implémentée totalement dans les règles, même si certaines ne seront pas réellement utiles dans la démonstration.

BindableObject

Tout d'abord, s'agissant d'une classe décrivant des objets qui seront bindés à une interface visuelle, nous avons besoin qu'elle implémente les mécanismes de notification de

[10] http://www.e-naxos.com/Blog/?tag=/linq

changement de valeur des propriétés. En réalité *une classe bien écrite implémente toujours ce comportement* car on ne sait pas à l'avance s'il sera utile ou non, et il l'est le plus souvent…

Shot doit ainsi supporter l'interface INotifyPropertyChanged.

L'implémentation de cette interface ne pose pas de problème particulier et peut être fait directement dans la classe. Pour simplifier la tâche je vais en réalité faire hériter Shot de la classe BindableObject. Cette classe-utilitaire a été écrite originellement par Josh Smith. Elle se trouve dans le sous-répertoire Utils de l'application.

Elle implémente tout le nécessaire pour gérer la notification de changement de propriété qu'elle agrémente de quelques astuces comme la mise en cache des arguments pour éviter la fragmentation du tas.

L'utilisation de la classe BindableObject offre deux autres avantages :

- En mode debug les instances vérifient les noms des propriétés lors de la notification de changement
- La classe expose un second événement AfterPropertyChanged qui peut s'avérer utile dans certaines situations.

En effet, l'événement de notification de changement de valeur de propriété ouvre un gouffre béant aux bogues : le paramètre passé, le nom de la propriété, est une chaîne de caractères ! On perd ici tous les avantages d'un langage fortement typé.

Il est tout à fait possible de faire une faute d'orthographe, de majuscule mal placée ou un bête copier/coller non vérifié et de se retrouver avec une notification qui ne sert à rien… Notifier que « LaPropriété » vient de changer de valeur, alors que le nom réel dans le code C# est « LaPropreté » (un peu de dyslexie…) ne déclenchera pas toute la cascade prévue et créera un bogue assez sournois à dépister.

Il est essentiel lorsque vous créer des classes supportant INotifyPropertyChanged de vous assurer que les noms de propriétés passés sont bien identiques à ceux utilisés dans le code C#.

Comment s'assurer réellement que le code ne présente pas ce type de faille ?

C'est toute l'idée de BindableObject que d'offrir, en mode debug uniquement, un contrôle du nom passé en paramètre par le biais de la réflexion.

Des implémentations plus complètes existent, mais un tel niveau de sécurisation se paye cher par le ralentissement causé par la réflexion. Si cela vous intéresse je vous invite à

découvrir une solution assez sophistiquée mais assez élégante proposée dans le billet suivant (c'est ancien mais toujours intéressant) :

http://michaelsync.net/2009/04/09/silverlightwpf-implementing-propertychanged-with-expression-tree

IEditableObject

Tout objet qui sera soumis à une modification, par exemple dans une data grid ou autre, doit pouvoir supporter l'édition transactionnelle. C'est-à-dire que l'utilisateur doit pouvoir annuler à tout moment ses modifications même après avoir modifié plusieurs champs et tant qu'il n'a pas validé ou annulé la totalité de la session de modification.

Hélas, cela n'est pas automatique. L'interface gère souvent la touche Escape ou Ctrl-Z pour annuler une modification en cours. Cela ne concerne que le champ en cours de saisie généralement. Si d'autres champs du même objet ont déjà été modifiés il est trop tard.

Un objet bien écrit doit ainsi proposer un moyen de revenir en arrière sur la totalité des modifications. Il s'agit donc bien d'une transaction. Celle-ci se limite toutefois à une session de saisie : dès qu'un champ est modifié, l'objet passe en mode « édition en cours » et l'envoi d'une commande doit lui permettre d'annuler ou d'accepter en bloc toutes les modifications intervenues depuis ce moment.

Le Framework .NET propose pour cela l'interface IEditableObject. Cette interface est prise en compte par des objets comme la datagrid.

Pour mettre en œuvre cette interface il suffit de répondre au trois méthodes qu'elle introduit :

- BeginEdit()
- CancelEdit()
- EndEdit()

BeginEdit() place l'objet en mode édition, CancelEdit() annule toutes les modifications et retour à l'état « normal », EndEdit() valide les modifications et sort du mode d'édition.

L'idée première pour implémenter cette interface consiste à créer des champs doublons, des backups locaux de chaque champ interne. Par exemple si j'ai une propriété « Titi » dont le champ interne est « titi », l'idée serait de créer aussi « titi_backup ». Lorsque BeginEdit() est exécuté on copierait tous les champs dans leur équivalent de backup pour faire une sauvegarde de l'état de l'objet. CancelEdit() reprendrait les valeurs sauvegardées et quant à EndEdit() elle ne ferait rien.

Cela marche, bien sûr, mais encore une fois on est très loin de la démarche qualitative que nous nous sommes fixés…

En fait il existe un design pattern parfaitement adaptée à un tel cas, il s'agit de Memento (GoF[11]).

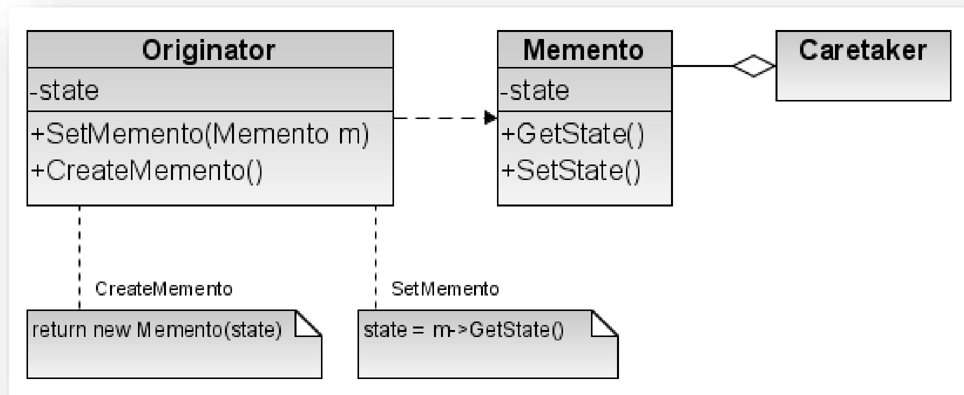


Figure 16 - Design pattern Memento (GoF)

Dans le schéma classique de cette pattern (voir figure 16), *l'Originator* est la classe souhaitant mémoriser son état (ou ses états). La classe *Memento* sert de conteneur sauvegardant l'état de l'objet (ou les états). La classe *CareTaker* se charge de conserver toutes les instances des Memento.

Pour mettre en œuvre l'interface IEditableObject nous utiliserons le principe de cette pattern sans en implémenter tous les détails.

De Memento nous allons conserver l'idée principale : les états de l'objet sont stockés dans l'instance d'une autre classe.

Ainsi, la classe Shot définit en interne une structure shotData :

```
internal struct shotData
    {
        internal string url;
        internal string title;
        internal int rating;
    }
```

En plus de cette structure elle définit trois champs :

[11] Le Gang of Four, abrégé en GoF, désigne les quatre informaticiens Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, auteurs des ouvrages "SmallTalk Best Pratice Patterns" et "Design Patterns : Catalogue de modèles de conception réutilisables" une référence. (Wikipédia).

```
private shotData data;
private shotData backup;
private bool inTxn = false;
```

Le premier représente les états en cours de l'objet. Toute manipulation des champs passe ainsi par l'indirection data.leChamp.

Le second champ privé déclare la sauvegarde, et le troisième est un drapeau nous permettant de savoir si l'objet est en mode édition ou non.

L'implémentation de IEditableObject se trouve ainsi être la suivante :

```
#region IEditableObject
    public void BeginEdit()
    {
      if (!InEditTransaction)
      {
        backup = data;
        InEditTransaction = true;
      }
    }

    public void CancelEdit()
    {
      if (InEditTransaction)
      {
        data = backup;
        InEditTransaction = false;
      }
    }

    public void EndEdit()
    {
      if (InEditTransaction)
      {
        backup = new shotData();
        InEditTransaction = false;
      }
    }
#endregion
```

InEditTransaction est une propriété renvoyant la valeur du drapeau inTxn avec un getter public et un setter privé.

> *Notez que le code utilise de simple affectations car les données sont stockées dans une structure, un type par valeur, et non dans une instance de classe (type par référence). C'est une nuance qui semble souvent échapper à de nombreux développeurs même expérimentés.*

A partir de cet instant les instances de la classe Shot supporte le mode d'édition transactionnel tel qu'il est mis à profit par exemple dans les datagrid.

Non indispensable dans cette démonstration, cette interface doit en réalité être implémentée quasi systématiquement si les objets peuvent participer à des modifications via l'IHM principalement.

Modèle, résumons

La partie Modèle de l'application est terminée. Elle est constituée de deux classes. La première décrit un Shot, une photo. Elle implémente plusieurs interfaces qui, en situation réelle, sont indispensables à ce type d'objet qui sera utilisé par l'IHM.

La seconde classe est statique et retourne un diaporama complet (liste de Shot) après lecture et analyse d'un fichier XML par le biais de Linq to XML.

> *On notera que notre exemple n'implémente pas de séquence de persistance des données. Le fichier de données étant intégré au XAP cela n'était pas directement possible. Il aurait été nécessaire soit de mettre en place un service Web, soit d'effectuer une copie de travail du fichier XML dans l'Isolated Storage de Silverlight. Des complications sans intérêt avec le but poursuivi par cet article.*

Les bonnes pratiques imposent de concevoir maintenant un projet de test validant le comportement du Modèle. Pour éviter de faire exploser la taille de ce (très gros) article nous ferons l'impasse sur les projets de test… Mais ils sont absolument indispensables, ne l'oubliez jamais et ne croyez pas gagner du temps en évitant de les créer, c'est un mauvais calcul !

Le Service Locator

M-V-VM n'impose pas directement l'utilisation du pattern d'Inversion de Contrôle présentée dans la première partie de cet article, mais les bonnes pratiques réclament de la mettre en œuvre pour mieux séparer les Vues des Modèles de Vue (ViewModel). Elle serait indispensable aussi pour séparer les ViewModels des Models s'il y en avait plusieurs, voire

pour isoler les Models des couches BOL[12] et DAL[13] qui pourraient se situer au niveau inférieur.

> *On se rappelle aussi que l'un des énormes avantages de M-V-VM est de permettre, enfin, une véritable séparation du travail entre informaticiens et infographistes. Mettre en place un Service Locator entre les Vues et les Modèles de Vue aide à gérer au mieux cette séparation indispensable.*

Je ne vous présenterai pas à nouveau le design pattern du Service Locator puisqu'elle a été discutée en première partie, mais rappelons brièvement son fonctionnement :

L'idée est de découpler un « appelant » de « l'appelé ». Un bloc de code doit communiquer avec un autre mais nous ne voulons pas que cette dépendance soit codée en dur.

L'injection de dépendance est une façon de mettre en œuvre l'Inversion de Contrôle, je l'ai jugée un peu trop complexe ici et lui ai préféré une autre solution, celle du localisateur de service (Service Locator) dans une version ultra simplifiée.

Les deux patterns sont proposées par Prism, et le ServiceLocator est aussi la solution retenue par Laurent Bugnion dans sa librairie « MVVM Light Toolkit » que vous trouverez à cette adresse : http://www.galasoft.ch/mvvm/getstarted/

Le projet de démonstration utilise ainsi un localisateur de service pour isoler les Vues des Modèles de vue. Il s'agit d'une implémentation très basique mais remplissant à merveille l'objectif.

```
public class ServiceLocator
{
    #region private field (ViewModel instances)

    private SlideShowViewModel slideShowVM;
    private AboutScreenViewModel aboutScreenVM;
    private MainPageViewModel mainPageVM;

    #endregion

    #region public access point to viewmodels

    /// <summary>
    /// Gets the SlideShow ViewModel property.
    /// </summary>
    public SlideShowViewModel SlideShowViewModel
```

[12] Business Object Layer : couche des objets métier.
[13] Data Access Layer : couche d'accès aux données.

```
    {
      get
      {
        if (slideShowVM == null)
          slideShowVM = new SlideShowViewModel();
        return slideShowVM;
      }
    }

    /// <summary>
    /// Gets the AboutScreen ViewModel property.
    /// </summary>
    public AboutScreenViewModel AboutScreenViewModel
    {
      get
      {
        if (aboutScreenVM == null)
          aboutScreenVM = new AboutScreenViewModel();
        return aboutScreenVM;
      }
    }

    /// <summary>
    /// Gets the MainPage ViewModel property.
    /// </summary>
    public MainPageViewModel MainPageViewModel
    {
      get
      {
        if (mainPageVM == null)
          mainPageVM = new MainPageViewModel();
        return mainPageVM;
      }
    }


    #endregion

  }
```

On peut difficilement faire plus simple ! Trois propriétés, une pour chaque ViewModel de l'application.

Chaque propriété renvoie ou créée le Modèle de Vue demandé. Cela donne un fonctionnement de type singleton (faux singleton, un vrai singleton implémente la pattern dans son propre code, ici cela serait de la responsabilité de chaque ViewModel).

Concernant cette démonstration le choix qui est fait est que toute fiche est créée lors de son premier appel. Ensuite elle reste vivante jusqu'à la fin de l'application. Selon le type d'application ce choix peut être parfait ou désastreux. On pourrait alors agrémenter le Service Locator de méthodes permettant de tuer les instances inutiles (MVVM Light propose une méthode Clear() pour chaque ViewModel dans son ServiceLocator par exemple). De même le choix du faux mode singleton ne s'adapte pas forcément à toutes les vues et à toutes les applications.

A vous de choisir ces paramètres selon les besoins de vos applications.

> *J'ai pris ici une liberté avec le design pattern original : Service Locator est sensé localiser les services uniquement. Ce sont les appelants à ces services qui doivent créer les instances et gérer le cycle de vie des services. Notre Service Locator s'occupe de tout cela. De même, la localisation des services se limite ici à exposer une propriété pour chaque type de ViewModel. Nous allons voir que cela est très pratique pour bénéficier d'un binding rapide et simple et accéder aux données en mode design. Un Service Locator plus sophistiqué pourrait permettre à chaque ViewModel de s'enregistrer dans une liste avec une clé par exemple et le binding devrait alors être effectués plutôt par code qu'en Xaml. Bref, c'est le principe qui compte, à vous de l'exploiter au mieux selon les applications à écrire. Dans ces dernières versions MVVM Light propose un conteneur d'Inversion de Control, il en existe d'autres implémentations plus complètes (comme Unity qui est parfait pour faire de l'Injection de Dépendance).*

Dernière petite chose, on pourrait se demander s'il ne serait pas plus judicieux de créer des interfaces pour les ViewModels et les Models. Avec le principe du Service Locator (qui peut être amélioré à votre guise) cela n'apparaît pas nécessaire. Les Vues et les ViewModels ne se connaissent pas, la magie du data binding via le DataContext évite de créer une isolation par interface. C'est le Service Locator qui fait tampon en quelque sorte. Si on doit fournir des mocks ou des stubs, il suffit pour les tests de modifier le Service Locator (voire de le compléter avec des sections de compilation conditionnelle). Mais là encore, à chacun de voir ce qui lui convient le mieux.

Utilisation du Service Locator

Le service locator est très facile à mettre en œuvre, au moins dans la version minimaliste exposée ici. Mais il est tout aussi facile à utiliser !

La première chose à faire consiste à créer une instance du Service Locator qui puisse être vue par toute l'application. Le plus simple et le mieux approprié consiste à créer une ressource dans App.xaml :

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="mvvm_2.App"
        >
  <Application.Resources>
    <vm:ServiceLocator xmlns:vm="clr-namespace:mvvm_2.Utils"
                                x:Key="ServiceLocator" />

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

L'instance ainsi créée sera disponible dans toutes les Vues de l'application. Prenons l'exemple de la fiche About Box (que nous verrons plus en détails dans un moment) et regardons comment elle est liée à son ViewModel au travers du Service Locator :

```
<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="mvvm_2.Views.AboutScreen"
        d:DesignWidth="640" d:DesignHeight="480"

DataContext="{Binding Source={StaticResource ServiceLocator},
                            Path=AboutScreenViewModel}"
  >
```

Comme le montre le code Xaml du UserControl « AboutScreen » ci-dessus, c'est au niveau de la déclaration de ce dernier que nous lions son DataContext par binding à la ressource statique ServiceLocator. Nous faisons pointer le Path sur AboutScreenViewModel, le nom de la propriété du Service Locator renvoyant l'instance du ViewModel.

Cette construction, outre d'être simple et de ne pas passer par du code C#, offre l'avantage de rendre disponible les données du ViewModel pendant la conception de l'application, que cela soit sous Blend ou Visual Studio. C'est très pratique pour affiner la mise en page. On constate aussi que la Vue ne connaît rien de son ViewModel, juste un nom de propriété dans le ServiceLocator. Si nous souhaitons retourner un autre ViewModel pour tester l'interface il suffit de changer la création de l'instance dans le getter de la propriété du ServiceLocator. De même on comprend que le ViewModel en sait encore moins sur la ou les vues qui se connectent à lui.

Il ne reste plus qu'à développer les vues et à effectuer les binding nécessaires des éléments visuels, sachant que ce binding sera relatif au DataContext global de la vue.

Dans un premier temps nous allons voir comment cela se passe avec la fiche About Box.

La Vue About Box

Il s'agit d'un UserControl classique que nous ajoutons au projet dans le sous-répertoire **Views** créé à cet effet. Le fichier s'appelle « AboutScreen.xaml ».

Une fois terminée (avec le binding en place) la fiche ressemble à cela (sous Blend 4) :

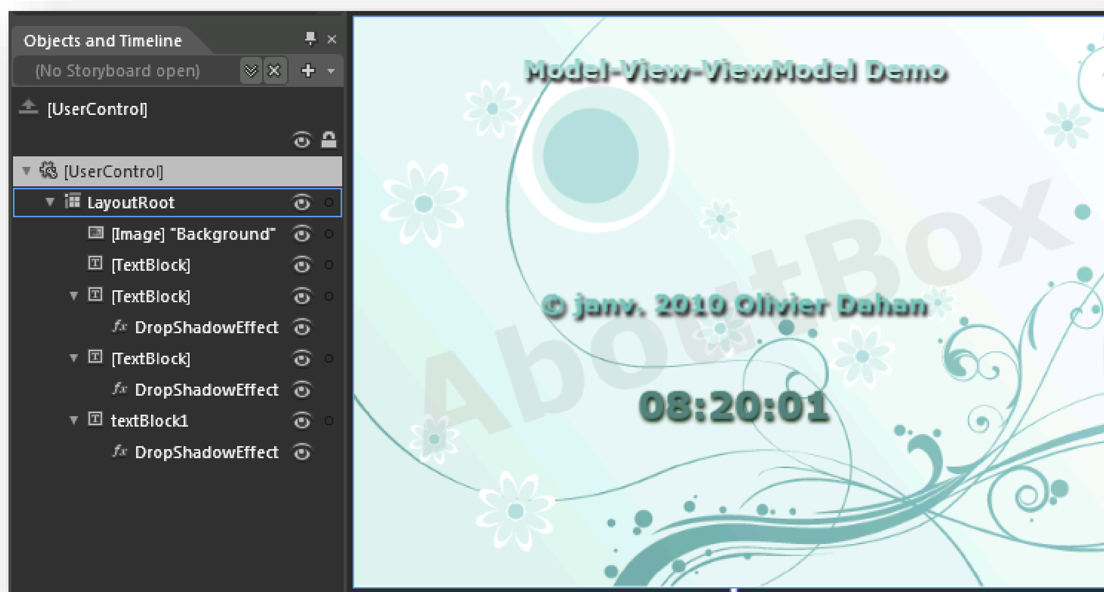


Figure 17 - La Vue About box

La vue est une fiche Silverlight tout ce qu'il y a de plus classique (donc un simple UserControl). Elle se compose d'une grille de fond (LayoutRoot) contenant une image, le mot « AboutBox » écrit en diagonal avec une opacity faible ainsi que de trois zones TextBlock agrémentées d'un effet de *Drop Shadow*.

Les deux derniers TextBlock vont être liés au ViewModel de la fiche car nous souhaitons d'une part que la date des copyrights s'adapte au mois en cours et que, d'autre part, une horloge soit affichée.

Le premier cas est le plus simple, le TextBlock doit être bindé une fois et en mode One Way à une propriété du ViewModel. C'est le B-A-BA du pattern M-V-VM.

Le second cas est déjà plus intéressant. En effet le Textblock de l'horloge sera lui aussi lié à une propriété du ViewModel, mais son contenu est modifié régulièrement. Il faut qu'un

mécanisme assure la notification de la Vue afin que celle-ci rafraichisse l'affichage du Textblock.

Avant de connecter tous ces éléments au ViewModel il nous faut créer ce dernier !

Le Modèle de Vue pour AboutScreen.xaml

Dans le sous répertoire **ViewModels** nous allons ajouter une nouvelle classe. Elle s'appellera AboutScreenViewModel.cs reprenant le nom de la vue suffixé par ViewModel.

Les ViewModel sont des abstractions des vues. Ils doivent fournir à ces dernières des propriétés directement « bindable » pour simplifier la mise en œuvre de l'ensemble. Si les données du Modèle doivent être adaptées, les ViewModel s'en chargeront. Ils peuvent le faire directement ou bien en s'accompagnant de Convertisseurs.

> *Dans une démarche visant à simplifier la collaboration entre designers et informaticiens je préconise la première approche. Il est plus difficile de demander à un designer de faire un binding complexe en Xaml intégrant l'appel à un convertisseur que de lui fournir une simple liste de propriétés à binder par leur nom directement.*

Une classe comme une autre

Un ViewModel est avant tout une classe comme une autre. C'est la façon dont on va s'en servir qui en fait un ViewModel et non pas un quelconque contenu technique plus ou moins sophistiqué.

> *Il est très important de noter que l'application d'une design pattern ou de façon générale d'une méthodologie est une activité symbolique et conceptuelle. Les objets manipulés et le code écrit sont « normaux » et exploitent les possibilités usuelles du langage et de la plateforme. Tout est dans l'intention.*

Pour faire fonctionner la vue AboutScreen le ViewModel doit ainsi proposé deux propriétés :

- Une chaîne de caractères fournissant le copyright
- Une chaîne de caractère fournissant l'heure courante formatée.

Pour le copyright vous devinez aisément le code :

```
public string Copyrights
    {
      get
      {
        return "© " +
          DateTime.Now.ToString("MMM") + " " +
```

```
            DateTime.Now.Year + " Olivier Dahan";
        }
    }
```

Sauf à supposer que l'utilisateur reste plus d'un mois entier sur le logiciel (ou bien qu'il se connecter à minuit moins une seconde du dernier jour du mois) on peut considérer que cette chaîne ne changera pas au cours d'une session de travail. Elle n'implémente donc aucun mécanisme de notification.

Pour l'heure les choses se compliquent un tout petit peu. Mais pas au niveau de la propriété qui est bien celle à laquelle on s'attend :

```
public string CurrentTime
    {
        get { return DateTime.Now.ToString("hh:mm:ss"); }
    }
```

Mais alors comment la Vue va-t-elle savoir quand se rafraichir ?

Le ViewModel déclare un Timer qui est mis en route dans son constructeur :

```
private DispatcherTimer clockTimer =
    new DispatcherTimer() { Interval = new TimeSpan(0, 0, 1) };

public AboutScreenViewModel()
    {
        clockTimer.Tick += clockTimer_Tick;
        clockTimer.Start();
    }
```

Bien entendu ce timer ne va pas rafraichir lui-même le TextBlock ! Sinon il n'y aurait plus aucun intérêt à M-V-VM…

La vue sera notifiée automatiquement, via le data binding et son DataContext, le ViewModel se contente uniquement de notifier le changement de valeur de la propriété :

```
void clockTimer_Tick(object sender, EventArgs e)
    {
        RaisePropertyChanged("CurrentTime");
    }
```

On voit ici que nous appelons RaisePropertyChanged, une méthode héritée de BindableObject, une classe étudiée plus haut que nous avons introduite dans notre exemple

pour ses multiples avantages et dont nous avons fait hériter AboutScreenViewModel. Bien entendu on peut se passer de cet artifice. Il faut alors que la classe AboutScreenViewModel implémente directement INotifyPropertyChanged et que le gestionnaire du Tick horloge déclenche l'événement PropertyChanged en passant le nom de la propriété horloge (CurrentTime).

Nom de la vue

Si vous vous rappelez les copies d'écran de l'application plus haut dans cet article vous aurez peut-être noté que le bandeau inférieur de la page principale affiche le nom de la vue en cours.

Comment la page principale (qui est une vue comme les autres) peut-elle accéder à une information cachée dans une autre Vue imbriquée ? Cela fait partie des petites spécificités de la Vue principale que nous étudierons plus loin. Mais pour résoudre le problème il a été nécessaire de déclarer une interface IViewInfo que voici :

```
namespace mvvm_2.ViewModels
{
   public interface IViewInfo
   {
      string Name { get; }
   }
}
```

IViewInfo déclare une propriété Name dont le rôle sera de retourner le nom de la Vue.

Pour cela toutes les vues secondaires (donc sauf l'écran principal) implémentent cette interface. Pour la Vue AboutScreen le code est le suivant :

```
public string Name
      {
         get
         {
            return "About box";
         }
      }
```

Le code complet du Modèle de Vue pour AboutScreen

Après les visions parcellaires, voici le code complet de la vue pour que vous puissiez situer les morceaux de code étudiés plus haut (je vous fais juste grâce des usings que vous retrouverez dans le code source fourni avec l'article) :

```
namespace mvvm_2.ViewModels
{
  public class AboutScreenViewModel : BindableObject, IViewInfo
  {
    public AboutScreenViewModel()
    {
      clockTimer.Tick += clockTimer_Tick;
      clockTimer.Start();
    }

    void clockTimer_Tick(object sender, EventArgs e)
    {
      RaisePropertyChanged("CurrentTime");
    }

    public string Copyrights
    {
      get
      {
        return "© " +
          DateTime.Now.ToString("MMM") + " " +
          DateTime.Now.Year + " Olivier Dahan";
      }
    }

    public string CurrentTime
    {
      get { return DateTime.Now.ToString("hh:mm:ss"); }
    }

    private DispatcherTimer clockTimer = new DispatcherTimer()
                  { Interval = new TimeSpan(0, 0, 1) };


    public string Name
    {
      get
      {
        return "About box";
      }
    }
  }
}
```

Un ViewModel n’est vraiment pas quelque chose de compliqué !

Binding de l’AboutScreen à son ViewModel

Lorsque nous avons étudié un peu plus haut le ServiceLocator nous avons étudié comment la vue était liée à son ViewModel par son DataContext via le ServiceLocator. Mais revoyons rapidement la syntaxe :

```
<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="mvvm_2.Views.AboutScreen"
        d:DesignWidth="640" d:DesignHeight="480"

DataContext="{Binding Source={StaticResource ServiceLocator},
                            Path=AboutScreenViewModel}"
  >
```

Dans la balise de description du UserControl « AboutScreen.xaml » nous trouvons ainsi un binding sur la ressource statique ServiceLocator (créée dans le fichier App.xaml) et plus précisément sur son chemin AboutScreenViewModel.

Par cette simple instruction la Vue AboutScreen se trouve liée à son Modèle de Vue (AboutScreenViewModel) mais par l'indirection du ServiceLocator (pattern d'Inversion de Contrôle).

La simplicité de ce binding fait que les développeurs préfèrent le taper directement en Xaml. Ce n'est pas forcément le cas des infographistes ou des développeurs qui aiment les automatismes !

Avec Visual Studio 2010 nous allons voir que tout peut être fait à la souris en quelques clics (il serait possible de faire le binding dans Blend aussi, outil plus pratique pour l'infographiste) :

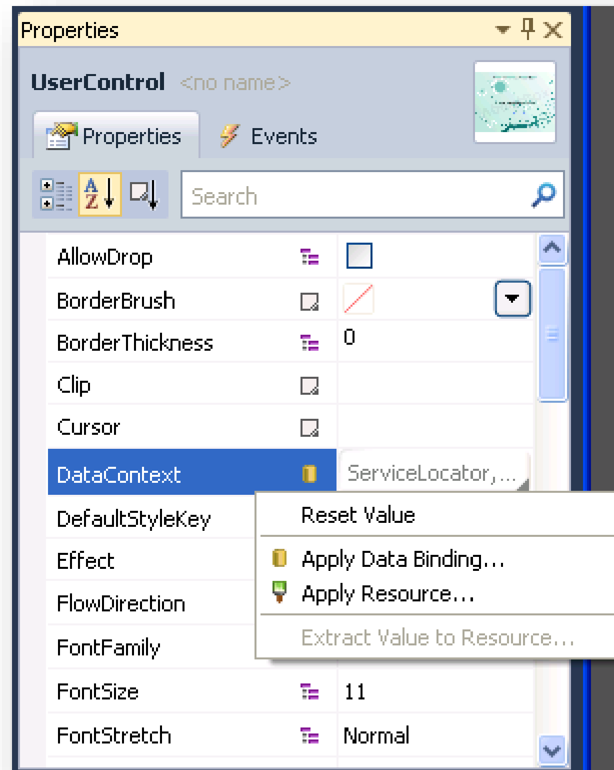


Figure 18 - Data binding du DataContext de la Vue au ViewModel via le ServiceLocator

Une fois le UserControl (la Vue) sélectionné, nous ouvrons le menu de binding de sa propriété DataContext.

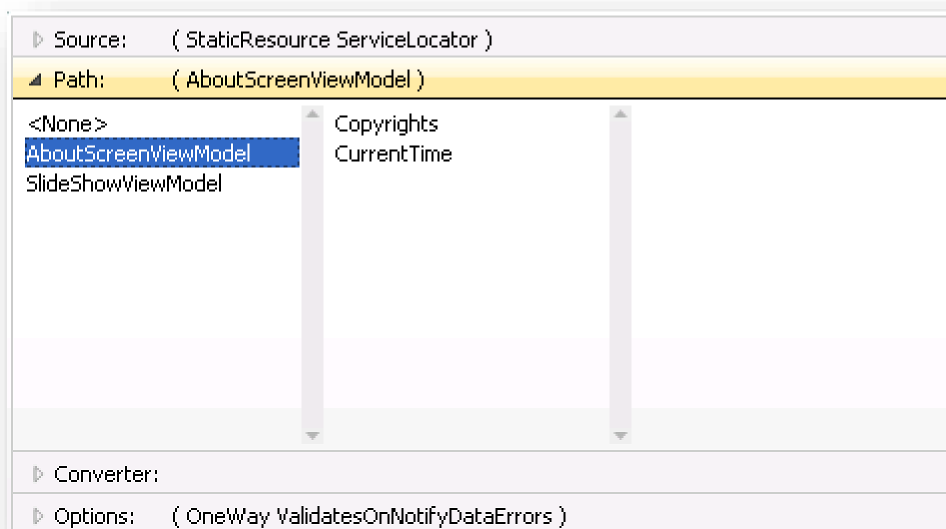


Figure 19 - Databinding du DataContext

Comme on le voit ci-dessus la source a déjà été renseignée (StaticResource ServiceLocator), nous venons d'ouvrir le volet « Path » pour choisir la propriété de la source qui sera liée au DataContext de la Vue. Visual Studio nous montre les deux propriétés existantes, nous choisissons AboutScreenViewModel. Dans le panneau central (à droite), nous pouvons même prendre connaissances des propriétés exposées.

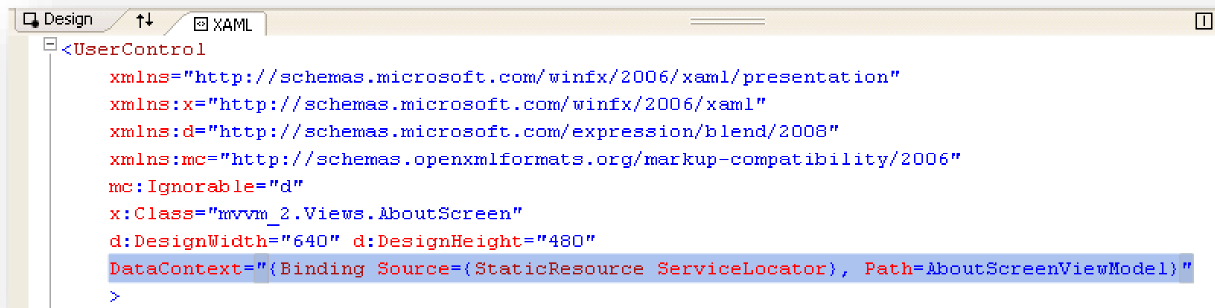


Figure 20 - Data binding du DataContext de la Vue – résultat

La figure ci-dessus montre le code Xaml généré par notre manipulation. Il est bien entendu identique à celui que nous avions écrit à la main, et c'est heureux !

A partir de ce moment, et si le projet a été construit (ses classes sont alors compilées), le binding des éléments de la Vue avec le Modèle de Vue s'effectue de façon très simple grâce aux nombreuses améliorations portées à Visual Studio 2010 en ce domaine, ou avec Blend 4. Les données sont même directement accessibles ce qui facilite la mise en page de la vue.

Prenons l'exemple du TextBlock indiquant l'heure et faisons la manipulation avec les deux environnements :

Data binding par Visual Studio 2010+

La première étape consiste à cliquer sur le TextBlock dans la surface de design.

Vous noterez que dans cette approche les composants visuels n'ont pas besoin d'être nommés, un gain de temps appréciable. Toutefois je vous conseille de donner des noms, ou de demander au designer d'en donner afin de lever toute possibilité d'ambigüité. Rappelons que M-V-VM est justement bien adaptée à la séparation du travail entre designers et informaticiens et qu'ils peuvent être amenés à travailler de façon totalement isolée sur un projet ce qui réclame malgré tout un peu d'organisation et une bonne communication.

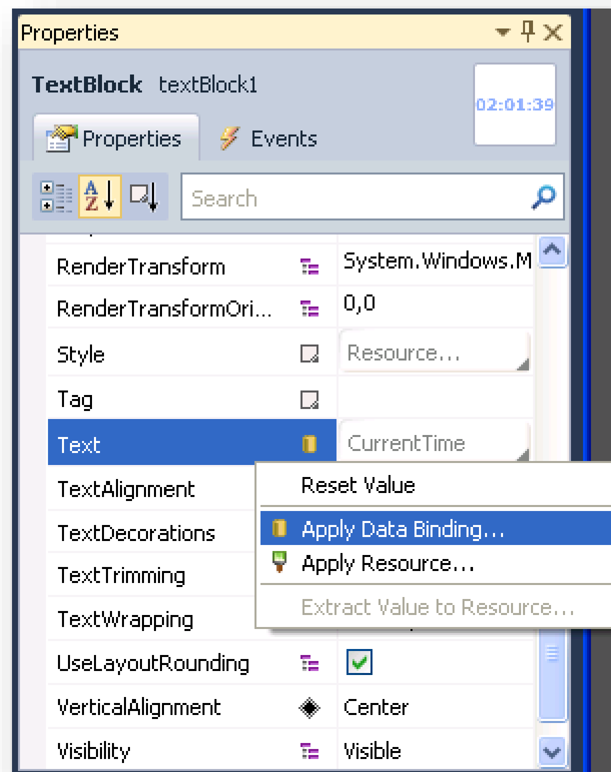


Figure 21 - Binding du TextBlock étape 1

L'étape 1 consiste à cliquer sur la propriété Text qu'on désire binder et à ouvrir le menu local du binding. Nous allons choisir « Apply Data Binding » (Appliquer le Data Binding).

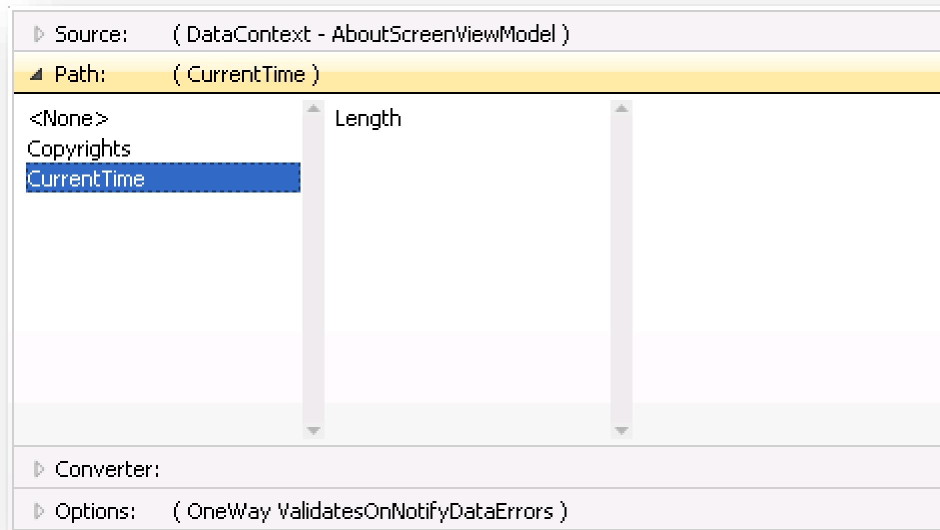


Figure 22 - Binding d'un TextBlocl étape 2

Comme le montre la figure ci-dessus le menu précédent ouvre une fenêtre très complète constituée de volets horizontaux repliables. En haut, la source du binding étant fixée nous pouvons en prendre connaissance (DataContext – AboutScreenViewModel). Difficile de faire une erreur donc puisque tout est écrit en clair. C'est l'avantage de la démarche que je vous propose dans cet article.

Le volet « Path » est ouvert et indique vers quelle propriété de la source pointe le data binding. Nous avons choisi « CurrentTime » dans la liste qui est proposée, ce qui est indiqué entre parenthèses dans la barre du volet Path.

On note la présence d'autres volets comme le « Converter » permettant d'indiquer un éventuel convertisseur de données, ou bien le volet « Options » qui indique pour l'instant un binding simple (OneWay ValidatesOnNotifyDataErrors).

Si la présentation est nouvelle, l'ensemble du data binding et de ses options est conforme à ce qu'on en connait. Nous n'inventons rien à ce niveau, M-V-VM utilise les briques existantes de Xaml.

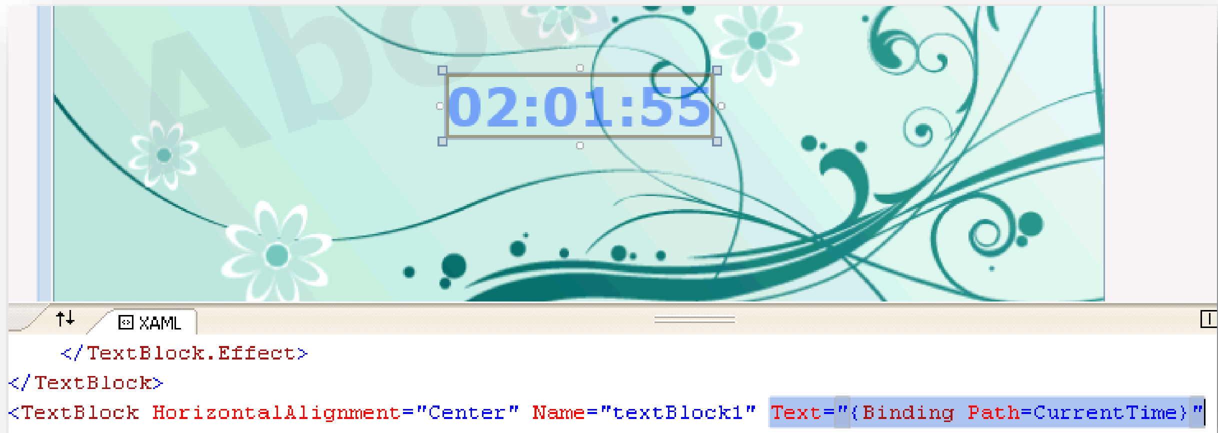


Figure 23 - Binding d'un TextBlock – code Xaml généré

La figure ci-dessus montre le code Xaml généré par l'opération de binding effectuée dans Visual Studio 2010. La partie surlignée en bleu est celle qui nous intéresse.

On voit que maintenant le TextBlock dans la surface de design contient une valeur, et ce que ne peut montrer cette page, que cette valeur change toutes les secondes.

L'ensemble de la chaîne mise en place Modèle – Vue – Modèle de Vue (M-V-VM) fonctionne parfaitement !

Data binding sous Blend

Nous allons refaire la même manipulation mais sous Blend.

Le principe de base est le même : sélection du TextBlock dans la surface de design (ou dans l'arbre de l'onglet « Objects and Timelines »), ouverture de l'onglet des propriétés, localisation de la propriété Text du TextBlock.

La figure ci-dessous montre cette première étape. On note que la propriété Text est entourée d'un cordon jaune. Cela signifie qu'elle est déjà bindée. Rien de plus normal puisque nous l'avons déjà bindée sous VS 2010 et que la manipulation que nous allons faire n'est qu'une redite.

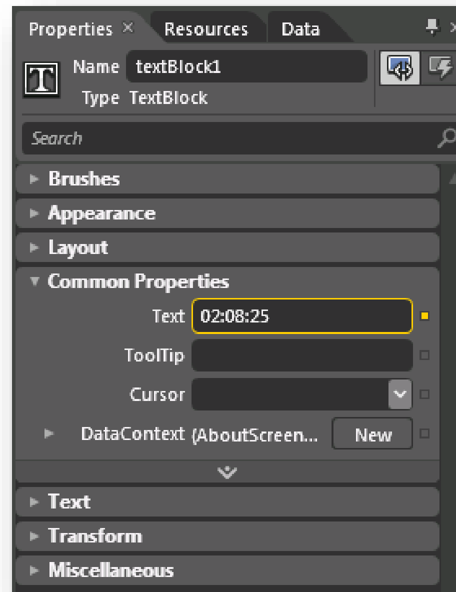


Figure 24 - Data binding d'un TextBlock sous Blend étape 1

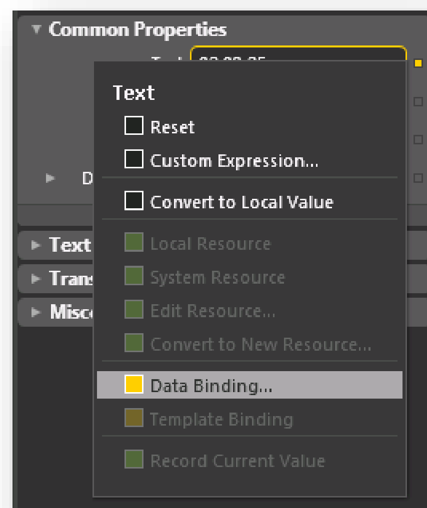


Figure 25 - Binding d'un Textblock étape 2

La seconde étape consiste à cliquer sur le carré de binding à droite de la propriété. Le menu qui s'ouvre permet de choisir l'option qui nous importe ici « Data binding ».

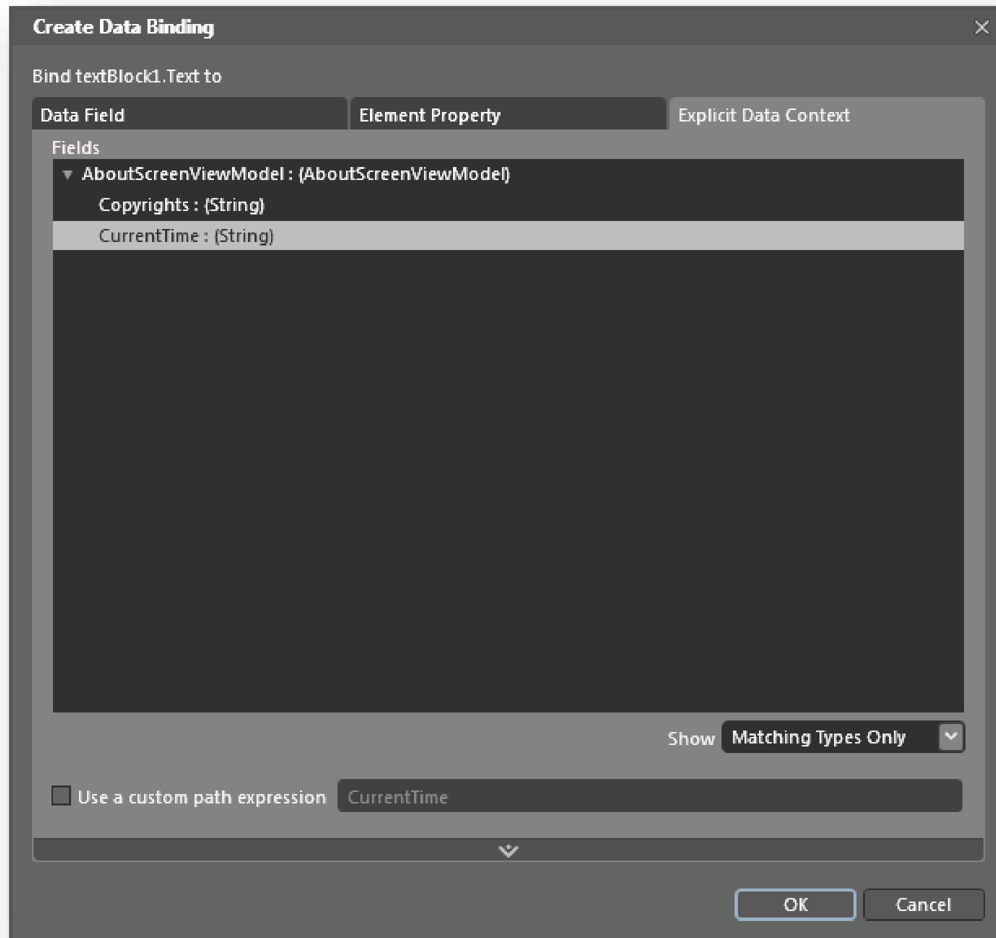


Figure 26 - Binding d'un TextBlock étape 3

La troisième étape consiste, comme sous Visual Studio, à sélectionner un champ du DataContext, ici CurrentTime.

Comme le montre la figure suivante nous obtenons bien le même résultat…

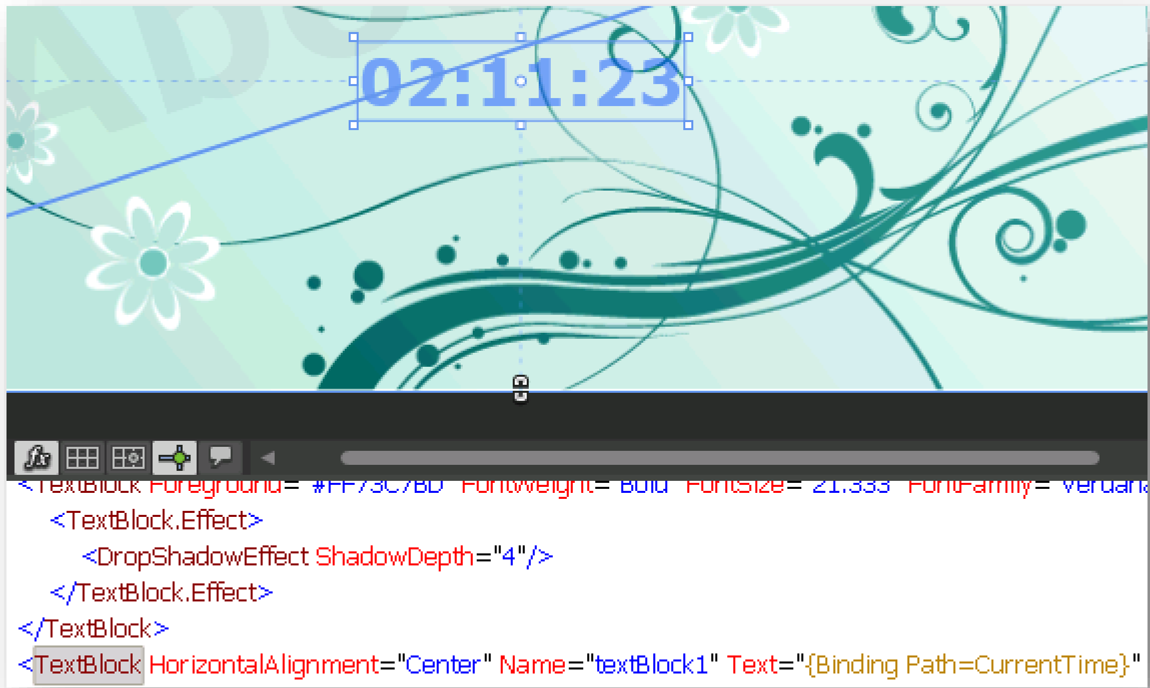


Figure 27 - Binding d'un TextBlock – résultat

La Vue Principale

Il faut bien y venir… J'ai commencé l'exposé par la Vue AboutScreen car il s'agissait de vous montrer les principes de M-V-VM dans le contexte le plus simple possible.

Mais qu'il s'agisse de la vue about box ou bien de la vue diaporama que nous verrons plus loin, il est clair qu'il faut régler le problème posé par la vue principale.

Plusieurs options s'offre à nous. La première et la plus radicale serait de dire qu'il n'y a pas de vue principale !

En tout cas celle-ci pourrait se résumer à un UserControl vide possédant juste un LayoutRoot de type Border par exemple. Chaque vue afficherait un ou plusieurs boutons de navigation et changer de vue consisterait seulement à modifier le contenu du Border de l'écran principal.

Ce n'est pas une façon de pratiquer très conforme aux bonnes pratiques… Cela obligerait chaque vue à gérer un menu ou des boutons de navigation, ce qui n'est pas une bonne idée, et cela voudrait dire que l'écran principal serait traité d'une façon différente des autres vues. Il n'y a en fait aucune raison de ne pas traiter la vue principale comme toutes les autres vues.

Et c'est ce que nous allons faire.

> *Nota : les principes de navigation introduits dans Silverlight 3 ne sont en rien incompatibles avec M-V-VM, si je ne les utilise pas ici c'est par choix pour ne pas éparpiller de trop l'article.*

Rappelons l'aspect de la vue principale :

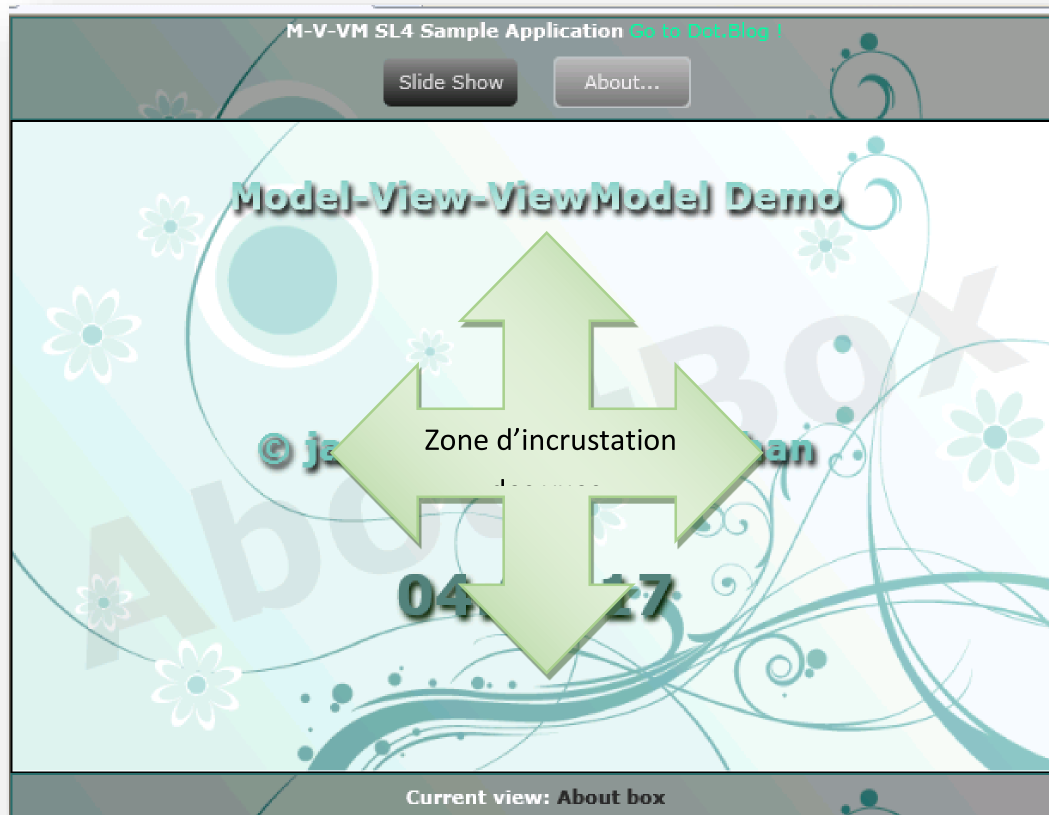


Figure 28 - La Vue Principale

Cette vue se compose de trois zones horizontales :

- Un bandeau contenant les boutons de navigation ainsi qu'un titre et un lien hypertexte vers mon blog
- Une zone centrale qui reçoit les Vues (l'about box sur la figure 28)
- Un bandeau inférieur qui contient le nom de la vue en cours

Cette vue, et son ViewModel, fonctionne exactement comme celle que nous venons d'étudier. Toutefois elle soulève quelques problèmes nouveaux.

Par exemple elle doit rester visible et doit offrir une zone d'accueil pour les autres vues. Elle doit aussi afficher le nom de la vue en cours. De plus, elle doit contenir des boutons et même un lien hypertexte. Ces derniers éléments ne sont pas juste des zones de visualisation

comme des TextBlock. Il y a un cycle action / réaction qui impose le traitement de certains événements. Comment cela peut-il se résoudre avec le principe des Modèles de Vue qui exposent de simples propriétés ?

C'est toute la question du **Commanding** !

Le Commanding

La gestion des commandes dans un logiciel est un problème crucial que les informaticiens ont pris au sérieux depuis très longtemps.

Déjà sous Win32 il existait des solutions pour permettre à un logiciel de regrouper les commandes sous formes d'objets réutilisables ne serait-ce que pour maintenir une parfaite synchronisation entre les boutons d'une Toolbar et les entrées de menu correspondantes.

Déjà à cette époque certains se souciaient de l'expérience utilisateur et proposaient même un moyen de personnaliser menus et Toolbars.

Toutes ces avancées ergonomiques justifiaient la présence d'un système centralisé de gestion des commandes.

Sous des environnements comme les MFC de C++ ou la VCL de Delphi, la librairie de base étant assez figée et beaucoup de choses furent facilement réglées soit en ajoutant des propriétés et des méthodes à certaines branches de composants (boutons, menus, items de menu) ou bien en fournissant des librairies totalement nouvelles (composants tiers) ayant leur propre logique.

Sous WPF et Silverlight la souplesse et la versatilité de Xaml peuvent s'avérer parfois un handicap pour résoudre des problèmes aussi simple. Car comment imposer un jeu de composants pour les Toolbar ou les menus alors même qu'en Xaml tout objet vectoriel peut devenir un bouton, alors que tout conteneur peut servir de Toolbar ?

Un environnement limité offre un avantage : il est facilement « cernable » et proposer des « verrues » qui se détachent totalement de la base est parfaitement accepté puisque sinon on ne pourrait (presque) rien faire.

Mais sous WPF et Silverlight il est très difficile d'accepter un tel raisonnement. Il faut absolument trouver un moyen générique et conforme à l'esprit de ces environnements pour régler le problème de la gestion des commandes. Et il est souvent plus facile de créer un nouveau compilateur ou un langage tel que Xaml que d'inventer toutes les méthodes et les bonnes pratiques pour le maîtriser !

Sous WPF, dont la taille du Framework ne compte pas réellement, Microsoft a pu implémenter une gestion de commande qui se fonde sur l'interface ICommand et des classes de service (CommandManager par exemple).

Sous Silverlight utilisant un Framework réduit il a fallu attendre la version 4 pour qu'une partie (une partie seulement hélas) du système de commande de WPF soit ajoutée.

On y retrouve l'interface ICommand, mais pas toute la logique présente dans WPF qui automatise notamment la gestion des états des commandes (actives ou inactives principalement) ce qui permet de griser un bouton lorsque la commande correspondante n'est pas disponible.

ICommand était déjà présente dans Silverlight 3 mais il n'existait vraiment rien d'autre. Sous Silverlight 4 une partie de la gestion de commande WPF est présente et c'est elle que nous allons exploiter maintenant. Un exemple valant bien mieux que de trop longs discours…

L'interface ICommand

Cette interface propose deux méthodes et un événement :

- CanExecute() qui doit retourner true ou false selon que la commande est disponible ou non
- Execute() qui exécute réellement la commande
- CanExecuteChanged, un événement que l'objet commande peut lever si l'état de CanExecute() change, ce qui, en cascade permet à l'IHM de se mettre à jour.

Dans l'idéal, si Silverlight était WPF, ce que je viens d'écrire concernant l'événement CanExecuteChanged serait vrai. Or, comme le CommandManager et les autres classes n'existent pas, ce retour d'information de la commande vers l'IHM ne se fait pas tout seul.

Nous verrons que nous serons obligés d'inventer une solution pour palier ce manque, même sous SL 4+.

Le principe de fonctionnement des commandes

Une commande est un objet qui implémente l'interface ICommand, donc qui sait proposer les deux méthodes principales listées ci-dessus.

L'objet commande contient en outre tout le code nécessaire à l'exécution de la commande. Un object commande peut accepter un paramètre modifiant son comportement.

La commande peut être n'importe quelle classe, l'important c'est qu'elle implémente ICommand. De fait une commande peut toujours être vue comme une variable de type ICommand.

C'est là que nous sommes intéressés pour la mise en œuvre de M-V-VM…

En effet, *si une commande peut être vue comme une variable, elle peut donc être une propriété du ViewModel* !

Et qui dit propriété dit data binding !

Oui, mais on ne s'emballe pas… Certes il est possible d'exposer une propriété de type ICommand dans un ViewModel. Et bien sûr il va donc être possible de se binder à cette dernière. Mais binder quoi ? Quel objet va se lier à cette propriété et comment va-t-il savoir comment s'en servir ?

Prenons la classe Button. Pour répondre au Click d'un bouton il faut écrire un gestionnaire d'événement. C'est dans celui-ci qu'on pourra réagir. Comment intervient ICommand dans cette affaire d'événement ? Peut-on binder le « Click » d'un Button à un ICommand ?

Certes non. On peut éventuellement retourner une expression Lambda pour gérer un événement mais pas une instance d'une interface. Les types ne sont pas compatibles.

Sous Silverlight 3 qui ne faisait qu'implémenter ICommand et rien d'autre, le problème restait entier, ce qui impliquait la mise en place de solutiosn externes comme MVVM Light, Caliburn, Cinch, Silverlight.FX, Prism, et toutes les librairies dont je vous ai parlé en introduction.

Avec Silverlight 4 un petit morceau du système de commande de WPF a été ajouté. C'est peu mais c'est énorme… En effet, la classe ButtonBase qui est à l'origine de la branche donnant naissance à Button, mais aussi à l'HyperLink et à d'autres composants dérivés du bouton, cette classe comporte désormais une nouvelle propriété qui est « Command ». Son type ? ICommand bien entendu. Une seconde propriété la complète, « CommandParameter ». Le fameux paramètre qui peut être passé à la commande.

De fait, avec Silverlight 4+ il devient possible de binder la propriété Command d'un Button **sans avoir à programmer de gestionnaire d'événement** pour son Click.

Je pense que vous commencez à comprendre la nuance et l'espace de possibilités qui s'ouvre ainsi…

Dès lors il n'est plus nécessaire de coder les gestionnaires d'événement dans le code-behind de la Vue, les commandes sont des propriétés du ViewModel comme les autres et les boutons et assimilés de la Vue y sont bindés via leur propriété Command.

Fantastique !

Encore une fois il faut savoir mesurer sa joie. C'est un grand pas en avant mais cela ne règle pas tout.

Par exemple le système de commande WPF n'étant pas totalement intégré à Silverlight 4 la prise en compte de l'état de la commande (enabled / disabled) n'est pas automatique. Il faudra donc pallier à ce problème.

Autre limitation, mais que l'on retrouve sous WPF aussi, est que tout ce système de commande fonctionne sur le ButtonBase, et, sous-entendu sur l'événement Click. C'est un bon début mais il existe bien d'autres événements souris ou clavier pour faire une « vraie » application …

Ainsi il n'est pas possible pour l'instant de gérer un drag'n drop dans le ViewModel, il n'existe pas de ICommandDragDrop, pas plus que de ICommandMouseWheel.

Est-ce grave ? Un peu. Un peu seulement. Forcément cela interdit d'implémenter un modèle M-V-VM totalement « pur ». Mais un peu seulement car finalement M-V-VM n'est pas une pattern intégriste ☺ elle prévoit des entorses et accepte que certaines choses soient prises en charge dans le code-behind d'une Vue (j'en ai parlé en tout début d'article).

Par exemple la gestion d'un drag'n drop sera typiquement pris en charge dans le code-behind. Mais attention : uniquement la partie « IHM » du drag'n drop ! A un moment donné cette opération va déclencher quelque chose, une action. Et là il faudra aller chercher la ICommand correspondante dans le ViewModel.

Cela ne fait pas uniquement que sauver les apparences, c'est l'intégrité même de M-V-VM qui est sauvée et cela prouve qu'en acceptant que certaines opérations de gestion de l'IHM soit placées dans le code-behind de l'IHM (ce qui est logique et cohérent), la pattern peut être totalement implémentée, avec tous les avantages qu'on est en droit d'en attendre.

Il s'agit bien d'interpréter et d'adapter M-V-VM à la réalité des outils et non pas de sacrifier la pattern ou de faire un odieux bricolage.

La gestion de la surface d'accueil des vues

Notre Vue principale possède une surface centrale qui va accueillir les différentes vues de l'application.

Deux problèmes ici :

- La gestion purement visuelle des transitions
- La gestion technique via le ViewModel

Pour la partie graphique j'ai choisi de créer un UserControl appelé « Frame ». Il s'agit d'un composant très simple puisqu'il possède un grille LayoutRoot qui elle-même contient un

ContentPresenter. L'avantage de ce dernier est d'offrir une propriété Content qui peut être data bindée. Attention, ce contrôle ne doit pas être confondu avec le composant Frame de Silverlight qui est utilisé pour la navigation.

Ne reste plus qu'au code de Frame d'exposer ce contenu via une propriété de dépendance (voir mon article à ce sujet[14]) appelée FrameContent de type UIElement.

Si vous regardez la démo vous remarquerez que le passage d'une page à l'autre se fait avec une transition de type fading. Cela pour rappeler que l'esthétique est très important dans une application Silverlight, même s'il s'agit d'une démonstration technique. Regardez le code pour comprendre comment l'effet est rendu (création de storyboards dynamiques).

Le problème visuel est donc réglé, le ViewModel de la fiche principale expose une propriété qui pourra être bindée avec une instance d'une vue, ce qui affichera cette dernière dans la zone prévue.

Le Modèle de Vue de la fiche principale

Une fois les questions du commanding et de l'affichage des vues imbriquées réglées, il ne reste plus qu'à écrire le code du ViewModel de la fiche principale.

Les commandes – méthode de base

Néanmoins, si nous avons parlé des commandes et de leurs principes nous n'avons pas vu comment les implémenter. Alors prenons un exemple, celui de l'affichage de la About box :

```
/// <summary>
/// Show About box command
/// This is showing how to implement the command without the help
/// of RelayCommand class
/// </summary>
public class ShowAboutBoxImpl : ICommand
{
  private MainPageViewModel main;

  private ShowAboutBoxImpl() { }
  public ShowAboutBoxImpl(MainPageViewModel mainPage)
  { main = mainPage; }

  public bool CanExecute(object parameter)
  {
    return !(main.CurrentPage is Views.AboutScreen);
  }
```

[14] « Les propriétés de dépendance et les propriétés jointes sous WPF » accessible ici : http://www.e-naxos.com/Blog/post.aspx?id=ee3a2372-acd9-4650-a1d9-76ce4f21e483

```
    public event EventHandler CanExecuteChanged;

    public void Execute(object parameter)
    {
      main.CurrentPage = new Views.AboutScreen();
    }
  }
```

Sans surprise, la classe ShowAboutBoxImpl implémente l'interface ICommand. Le constructeur de base est caché car la commande a besoin de connaître la fiche principale pour mettre à jour ses propriétés. Il existe donc un autre constructeur acceptant l'instance en paramètre.

Il s'agit d'une entorse à la séparation des blocs, et en toute logique la commande devrait utiliser le ServiceLocator pour récupérer l'instance de la fiche principale. Mais l'implémentation que je vous propose ici en exemple n'est pas celle qui va être utilisée en réalité. C'est pourquoi elle n'est pas « peaufinée ». Son intérêt réside dans la démonstration que tout peut être réalisé avec les outils de Silverlight 4, sans librairie externe.

Si cette classe était utilisée, on en créerait une instance dans le constructeur du ViewModel, instance qui serait exposée sous la forme d'une propriété de type ICommand.

Mais on peut faire plus efficace.

Prism propose la classe DelegateCommand, mais je ne voulais pas utiliser Prism dans cet article où tout, ou presque, est réalisé uniquement avec les éléments de base de Silverlight pour que vous compreniez le principe et que vous soyez libre de choisir ensuite la librairie qui vous convient le mieux[15].

Repiquer DelegateCommand dans Prism sans utiliser le reste n'est pas aisé, Prism est un bloc proposant beaucoup d'autres choses essentielles. Un tel découpage aurait plutôt été du genre massacre à la tronçonneuse.

En revanche MVVM Light Toolkit dont j'ai déjà parlé ici plusieurs fois, propose une classe assez proche, RelayCommand. Et comme ce toolkit est « light » il est bien plus facile d'en extraire un petit bout… La classe a d'ailleurs été développée par Josh Smith et non par Laurent Bugnion, auteur du toolkit. Comme je vous ai donné les adresses de tous ces gens, je

[15] Soyons clairs, j'adore Prism ! Mais faire un article sur M-V-VM est déjà bien long, s'il avait fallu présenter Prism en même temps cela aurait été impossible… Déjà près de 60 pages à ce niveau, ce n'est plus un article c'est déjà un livre !

ne peux que vous suggérer d'aller visiter leurs sites pour voir toutes les autres bonnes idées qu'ils proposent.

Pourquoi utiliser RelayCommand au lieu de développer chaque commande comme l'exemple montré un peu plus haut ?

Simplement parce que créer une classe pour chaque commande peut s'avérer un peu lourd. Et je veux vous faire voir que M-V-VM est en réalité simple et léger à mettre en œuvre. Grâce à la classe RelayCommand nous allons éviter de créer une nouvelle classe pour chaque commande. En effet, RelayCommand, qui implémente ICommand, possède des constructeurs permettant d'enregistrer directement des expressions lambda pour les deux méthodes de ICommand.

De ce fait, créer une nouvelle commande se résume à créer une nouvelle instance de RelayCommand en passant en paramètre une, voire deux expressions Lambda. Facile, rapide et bien plus court. Donc moins sujet aux bogues.

Je vous invite à aller sur le site de Laurent Bugnion (http://blog.galasoft.ch/Default.aspx) pour lire les explications complètes sur la classe RelayCommand, cela évitera les redites.

Le code du modèle

```
public class MainPageViewModel : BindableObject
{
   #region private fields
   private UIElement currentPage;

   // commanding
   private RelayCommand showAboutBoxImpl;
   private RelayCommand linkToODWebImpl;
   private RelayCommand showSlideShowImpl;

   private List<RelayCommand> commands = new List<RelayCommand>();
   #endregion

   #region Ctor
   public MainPageViewModel()
   {
      // init commands
      showAboutBoxImpl = new RelayCommand(() =>
         { CurrentPage = new Views.AboutScreen(); CheckCommandState();},
         () => !(this.CurrentPage is Views.AboutScreen));
```

```
        commands.Add(showAboutBoxImpl);

        linkToODWebImpl = new RelayCommand(() => HtmlPage.Window.Navigate(new
Uri("http://www.e-naxos.com/blog"), "_blank"));
        commands.Add(linkToODWebImpl);

        showSlideShowImpl = new RelayCommand(() =>
            { CurrentPage = new Views.SlideShow(); CheckCommandState(); },
            () => !(this.CurrentPage is Views.SlideShow));
        commands.Add(showSlideShowImpl);

        // init first displayed view (about box)
        if (showAboutBoxImpl.CanExecute(null))
                        showAboutBoxImpl.Execute(null);
    }
    #endregion

    #region public properties

    /// <summary>
    /// Current page (view) displayed in the central frame
    /// </summary>
    public UIElement CurrentPage
    {
        get { return currentPage; }
        private set
        {
            currentPage = value;
            RaisePropertyChanged("CurrentPage");
            RaisePropertyChanged("CurrentPageName");
        }
    }

    /// <summary>
    /// Current page name
    /// </summary>
    public string CurrentPageName
    {
        get
        {
            if (currentPage == null) return "(no page loaded)";
            var vi = (IViewInfo)((UserControl)(currentPage)).DataContext;
            if (vi == null) return "(no name!)";
            return vi.Name;
        }
    }
    #endregion
```

```
#region Commanding

/// <summary>
/// Show about box
/// </summary>
public ICommand ShowAboutBox
{
   get { return showAboutBoxImpl; }
}

/// <summary>
/// Link to Olivier's blog
/// </summary>
public ICommand LinkToODWeb
{
   get { return linkToODWebImpl; }
}

/// <summary>
/// Display slide show view
/// </summary>
public ICommand ShowSlideShow
{
   get { return showSlideShowImpl; }
}

// About box command display name
public string ShowAboutBoxCommandName
{ get { return "About..."; } }

// Link to OD's web command display name
public string LinkToODWebCommandName
{ get { return "Go to Dot.Blog !"; } }

// Slide show command display name
public string ShowSlideShowCommandName
{ get { return "Slide Show"; } }
#endregion

#region Commanding - "Enabled" state check
private void CheckCommandState()
{
   foreach (var rc in commands)
   {
      if (rc!=null) rc.RaiseCanExecuteChanged(true);
   }
```

```
    }
    #endregion

    #region Commanding - class sample

}
```

Décortiquons un peu ce code.

Première chose à noter, le Modèle de Vue hérite de BindableObject que nous avons déjà présenté. L'une des raisons est que le ViewModel doit supporter INotifyPropertyChanged ce qui est assuré par cet ascendance.

La première région, les champs privés, déclare une référence qui stockera la page courante (la vue active) ainsi que les trois RelayCommand nécessaires (les deux boutons et le lien hypertexte).

On voit aussi qu'une liste de RelayCommand est créée. Nous allons y revenir car cela est important.

Suit le constructeur qui s'attache à créer les instances de chaque RelayCommand en passant les deux expressions Lambda. La première est celle qui exécute l'action, la seconde, optionnelle, est celle qui renvoie l'état de la commande (enabled ou disabled).

> *On notera que les Vues sont recréées à chaque fois qu'on les demande, le ServiceLocator se chargeant de fournir toujours le même ViewModel cela donne l'impression d'une continuité quand on revient sur une Vue. Ces comportements sont des choix arbitraires pour la démonstration.*

Chaque commande créée est ajoutée à la liste commands.

Enfin, le constructeur initialise le premier affichage, cela est purement cosmétique, et il utilise pour cela la commande qui vient d'être créée pour l'affichage de la about box. On remarque l'utilisation qui est faite de ICommand : on commence toujours par tester si la commande est disponible avant d'appeler son exécution.

On trouve ensuite la section des propriétés.

On note l'utilisation de RaisePropertyChanged de la classe ancêtre pour notifier les changements de propriété. On note aussi que certaines propriétés comme CurrentPage lancent plusieurs notifications pour forcer le rafraichissement d'une propriété qui est liée, le nom de la vue en cours.

Pour obtenir le nom de la vue en cours on récupère l'objet DataContext de la vue courante et on cherche à savoir si celui-ci supporte l'interface IViewInfo (présentée plus haut dans cet article). Cette interface que les ViewModel supportent tous retourne une chaîne contenant le nom de la vue à afficher.

On notera la présence de propriété texte de type xxxCommandName. Ces propriétés retournent le texte à afficher dans la commande, ici les boutons ou l'hypertexte. L'intérêt dans la démonstration est limité mais dans la réalité il est souvent essentiel de prévoir un mécanisme de ce type pour simplifier la localisation des IHM. Le plus souvent les propriétés iront puiser le texte à afficher dans un fichier de ressource propre à la culture de l'utilisateur. Ainsi le texte même des boutons ou des liens hypertexte sont liés par data binding au ViewModel, centralisant tous les textes localisables de l'application.

La région Commanding expose les commandes RelayCommand sous la forme de ICommand, type reconnu par ButtonBase et ses descendants.

Enfin, nous trouvons une méthode CheckCommandState().

Cette méthode est appelée à chaque fois qu'une commande est exécutée, après celle-ci (on le voit dans les expressions Lambda des RelayCommand).

Son but est de simuler une partie inexistante de la gestion de commandes de WPF : la mise à jour des états des commandes.

C'est donc ici que la liste des commandes est utilisée. La méthode balaye toutes les commandes sans distinction et demande à chacune d'elle de lever l'événement CanExecuteChanged (de ICommand). Lorsqu'une commande déclenche cet événement, le ou les objets qui sont liés sont avertis que l'état de la commande (sa disponibilité) vient de changer, ce qui force la mise à jour de l'affichage de cet état. Par exemple un Button deviendra grisé si la commande n'est plus disponible.

Nous utilisons ce principe dans la démonstration pour que les deux boutons activant les vues se désactivent automatiquement lorsque la vue correspondante est déjà la vue active…

La vue principale – résumé

Un peu plus complexe que la vue About box, la vue principale a été l'occasion de résoudre de nouveaux problèmes tels que la gestion des commandes, le rafraichissement des états des commandes, l'incrustation d'une vue dans une autre.

Progressivement nous avons mis en place tous les éléments qui forment la pattern M-V-VM au sein d'une application qui fonctionne et qui ne reste pas qu'une approche théorique.

Nous avons utilisé ce que Silverlight 4 met à notre disposition sans faire appel à des librairies plus ou moins lourdes ou sophistiquées, même si, pour simplifier certains aspects de la programmation nous avons créés ou utilisés ou des classes utilitaires (la Frame, le RelayCommand, le BindableObject principalement). Ces classes offrent des comportements intéressants mais nous avons aussi montré qu'elles n'étaient pas indispensables.

Les ViewModel peuvent fort bien implémenter INotifyPropertyChanged au lieu d'hériter de BindableObject. Les commandes peuvent être créées sous forme de classes au lieu d'utiliser RelayCommand. Le contrôle Frame est un artifice qui permet uniquement un effet visuel et n'a pas de lien avec la pattern M-V-VM.

Tous les problèmes d'implémentation du pattern ne sont pas pour autant balayés. Nul doute que les librairies que nous avons présentées (Prism, Caliburn…) couvrent des aspects qui n'ont pas forcément été abordés ici. Armez du présent article, à vous de juger de leur pertinence !

Mais avec ce qui a été dit jusqu'à maintenant une application comme celle de démonstration peut entièrement être mise en œuvre.

Reste un dernier point qui n'a pas été traité : la gestion des dialogues de confirmation. Une application réelle a besoin à un moment ou un autre d'afficher des dialogues de ce type à l'utilisateur. Puisque c'est le ViewModel qui exécute le code et qu'il ignore la ou les vue qui lui sont connectées, comment peut-il intervenir au niveau de l'IHM pour afficher un dialogue ?

C'est une question essentielle que nous aborderons dans le dernier volet de cet article, la vue diaporama…

La Vue Diaporama

Pour se rafraichir la mémoire, regardons à nouveau la vue Diaporama :

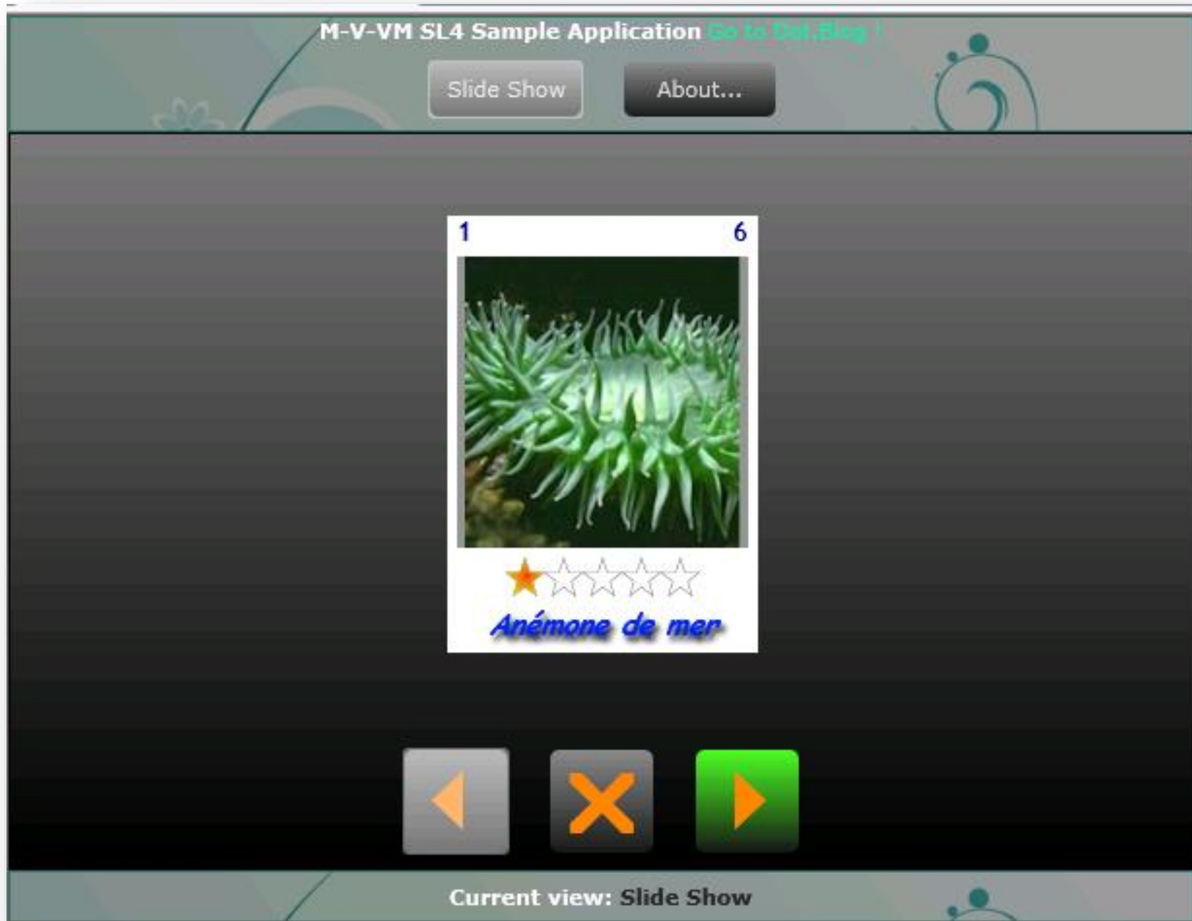


Figure 29 - La Vue Diaporama (rappel)

Fonctionnement

La vue propose un espace central affichant une photo provenant d'un diaporama complet décrit sous la forme d'un fichier XML.

L'image affichée est entourée de plusieurs zones. En haut se trouve le numéro de l'image dans la séquence et à droite le nombre d'images dans le diaporama.

Sous l'image se trouve un composant Rating afficher une valeur comprise entre 0 et 5. Sous ce composant est affichée la description de la photo visionnée.

En bas de l'écran trois boutons servent à gérer le défilement en avant ou en arrière du diaporama, le bouton central permettant de remettre à zéro le rating de la photo en cours.

Lorsque le bouton central est cliqué un dialogue de confirmation est affiché (figure suivante).

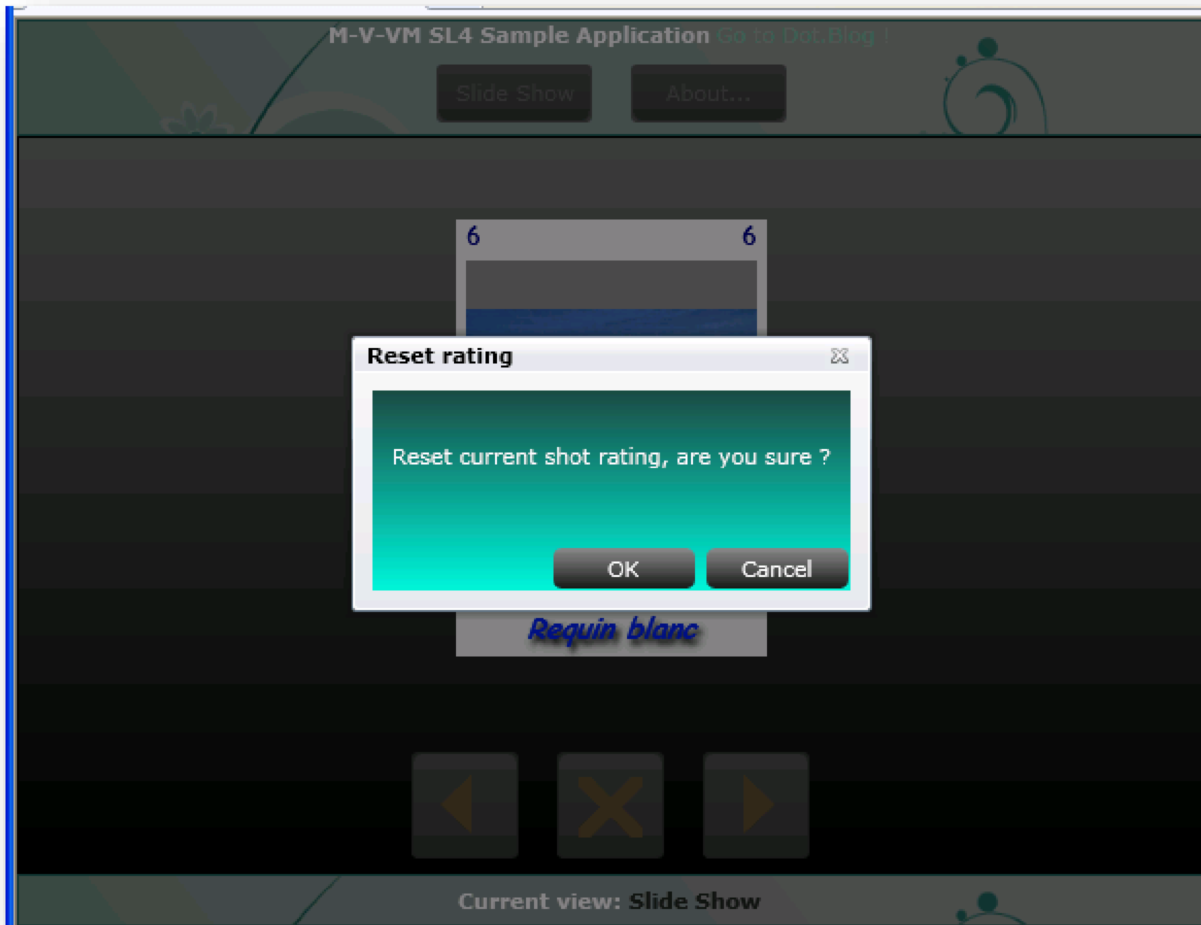


Figure 30 - Dialogue de confirmation

Difficulté particulière

Cette vue n'offre pas de grandes difficultés, tous ses besoins sont couverts par les mécanismes M-V-VM que nous avons vus jusqu'ici.

> *On notera que dans cet exemple le rôle d'adaptation des données n'a pas été confié au ViewModel mais à la classe de donnée elle-même. L'exemple est celui du Rating. L'application accepte des valeurs entre 0 et 5 alors que le composant Rating utilise une plage de 0 à 1.0. Comme le ViewModel passe directement une collection de Shot à la Vue il lui aurait fallu cacher cette liste et en fournir une autre dont les membres ne seraient plus des Shot mais des adaptations. Cela compliquerait bien trop la démonstration. Ici c'est la classe Shot elle-même qui expose deux propriétés, l'une au format 0-5 et l'autre au format double de 0 à 1.0. C'est à cette propriété qu'est bindé le contrôle Rating. Les validations sont d'ailleurs aussi prises en compte par la classe Shot ce que vous pouvez voir dans le code de la classe qui utilise le namespace DataAnnotation et les attributs qu'il déclare.*

Reste un point essentiel : le dialogue de confirmation.

En effet, lorsque le bouton de remise à zéro du rating de la photo courante est cliqué, cela déclenche le fonctionnement de la commande qui se trouve dans le ViewModel. Comment ce dernier peut-il agir à son tour sur l'interface visuelle ?

Résoudre le problème du dialogue

Plusieurs approches sont certainement envisageables, aucune n'est parfaite. C'est certainement ici qu'on atteint les limites de ma démonstration faite uniquement avec les outils de base et que se justifie l'utilisation de certaines librairies spécialisées pour mettre en œuvre M-V-VM.

Toutefois je ne suis pas du genre à baisser les bras aussi facilement ! Et nous allons voir que sans aucune librairie tierce nous pouvons trouver des solutions.

Une première approche

La première chose à noter est que le modèle M-V-VM nous autorise à utiliser le code-behind si ce code concerne l'IHM. L'affichage d'un dialogue entre dans cette catégorie.

La seconde chose concerne la connaissance que la Vue à de son Modèle de Vue. Autant l'inverse n'est pas vrai (le ViewModel ne sait rien de la Vue qui se connecte à lui) et ne doit en aucun cas être « cassé », autant la Vue, au moins au travers de son DataContext et de tous les bindings qu'elle comporte est obligée de connaître le ViewModel. Pas la classe exacte, sinon l'intérêt de mettre en place un ServiceLocator entre Vues et Modèles de Vue tomberait à l'eau, mais au moins son contenu, sa structure.

Peut-on, en utilisant ces deux informations, trouver une solution élégante au problème du dialogue ?

Dans le cas de notre démonstration la réponse est « presque » oui. En effet il s'agit uniquement de confirmer une action initiée directement dans l'IHM. Il suffit donc à cette dernière d'intercaler la confirmation entre l'action originale et la commande.

Pour créer le dialogue j'utilise un ChildWindow (voir ma video « Les ChildWindow's de Silverlight 3 » http://www.e-naxos.com/Blog/post.aspx?id=b9ccd31a-5e5b-4f5a-a0f9-1134d6566d4f).

Voyons comment l'exécution de ce dialogue est intercalée entre le clic du bouton et la commande de remise à zéro du rating :

```
private void Button_Click(object sender, RoutedEventArgs e)
{
   var cd = new Views.ConfirmReset();
   cd.Closed += new EventHandler(cd_Closed);
```

```
        cd.Show();

    }

    void cd_Closed(object sender, EventArgs e)
    {
        var cd = (Views.ConfirmReset)sender;
        if (cd.DialogResult == null) return;
        if ((bool)cd.DialogResult)
        {
            if
    (((SlideShowViewModel)DataContext).ResetRating.CanExecute(null))
            ((SlideShowViewModel)DataContext).ResetRating.Execute(null);
        }
    }
```

Bien entendu nous sommes ici obligés de programmer un gestionnaire d'événement sur le clic du bouton (première partie du code) dans lequel la fenêtre de confirmation est créée puis affichée.

Dans le code gérant la fermeture de la fenêtre (seconde méthode) nous commençons par récupérer la valeur de sortie de cette dernière (un booléen nullable).

Si la réponse de l'utilisateur est positive nous récupérons l'instance du ViewModel se cachant derrière le DataContext de la vue en cours et nous le transtypons. Ne reste plus qu'à utiliser la commande « ResetRating » du ViewModel pour réaliser l'action.

On peut se demander si le transtypage ne créé pas ici un couplage fort entre la Vue et son ViewModel. La réponse est oui, bien entendu. D'un autre coté comme nous l'avons dit il y aurait une sorte d'hypocrisie à croire que la Vue est totalement indépendante du ViewModel : elle y est bindée par son DataContext et tout ce qu'elle affiche, toutes ses commandes sont bindées « nominativement » à des commandes ou des propriétés particulières de ce ViewModel là et pas un autre.

D'un autre côté l'argumentation semble oublier que sans cette référence « en dur » la vue pourrait, via le ServiceLocator, être reliée à une instance totalement différente du ViewModel et que tant que ce dernier offrirait au minimum les mêmes propriétés tout fonctionnerait. Et que cela est même l'un des avantages et la raison d'être du ServiceLocator… Nous ne pouvons pas en rester là, sur un tel paradoxe remettant en cause nos choix méthodologiques, c'est une évidence.

Si nous n’avions pas fait le choix du ServiceLocator, ce qui serait parfaitement légitime dans le cadre stricte de M-V-VM, il n’y aurait au final aucun problème puisque la Vue serait obligée (d’une façon ou d’une autre) de connaître son ViewModel pour l’instancier et s’y lier.

Mais le ServiceLocator est une pièce essentielle dans la construction de l’application, il nous assure une plus grande souplesse, un découplage fort entre Vues et Modèles de Vue et il facilitera les tests.

Peut-on s’en sortir autrement ?

On pourrait utiliser la réflexion pour retrouver la commande dans le DataContext mais cela serait trop lourd (mais efficace et générique). Dans une application réelle nul doute que je choisirai peut-être d’implémenter une classe utilitaire offrant ce service ce qui règlerait la question pour l’ensemble de l’application.

Mais peut-on trouver un moyen plus simple ?

Récupérer la commande dans le DataContext comme le fait Xaml lors du binding serait parfait, mais je me doute que cela passe, forcément, par la réflexion, ce qui revient à la solution précédente.

Il existe toutefois une ruse possible :

Nous allons définir un bouton caché qui aura sa propriété Command bindée correctement à la commande du ViewModel. Cela nous évite d’avoir à connaître la classe du ViewModel, et c’est ce que nous cherchons :

```
<Button Name="HiddenBtnResetRating" Content="" Height="1" Width="1"
Visibility="Collapsed" Command="{Binding Path=ResetRating}" />
```

Réécrivons le code de fermeture du dialogue :

```
void cd_Closed(object sender, EventArgs e)
    {
      var cd = (Views.ConfirmReset)sender;
      if (cd.DialogResult == null) return;
      if ((bool)cd.DialogResult)
      {
        if (HiddenBtnResetRating.Command.CanExecute(null))
          HiddenBtnResetRating.Command.Execute(null);
      }
    }
```

Grâce à cette astuce nous pouvons invoquer la commande via le bouton caché. Pas de code complexe, pas de réflexion.

Il faut en revanche se satisfaire de la présence d'un bouton caché ce qui n'est pas gênant en soi mais n'est pas aussi élégant qu'on pourrait l'espérer.

L'avantage de cette astuce c'est qu'elle fonctionne pour tous les cas où nous devons intercaler un traitement IHM entre une action de l'utilisateur et une commande fournie par un ViewModel.

Est-ce parfait ?

C'est un autre débat. Dans cet article j'ai justement fait le choix de n'utiliser aucune librairie qui pourrait éventuellement couvrir ce type de problème. Du M-V-VM complet, entièrement « à la main » pour vous aider à comprendre sa mise en œuvre mais aussi les problèmes que celle-ci peut poser…

Il n'était pas dans mes intentions de recréer un super Framework M-V-VM en trois lignes de code. MVP je suis, certes, mais cela signifie justement que j'ai une certaine expérience et celle-ci impose l'humilité tout autant que le constat que le père Noël n'existe pas ! ☺

Ce que proposent les librairies tierces

MVVM Light Toolkit propose une solution basée sur un service de messagerie offrant différents services dont l'affichage d'un message avec retour du résultat à l'appelant. C'est une solution intéressante avec des dialogues standard, mais qui peut devenir plus complexe si on désire utiliser un dialogue personnalisé comme nous le faisons dans la démonstration.

Cinch, un autre framework, utilise une stratégie proche du Service Locator qui enregistre des services pouvant être appelés depuis les ViewModels notamment. Il suffit de créer un service « ShowMessage », de l'enregistrer, puis de l'appeler depuis le ViewModel. Là encore cela fonctionne très bien pour les messages utilisant la boîte de dialogue standard mais devient plus complexe pour les dialogues personnalisés.

Silverlight.FX utilise une approche encore différente mais ne se base pas encore sur les possibilités de Silveright 4.

Prism et Prism pour WinRT utilise un système proche de la messagerie et MvvmCross propose lui aussi un mécanisme très proche de la messagerie de Mvvm Light.

Faisons le point

Personnellement, utiliser des dialogues standards ayant un look XP dans une application Silverlight, respect ou non de MVVM, c'est quelque chose dont je ne veux même pas entendre parler. C'est un contresens total. Une négation même de Xaml.

Alors, avec un petit bouton caché pour seul framework, tout en respectant MVVM à 100%, je trouve ma solution pas si mal que ça car elle permet d'invoquer un dialogue personnalisé correspondant au look & feel de l'application.

Faire mieux est possible, les différents Frameworks évoqués proposent des voies (même si elles ne me conviennent pas totalement en l'état), mais je suis certain que vous saurez faire preuve d'imagination et que parmi vous se trouve celui qui saura trouver une idée simple et efficace ! (qu'il n'oublie pas de venir nous en parler sur Dot.Blog, je lui ferai même une place pour publier son billet !).

CONCLUSION

Une application suivant les principes de M-V-VM du début à la fin, sans aucune entorse aux règles de base, agrémentée d'Inversion de Contrôle, d'incrustation de vues, de dialogues modaux, utilisant des données modifiables. Le tout écrit avec les moyens offerts par Silverlight 4+ (en raison de la gestion de commande) et sans aucune librairie tierce. A chaque fois qu'un bout de code étranger a été utilisé j'ai expliqué comment on pouvait parfaitement s'en passer.

Je pense que la mission technique que je m'étais fixée est remplie, même si tout est toujours perfectible (n'oubliez pas que cet article frôle les 70 pages A4 et qu'il vous est proposé gracieusement ☺).

Reste la mission pédagogique… Cet article vous a-t-il paru clair ? Vous a-t-il permet de mieux comprendre M-V-VM ? Vous a-t-il donné l'envie d'en savoir plus et de tester le concept ? Et mille autres questions que j'aimerai vous poser !

M-V-VM appliqué avec M-V-VM Light
[WPF et Silverlight]

Version 1.0 Août 2010

Sommaire

Code Source 153
Préambule 154
Pourquoi M-V-VM Light ? 154
Une documentation non-officielle ? 155
Se procurer M-V-VM Light 156
Visual Studio et Blend 157
Bref rappels sur M-V-VM 158
Origine 158
Les trois tiers de la pattern 159
La librairie MVVM Light 160
Présentation 160
Le code 160
Le Tree Map et les espaces de noms 161
GalaSoft.MvvmLight.Messaging 162
GalaSoft.MvvmLight.Command 163
GalaSoft.MvvmLight.ViewModelBase 163
GalaSoft.MvvmLight.Helpers 163
GalaSoft.MvvmLight.Extras 163
La matrice des dépendances 163
La matrice graphique des dépendances 165
Le diagramme de classes 167
Les outils de base 167
La messagerie 168
RelayCommand 168
ICleanup et ViewModelBase 169
Les messages 169
Les classes de la librairie « Extras » 170
La pattern MVVM et MVVM Light 170
La séparation Code fonctionnel / Interface Utilisateur 171
Aider à la séparation code / visuel 171
Inversion de contrôle 172

La gestion des commandes 174
ICommand 174
Une commande = une propriété bindable 175
Le support limité de ICommand 175
Simplifier la création des commandes avec RelayCommand 175
Echapper à la limitation du support de ICommand 176
Transformer les événements en commandes 176
La communication 177
Le besoin d'un mécanisme de messagerie 178
Messenger, le messager 178
Les notifications 180
Obtenir une réponse à un message 181
Messages avec ticket réponse 181
Message avec action 182
La généricité 182
Le cas … délicat des dialogues 182
Les changements de propriétés 183
Séquence de ShutDown pilotée par message 184
Le Modèle de Vue 185
Faisons le point ! 186
MVVM Light en pratique 187
La création d'un projet MVVM Light 187
Silverlight 187
Silverlight for Windows Phone 188
WPF 189
Expression Blend 190
Le contenu d'un projet MVVM Light 192
MainPage, le début du voyage 193
De la MainPage à App.xaml 195
De App.xaml au Localisateur de Services 196
Fin du premier voyage 200
Le Modèle de Vue 200

Fin du second voyage ... 202
Gérer les changements de valeurs ... 202
Créer les propriétés ... 203
Avertir l'interface du changement ... 206
Assurer le nettoyage ... 207
Le code ... 207
Méthodes anonymes, délégués et gestionnaires d'événement ... 208
Le cas de la date ... 209
Faisons le point ... 209
Gérer la communication ... 210
Les messages de notification de changement de valeur ... 210
Broadcaster les notifications de changement de valeur ... 211
Recevoir le message dans la vue ... 211
Le point sur les notifications de changement de valeur ... 213
Les notifications ... 213
Exemple de notification entre la Vue et le Modèle ... 214
L'application Exemple3 ... 215
Le Modèle ... 216
La Vue ... 220
Le point sur la notification ... 221
Exemple de notification de Dialogue ... 221
L'application Exemple 4 ... 222
Le Modèle ... 223
Le modèle de Vue ... 230
La Vue ... 231
L'application en action ... 233
Le point sur les notifications DialogMessage ... 235
La gestion des commandes ... 236
RelayCommand (version non générique) ... 237
Les autres éléments de ICommand ... 239
RelayCommand avec paramètre générique ... 242
EventToCommand ... 246

Le projet exemple 248

Le point sur les commandes 255

Conclusion 255

TABLE DES FIGURES

Figure 1 - Bing Translator affichant original et traduction 157
Figure 2 - Interactions des tiers dans MVVM 159
Figure 3 - Le Tree map du MVVM Light 162
Figure 4 - La matrice des dépendances de MVVM Light 164
Figure 5 - Matrice graphique des dépendances 166
Figure 6 - Les WeakAction 168
Figure 7 - IMessenger et Messenger 168
Figure 8 – RelayCommand 169
Figure 9 - ViewModelBase et ICleanup 169
Figure 10 - Les Extras (DispatcherHelper et EventToCommand) 170
Figure 11 - Les patterns de l'Inversion de Contrôle 172
Figure 12 - Les différentes classes de messages de MVVM Light 180
Figure 13 - Templates Silverlight 188
Figure 14 - Les templates pour Windows Phone 189
Figure 15 - Les templates pour WPF 190
Figure 16 - Les templates Silverlight pour Blend 191
Figure 17 - Les templates WPF pour Blend 191
Figure 18 - Squelette d'un template de projet MVVM Light 192
Figure 19 - Compilation du template (sous Silverlight) 193
Figure 20 - MainPage sous Visual Studio 194
Figure 21 - Le binding de la zone de titre 194
Figure 22 - Binding sous Blend 204
Figure 23 - binding sous Visual Studio 205
Figure 24 - Application exemple 2 - phase 1 206
Figure 25 - Notification Vue/Modèle 215
Figure 26 - la Vue de l'exemple DialogMessage 231
Figure 27 - Exemple 4 : Etat initial 233
Figure 28 - Exemple 4 : Après la validation d'une commande de 50 unités 234
Figure 29 - Exemple 4 : Tentative d'une commande négative 234
Figure 30 - Exemple 4 : Saisie supérieure à la validité, affichage du message 235
Figure 31 - Exemple 5 - RelayCommand simple 237
Figure 32 - Exemple 5 : le bouton + est "disabled" 241
Figure 33 - RelayCommand avec paramètre générique 244
Figure 34 - Le contrôle automatique des noms de propriétés de ViewModelBase 250
Figure 35 - Exemple 6 - Les propriétés de EventToCommand 251
Figure 36 - Exemple 6 - Le binding de la commande sous Blend 252

Figure 37 - Exemple 6 : Dans les starting-blocks ! ________ *253*
Figure 38 - Exemple 6 : une grande valeur positive ________ *254*
Figure 39 - Exemple 6 : une très grande valeur négative ________ *254*

CODE SOURCE

Cet article est accompagné du code source complet des exemples (ainsi que des images Jpeg de certains schémas difficilement lisible dans le corps de cet article) [Rendez vous sur Dot.Blog pour télécharger le zip, cherchez le nom de l'article dans la zone de recherche en haut de l'écran].

Projets fournis

Exemple1 : MVVM Light, projet de base (page 192)

Exemple2 : Gestion des changements de valeurs (page 202)

Exemple3 : Les notifications simples (page 215)

Exemple4 : Les notifications de dialogue (page 222)

Exemple5 : La gestion des commandes (page 237)

Exemple6 : Transformer les événements en commandes (page 248)

Extensions : Librairie de classe, extension de MVVM Light (page 229)

Exemple4Context : Exemple 4 utilisant les extensions (page 229)

PREAMBULE

Le pattern M-V-VM (Model-View-ViewModel) est une avancée méthodologique essentielle pour la mise en œuvre de logiciels modernes écrits sous WPF et Silverlight.

Le présent article est une présentation du Framework gratuit **M-V-VM Light** de GalaSoft et prend pour acquis la compréhension du pattern M-V-VM lui-même.

J'invite les lecteurs qui ne sont pas au fait de cette dernière à télécharger et consulter mes articles précédents sur la question, notamment l'article de plus de 70 pages paru en janvier dernier (voir mon billet « Article: **M-V-VM avec Silverlight** » http://www.e-naxos.com/Blog/post.aspx?id=c2053091-cd46-4523-aa37-08ba70e37c23). Ce dernier présente les concepts essentiels à la compréhension de ce qui va suivre.

D'autres billets en relations avec le sujet méritent aussi un détour pour se familiariser avec les problématiques et les solutions propres tournant autour du concept du pattern M-V-VM :

- **Silverlight MVVM : Les commandes**
- **MVVM, Unity, Prism, Inversion de contrôle…**
- **Simple MVVM**

Vous trouverez tous ces billets et articles en vous rendant sur **Dot.Blog** (http://www.e-naxos.com/Blog) et en cherchant l'expression « *mvvm* » ou en utilisant la recherche par tag suivante : http://www.e-naxos.com/Blog/?tag=/mvvm

La lecture d'un autre article me semble tout aussi indispensable pour comprendre ce qui va suivre, il s'agit de « **Le Binding Xaml – Maîtriser sa syntaxe et éviter ses pièges (WPF/Silverlight)** » (voir le billet http://www.e-naxos.com/Blog/post.aspx?id=a6a6ca3e-71ab-425c-86a6-ad17d405ca84). Article long aussi (77 pages) mais sa lecture et la compréhension des mécanismes qu'il explique apparaissent un préambule indispensable pour saisir la mise en œuvre de M-V-VM. [Cet article fait partie du tome « Xaml » de ALL DOT.BLOG].

POURQUOI M-V-VM LIGHT ?

M-V-VM Light est un petit Framework destiné à simplifier l'écriture d'applications mettant en œuvre **le pattern Model-View-ViewModel**. Cette librairie est proposée gratuitement, avec code source, par **Laurent Bugnion**, un MVP Suisse indépendant travaillant sous le nom de sa société : GalaSoft.

M-V-VM Light devient populaire car ce Framework ne s'occupe que de la mise en œuvre de M-V-VM. Il est petit, facilement compréhensible et rapide à cerner, à la différence de Frameworks plus ambitieux dont l'excellent Prism de Microsoft (http://compositewpf.codeplex.com/) qui à vouloir régler de nombreux autres problèmes de

développement réclament malgré tout un investissement important en auto-formation avant même de pouvoir s'en servir. Ce temps, qui nous est manqué à tous, est souvent un frein à l'adoption de certaines librairies. M-V-VM Light contourne l'obstacle en ne ciblant qu'un seul problème : mettre en œuvre facilement le pattern M-V-VM et rien d'autre. Il reste donc possible, pour régler d'autres problèmes de développement d'utiliser conjointement d'autres librairies (dont Prism), cela ne pose aucun souci.

Ce mini Framework évolue sans cesse, et suit les évolutions de WPF et de Silverlight, c'est un **code vivant**, point indispensable pour une utilisation en production.

Le seul problème qu'il faut reconnaître à la librairie est le manque de vraie documentation. Sur son site Laurent Bugnion propose quelques articles, des exemples et des liens, mais aucun document permettant d'avoir rapidement une vision d'ensemble de la librairie ni même de véritable documentation au sens habituel du terme.

Cet article, à sa façon, devrait pallier ce manque et vous rentre l'apprentissage de M-V-VM Light plus simple. En tout cas je l'espère !

[A la date de mise en page de ce PDF MVVM Light a évolué et couvre aussi Windows Phone et WinRT, toutefois ses mécanismes ont très peu changé et cet article reste une base pertinente pour prendre en main ce framework]

UNE DOCUMENTATION NON-OFFICIELLE ?

On pourrait voir cet article de cette façon mais ce n'est pas le cas. Ce qui m'intéresse ici n'est pas de ressortir les commentaires du code pour en faire un article mais bien d'expliquer ***comment se servir de chaque possibilité de la librairie et de situer les apports de MVVM Light vis-à-vis du pattern MVVM et de ses mécanismes propres.***

Pour cela il faut nécessairement planter le décor, c'est cette première partie qui peut être vue comme une documentation de la librairie. Mais le volet le plus intéressant se situe dans les explications permettant de comprendre pourquoi la librairie fait ce qu'elle fait et comment en tirer parti.

C'est pour cela que vous noterez que le titre de cet article est « M-V-VM *appliqué* avec M-V-VM Light » et non pas « Documentation de M-V-VM Light » … **Mon propos est une approche pratique, l'application du pattern M-V-VM, en se servant de l'une des librairies existantes aidant à cette mise en pratique.**

Dans un tel cadre, le choix de MVVM Light tombe sous le sens, pour toutes les raisons invoquées plus haut : un code simple, une surface fonctionnelle restreinte, un ciblage du pattern MVVM efficace, et un code de bonne qualité.

SE PROCURER M-V-VM LIGHT

Vous trouverez toutes les informations sur le site de Laurent Bugnion :

http://www.galasoft.ch/mvvm/getstarted/

Les premières versions étaient dotées d'un installeur mais le temps étant compté pour tout le monde Laurent n'a pas mis à jour ce dernier. Les dernières versions sont ainsi livrées sous forme de fichiers Zip qu'il faut décompresser dans certains répertoires de la machine. Il existe aussi des packages Nuget installables depuis Visual Studio, je vous conseille désormais ce type d'installation qui reste « accroché » à la Solution VS et qui groupe la librairie avec le code, ce qui évite les problèmes lors de la reprise d'un code ancien, code et librairie sont en phase.

Les explications sont assez claires, il suffit de les suivre pas à pas, cette partie est suffisamment documentée pour ne pas nécessiter de redites.

Pour ceux qui sont vraiment allergiques à l'anglais, n'oubliez pas que Microsoft fournit **Bing Translator** qui permet de naviguer sur un site traduit automatiquement (et avec une qualité de traduction assez correcte) : http://www.microsofttranslator.com/

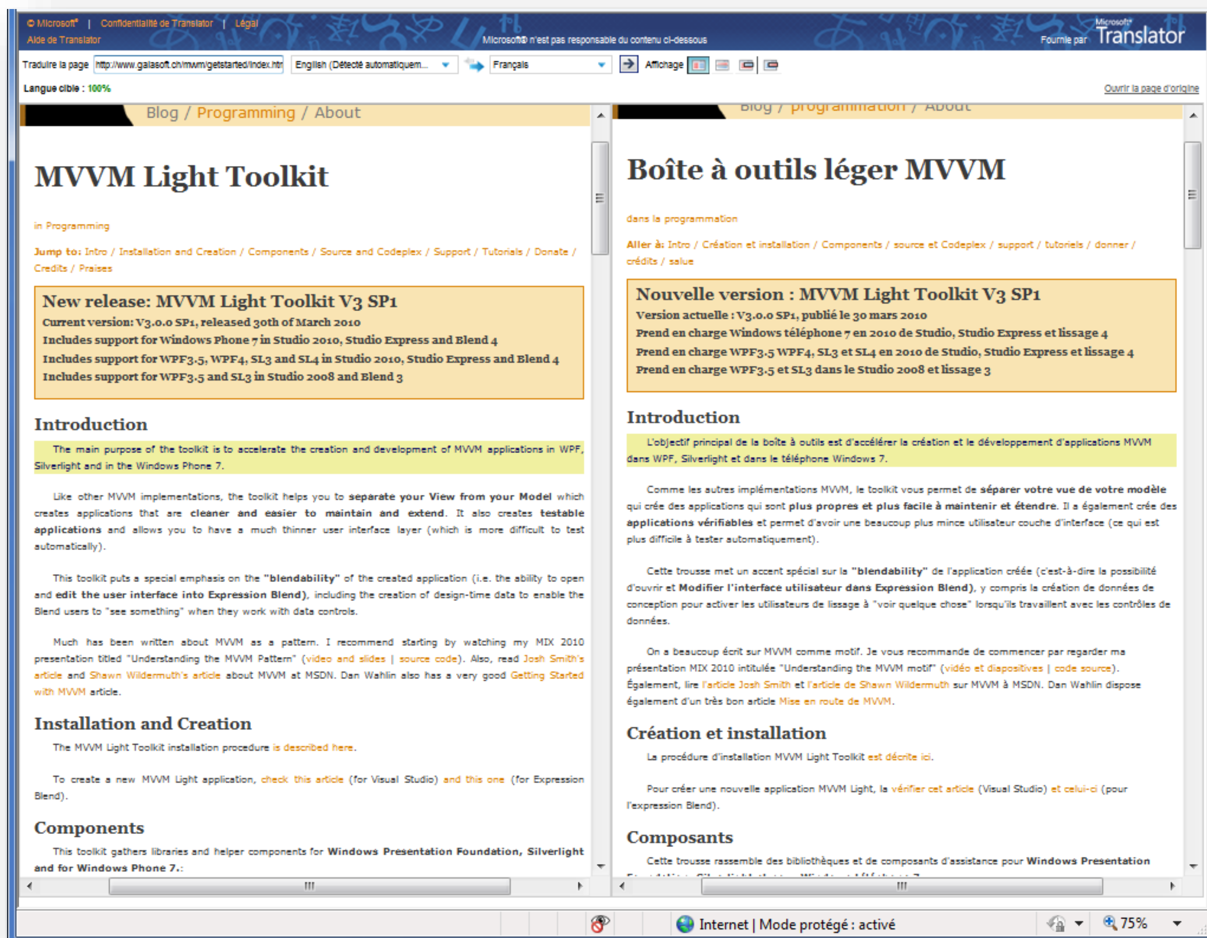


Figure 31 - Bing Translator affichant original et traduction

VISUAL STUDIO ET BLEND

M-V-VM Light permet de séparer la conception du visuel et du code, c'est le principe même du pattern M-V-VM. Mais lorsqu'on parle de conception visuelle, de Design, on est obligé de prendre en considération **Expression Blend**[16], le seul vrai outil conçu pour travailler sur les interfaces WPF et Silverlight. Les éditeurs intégrés de Visual Studio ne pouvant prétendre, au mieux, qu'au rang d'aides bien pratiques pour faire des mises en place rapides (tests, maquettes, utilitaires, etc).

Ne croyez pas ceux qui prétendent qu'on peut tout faire avec Visual Studio, et encore moins ceux qui vous disent qu'en tapant du Xaml à la main on arrive à tout. Ce sont les mêmes qui disaient qu'un site Web ne nécessitait rien d'autre que le bloc-notes comme outil de conception il y a 10 ou 15 ans… On voit bien avec le recul quels sont les sites Web qui ne

[16] Voir le site http://www.microsoft.com/expression/products/blend_overview.aspx

valent rien et ceux qui « ont de la gueule ». Ceux conçus avec le bloc-notes n'en font pas partie il faut se l'avouer… Idem pour Xaml sans Blend…

De fait, les librairies conçues pour WPF et Silverlight se doivent non seulement d'être utilisables sous Visual Studio mais elles doivent offrir quelque chose de nouveau qu'on appelle « la **blendabilité** » (*blendability*) c'est-à-dire la possibilité d'avoir un retour visuel s'intégrant dans Expression Blend, de disposer de behaviors simplifiant la création des interfaces pour un Designer, etc.

M-V-VM Light a été conçu avec la « blendabilité » en tête. On peut donc éditer visuellement les interfaces utilisant M-V-VM (les Vues donc) et on peut bénéficier de l'affichage des données en conception pour simplifier la mise en page grâce à certains choix que nous étudierons plus loin.

Cet aspect de la librairie n'est pas accessoire, Laurent Bugnion, comme tous ceux qui pratiquent Xaml sérieusement, apporte une grande importance à la « blendabilité ».

BREF RAPPELS SUR M-V-VM

Comme indiqué en introduction, j'invite le lecteur à prendre connaissance de mon long article sur M-V-VM ne pouvant ici faire la redite de 70 pages sans rendre le présent article impossible à digérer !

Origine

Toutefois, pour la forme, rappelons quelques points essentiels du pattern M-V-VM.

Model-View-ViewModel est un pattern créé par John Gossman et publiée le 8 octobre 2005[17]. John est l'un des architectes de WPF et Silverlight. Il s'agit donc d'un pattern spécialement pensé pour exploiter les possibilités de Xaml et ce, par un spécialiste de Xaml.

[17] Sur son blog « Tales from the Smart Client », dans l'article "Introduction to Model/View/ViewModel pattern for building WPF apps" publié le 8/10/2005 à 18:51 pour être précis ! L'adresse de cette archive étant : http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx

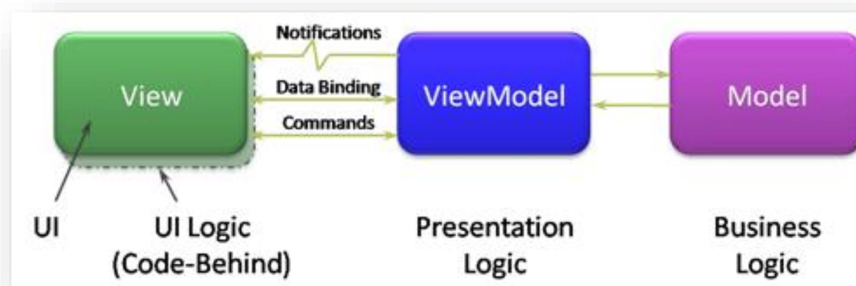


Figure 32 - Interactions des tiers dans MVVM

La figure ci-dessus rappelle la séparation des trois tiers définis par MVVM et les interactions entre ceux-ci.

Les trois tiers du pattern

Le pattern définit trois tiers qui sont :

- La **Vue** (View) définit le visuel, elle est constituée principalement d'un fichier Xaml et se présente sous la forme d'un UserControl ou d'un équivalent (Page, …) dont le fichier de code-behind est vide (ou presque). Le DataContext de la vue est initialisé pour pointer une instance d'un Modèle de Vue. La Vue définit aussi les Data Templates dans le sens où il s'agit d'éléments de présentation des données (visuel). En général la Vue est hébergée au sein d'une application hôte qui propose le Styling global (définition des dictionnaires de styles). Dans la pratique tout ce qui concerne les styles et les templates est stocké dans un ou plusieurs dictionnaires de ressources placés dans un répertoire à part du projet. Dans le pattern MVVM originale la Vue peut se connecter directement aux Modèles sans passer par un Modèle de Vue (cas très simples et rarement vus ni conseillés en pratique).

- Le **Modèle de Vue** (ViewModel) est une abstraction de la vue, il contient tout le code nécessaire au fonctionnement de l'interface : stockage des états, formatage des données, transcodage, persistance des données, mais aussi toutes les commandes de l'interface (ex : réponses aux clics). Le Modèle de Vue est totalement indépendant de l'interface visuelle mais il peut accéder aux Modèles et aux autres couches du logiciel (DAL, BOL …).

- Le **Modèle** représente les données de l'application. Dans la version la plus simple le Modèle peut être un fichier XML. Dans la réalité le Modèle contiendra des requêtes à des services de données (au sens large). De fait, le Modèle utilise généralement des couches intermédiaires comme un BOL et un DAL, mais il peut aussi bien dialoguer

directement avec un SGBD. Ce sont d'autres design patterns et d'autres bonnes pratiques qui imposeront la présence d'éventuelles couches supplémentaires.

Je m'arrêterai ici pour ce bref rappel.

LA LIBRAIRIE MVVM LIGHT

Présentation

Physiquement la librairie MVVM Light se présente actuellement sous la forme de deux assemblages,

- GalaSoft.MvvmLight.dll
- GalaSoft.MvvmLight.Extras.dll

La seconde contient quelques ajouts récents d'une grande utilité, la première contient l'essentiel de la librairie.

Ces assemblages sont disponibles en versions compilées pour les deux technologies dérivant de Xaml : WPF et Silverlight. Les binaires sont généralement proposés pour les versions les plus courantes (actuellement Silverlight 3 et 4 par exemple). Le code source de l'ensemble est disponible aussi.

MVVM Light est complété de *templates* permettant de créer sous Visual Studio et Expression Blend de nouveaux projets intégrant de base l'ensemble des références nécessaires (ainsi que le squelette de quelques unités indispensables).

Tout cela est du ressort de l'installation plus que de l'utilisation de MVVM Light, et vous trouverez toutes les informations nécessaires pour réaliser un setup correct sur le site de GalaSoft.

Tout repose donc sur la dll « GalaSoft.MvvmLight.dll », un exécutable pesant environ 25 Ko, ce n'est donc pas lui qui rendra le téléchargement de votre application plus lent !

Le code

Je ne vais pas ici commenter le code de chaque classe, vous savez lire un commentaire XML et les sources sont publiques… En revanche je vais essayer de vous donner une vision d'ensemble de ce code, information qui n'apparaît pour l'instant nulle part.

> *Pour analyser un code (étranger ou le vôtre) je vous conseille un outil particulièrement efficace, Visual NDpend[18] de Patrick Smacchia, MVP C#. Je ne peux hélas pas entrer ici dans les détails de la présentation de cet outil d'une grande puissance mais je conseille au lecteur d'aller sur le site de NDepend pour comprendre tout ce qu'il peut faire. Il ne faudrait pas réduire ce produit aux quelques schémas publiés ici.*

Le Tree Map et les espaces de noms

Cette vision particulière du code obtenue selon plusieurs critères (ici le nombre d'instructions IL) permet de visualiser rapidement les classes les plus importantes ainsi que les espaces de noms et leur taille respective.

[18] La page du site de NDepend : http://www.ndepend.com/Default.aspx

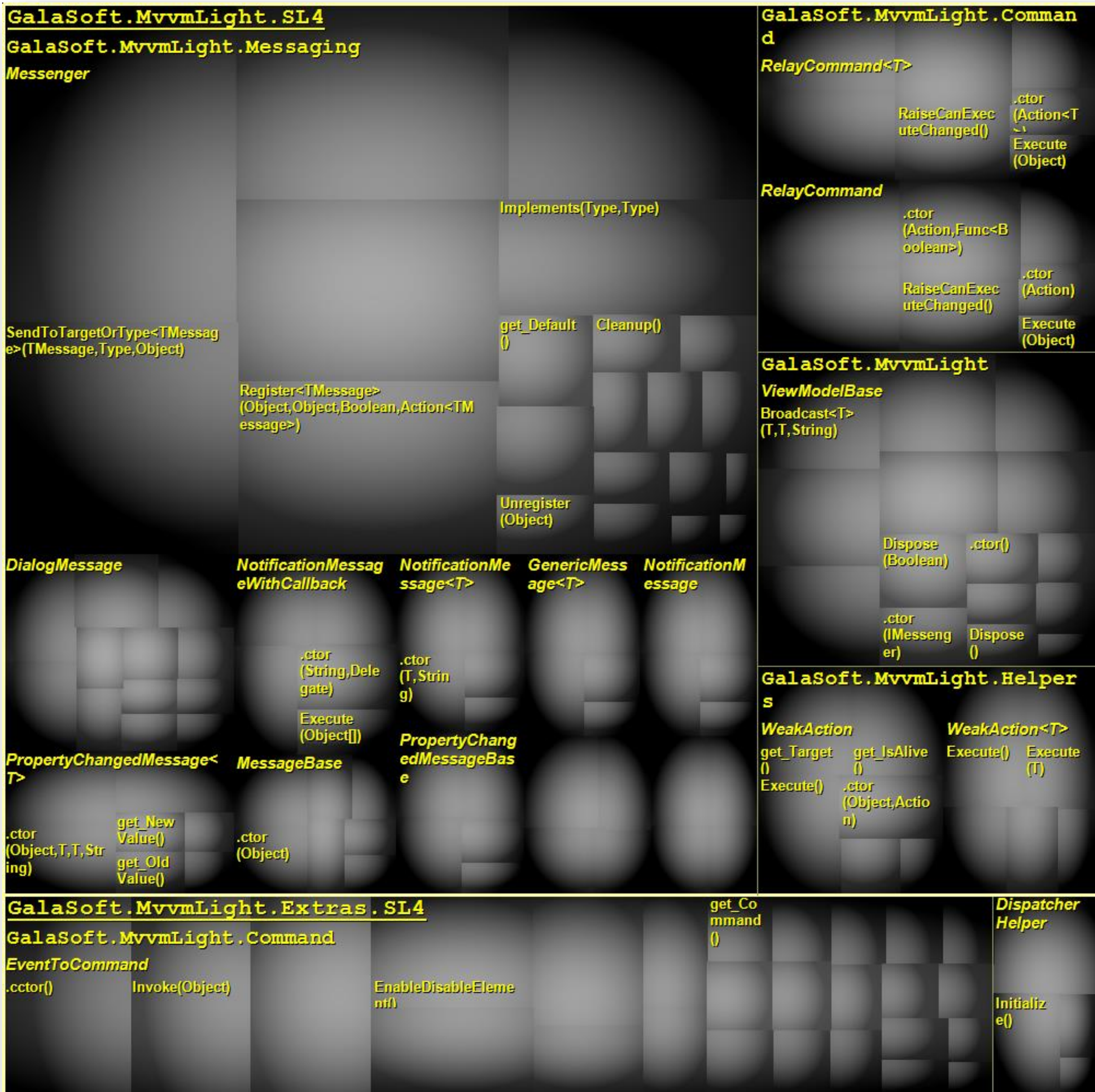


Figure 33 - Le Tree map du MVVM Light

On découvre ainsi les grandes parties qui constituent la librairie, parties sur lesquelles je reviendrai sans cesse dans la suite de cet article :

GalaSoft.MvvmLight.Messaging

Cet espace de noms regroupe les fonctions de messagerie. On voit nettement que ce bloc occupe une place importante dans l'ensemble MVVM Light, aussi bien en quantité de code que fonctionnellement (ça c'est moi qui vous le dit, le schéma ne le montre pas !). Les

classes essentielles de ce bloc sont Messenger, le gestionnaire de messagerie, et toutes les classes dérivées de MessageBase.

GalaSoft.MvvmLight.Command

Cet espace de noms regroupe les classes afférentes à la gestion de commande. La place modeste occupée ne doit pas laisser croire, au contraire, que cette partie serait moins essentielle que la précédente. Ce bloc expose principalement la classe RelayCommand et sa version générique RelayCommand<T>.

GalaSoft.MvvmLight.ViewModelBase

Cet espace de noms offre les bases pour créer les Modèles de Vue. Nous en verrons bien entendu le détail. La classe ViewModelBase dont hériteront vos Modèles de Vue est la plus importante de ce bloc.

GalaSoft.MvvmLight.Helpers

Toute librairie possède quelque part une zone de ce type où se trouve le code utilitaire utilisé par tout ou partie de l'ensemble. Ce bloc expose actuellement principalement WeakAction et sa version générique. Il s'agit de classes pouvant pointer une Action en utilisant une *Weak Référence* ou référence faible afin d'éviter les pertes mémoires. Pour ceux qui ne connaissent pas ce concept je vous conseille la lecture de mon article « Les références faibles sous .NET » un PDF assez court mais instructif (adresse : http://www.e-naxos.com/DnlManager.aspx?GROUP=12&FILEID=70).

GalaSoft.MvvmLight.Extras

Cet espace de noms définit des classes ajoutées récemment dans la librairie. Comme le nom le laisse supposer il ne s'agit pas d'un bloc de base mais de fonctions supplémentaires. On y trouve principalement le behavior EventToCommand et le DispatcherHelper. Des classes dont nous verrons plus loin l'utilité.

Cette première vision du code n'est qu'une approche simple, mais elle offre une vue d'ensemble sur les espaces de noms et les classes principales, un bon moyen de « prendre contact » avec la librairie.

La matrice des dépendances

Il s'agit d'une vue qui plonge plus en profondeur sur le code en montrant les dépendances existantes entre les diverses classes. Le snapshot que je vous propose possède un grain moyen, ni trop gros, ce qui n'apprendrait pas grand-chose, ni trop fin, ce qui serait trop long à comprendre.

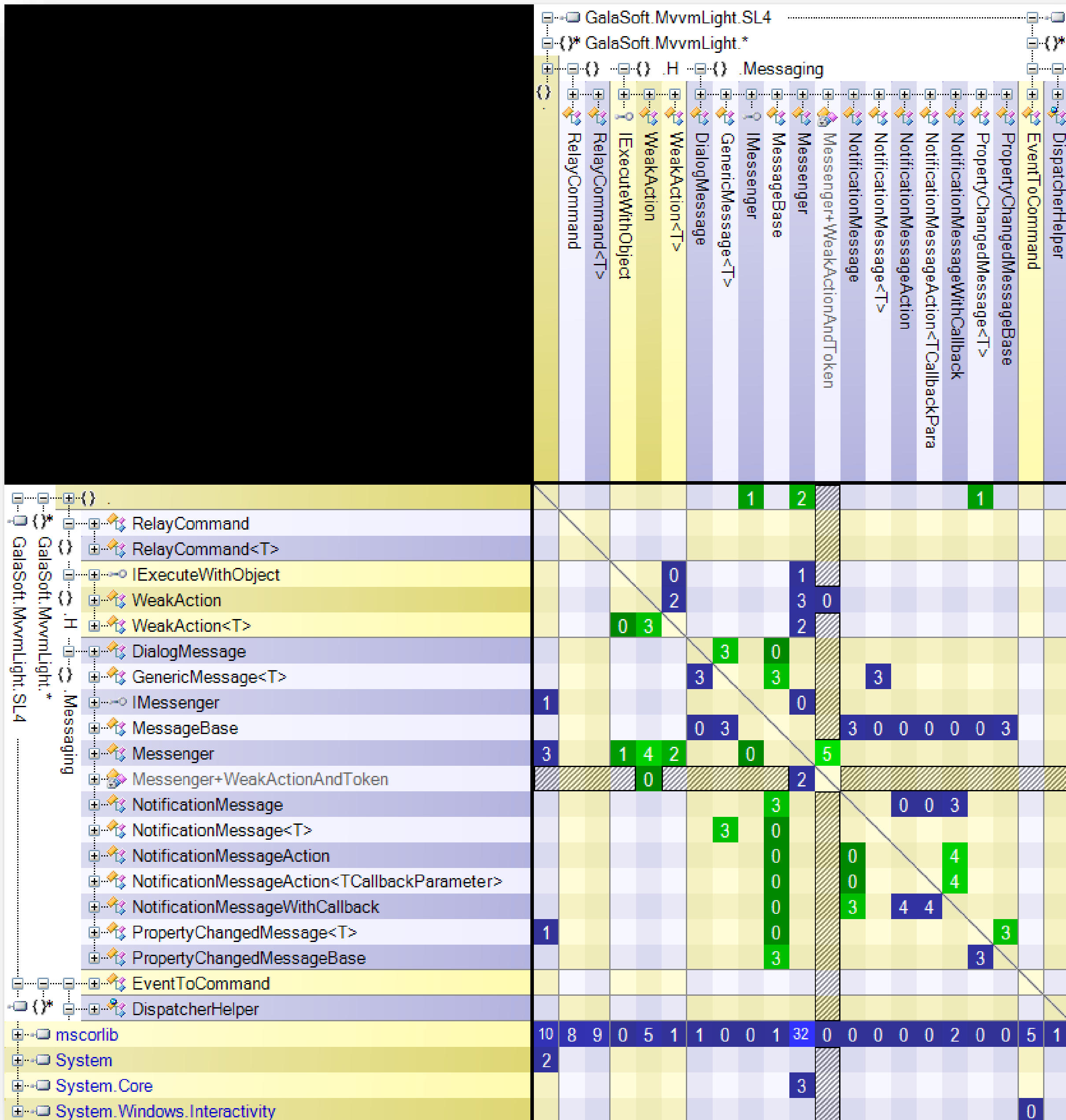


Figure 34 - La matrice des dépendances de MVVM Light

Comme ce schéma le montre, les classes de la librairie sont très souvent en relation directe avec mscorlib, le noyau du Framework .NET et l'interdépendance des classes est assez faible. MVVM Light est composé de blocs distincts n'ayant que peu de relations entre eux, ce qui est assez logique puisqu'il s'agit de fonctions assez différentes. Cela nous assure accessoirement qu'il ne s'agit certainement pas de code spaghetti…

La matrice graphique des dépendances

La représentation graphique des dépendances est souvent plus agréable à l'œil et plus facile à lire que la matrice précédente bien qu'il s'agisse des mêmes données[19] .

(page suivante pour assurer une meilleure lisibilité au schéma)

[19] Voici un bon sujet de réflexion pour tout designer en herbe d'applications Silverlight et WPF : l'influence d'un mode de représentation des données sur la compréhension de ces dernières et la mise en évidence des informations essentielles qu'elles véhiculent…

Figure 35 - Matrice graphique des dépendances

La résolution de l'image ne s'accorde que très peu aux limites du PDF de cet article, mais on peut malgré tout voir les principales relations ainsi que les nœuds les plus importants de la librairie.

Il faut voir ces schémas comme des aides pour saisir l'ensemble de la librairie, si vous disposez de NDepend vous pourrez reproduire facilement ces matrices et les afficher avec la résolution *ad hoc* pour profiter de chaque détail… Pour les autres vous noterez la présence des JPG de ces schémas dans le Zip de l'article.

Le diagramme de classes

In fine, c'est le document que préfèrent souvent les développeurs. Il montre simplement les classes et leurs relations, cela est loin de tout dire, mais il est vrai que ce diagramme UML est certainement le plus populaire parmi les 13 que propose UML 2.0.

Découpons le diagramme par segmentation fonctionnelle.

Les outils de base

MVVM Light utilise plusieurs interfaces, donc IExecuteWithObjet qui est principalement utilisée par WeakAction<T>. Comme annoncé, je n'entrerai pas dans le détail de tout le code de MVVM Light. Mais il faut dire quelques mots de WeakAction et de sa version générique.

A certains endroits, MVVM Light doit stocker des références vers des Action, de telles références créées, *de facto*, une relation entre l'objet stockeur et l'objet qui implémente l'Action ce qui peut, dans certains cas, empêcher la libération par le *Garbage Collector* de certains objets devenus inutiles.

Donc plutôt que de stocker des références de ce type vers les Action, MVVM Light utilise la classe WeakAction (et sa version générique) qui utilise des *Weak References* (références faibles[20]). L'objet pointé n'est pas réservé, il peut être libéré dès qu'il devient inutile. La référence faible notera simplement que l'Action pointée n'existe plus et ne l'invoquera plus. Ce procédé évite les « *memory leaks* » (pertes mémoire) et démontre le soin apporté à l'implémentation de MVVM Light pour rendre la librairie utilisable en production.

[20] Voir « Les références faibles sous .NET » un PDF assez court mais instructif (adresse : http://www.e-naxos.com/DnlManager.aspx?GROUP=12&FILEID=70)

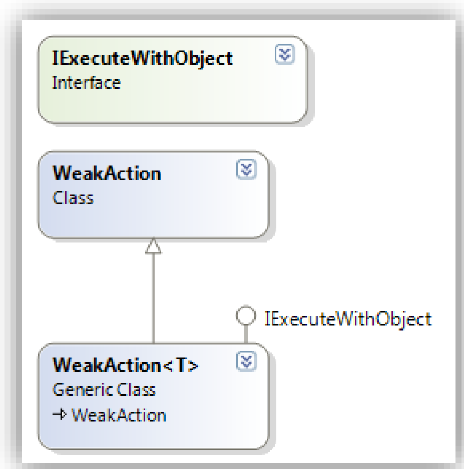


Figure 36 - Les WeakAction

La messagerie

Partie essentielle de MVVM Light comme nous le verrons souvent ici, la messagerie se compose d'une interface et d'une classe d'implémentation de cette dernière.

Figure 37 - IMessenger et Messenger

Messenger est une messagerie simple mais complète auprès de laquelle un récepteur s'enregistre pour recevoir des messages émis par un ou plusieurs émetteurs. Nous étudierons cela en détail plus loin.

Messenger implémente IMessenger, ce qui vous permet de créer, si besoin était, votre propre mécanisme de messagerie tout en restant compatible avec MVVM Light.

RelayCommand

RelayCommand et sa version générique permettent de simplifier l'implémentation des commandes sous MVVM Light. C'est une partie importante de MVVM Light, même si, en code, cette partie est assez légère. Nous étudierons bien entendu le mécanisme des commandes plus loin dans cet article.

Figure 38 – RelayCommand

ICleanup et ViewModelBase

ViewModelBase est la classe dont doivent hériter les Modèles de Vue sous MVVM Light. Ce n'est pas une obligation stricte, mais les services rendus par la classe de base sont précieux (messagerie, gestion de PropertyChanged) et devraient de toute façon être implémentés dans les Modèles de Vue. Autant gagner du temps et utiliser les services de cette classe.

ICleanup est une interface de nettoyage remplaçant dans les versions récentes de MVVM le modèle IDisposable. ViewModelBase supporte cette interface qui permet ainsi de demander à une Modèle de Vue de faire le ménage en relâchant les ressources qu'il possède sans toutefois être obligé de disposer l'instance. Certains peuvent trouver là une sophistication inutile, d'autre au contraire un mécanisme très pratique… Il n'y a de toute façon aucune obligation, chacun utilise ou non cette facilité selon ses besoins.

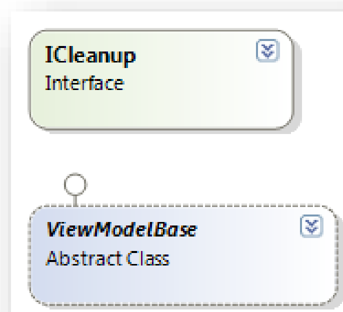


Figure 39 - ViewModelBase et ICleanup

Les messages

Je vous propose un diagramme complet des messages un peu plus loin (voir *Figure 42 - Les différentes classes de messages de MVVM Light,* page 180), il semble donc inutile de répéter cette partie du diagramme de classes.

Les classes de la librairie « Extras »

Ajoutée récemment à MVVM Light, la librairie « Extras », comme son nom le laisse supposer, contient des éléments non essentiels au fonctionnement de MVVM Light. Toutefois ils étendent de façon pratique les possibilités de la librairie.

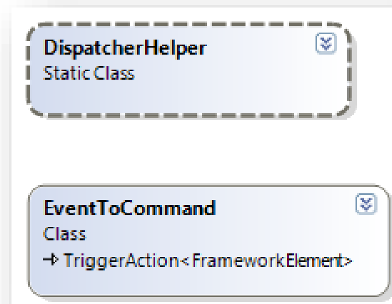


Figure 40 - Les Extras (DispatcherHelper et EventToCommand)

EventToCommand est un behavior, il a la particularité de permettre facilement la connexion d'un événement quelconque à une commande ICommand avec passage éventuel des paramètres originaux de l'événement. Ce mécanisme étend très largement le système de commande de Silverlight qui est plus pauvre que celui de WPF mais profite aussi à ce dernier et autorise l'écriture d'un code portable entre les deux environnements. Nous verrons des exemples pus loin.

DispatcherHelper est un *helper* qui permet de router automatiquement l'invocation d'une opération vers le thread de l'UI. Les templates de projet MVVM Light intègrent l'initialisation de la classe (qui doit repérer le thread de l'UI) mais il reste possible d'ajouter cette classe après coup dans un projet. Le grand avantage de cet *helper* est de ne pas avoir à se soucier des problèmes liés au thread de l'UI qui est le seul à pouvoir faire des mises à jour de l'affichage. En faisant transiter les Action via DispatcherHelper on est certain que quelques soient les circonstances, l'Action sera exécutée dans le thread de l'UI évitant ainsi mauvaises surprises et bugs difficiles à trouver parfois.

LE PATTERN MVVM ET MVVM LIGHT

La présentation du code qui précède se situe à un niveau très haut, ce qui permet de comprendre quelles sont les parties principales de la librairie. Mais il faut bien dire que cela nous en apprend assez peu sur le rôle de chacune de ses parties dans le grand jeu du pattern MVVM !

C'est pourquoi la première des choses à comprendre lorsqu'on parle de MVVM Light est de savoir **comment cette librairie s'insère dans la logique de mise œuvre du pattern MVVM**. C'est ce que je vais vous présenter dans cette section.

Tous les Frameworks déclarent de nombreuses classes pour leur fonctionnement dont seules quelques-unes sont généralement utilisées en « frontal ». MVVM Light ne déroge pas à la règle mais en raison même de sa spécialisation et de l'esprit « light » qui a présidé à sa conception, presque toutes les classes sont « frontales ». Il y a très peu de code « interne » et encore moins de code superflu.

Pour aider à la mise en œuvre du pattern Model-View-ViewModel, MVVM Light s'est attaché à régler quelques points clés et uniquement ceux-là. Nous allons voir maintenant lesquels et comment MVVM Light répond à ces besoins précis.

Rappelons que le pattern MVVM s'architecture autour de quelques principes simples dont découlent des besoins spécifiques de mise en œuvre :

- La séparation entre code fonctionnel et l'interface utilisateur (page 171)
- La gestion des commandes (page **Erreur ! Signet non défini.**)
- La communication asynchrone entre les tiers (page 177)
- Le tiers « Modèle de Vue » (ViewModel, page 185)

Dans la section qui suit nous allons voir comment MVVM Light répond à chacun de ces besoins.

La séparation Code fonctionnel / Interface Utilisateur

MVVM pose comme l'un de ces buts d'aider à une totale séparation entre le visuel (l'interface) et le code de l'application.

La technique de base consiste à séparer clairement le Modèle (les données) de la Vue (l'interface) en créant un intermédiaire le « Modèle de vue » (ViewModel) chargé de la partie fonctionnelle autant que de l'adaptation des données pour la Vue et le maintien de l'état de cette dernière (C'est un raccourci, voir mon article complet sur MVVM pour une définition plus fine et plus exacte).

La connexion entre la vue et son Modèle de Vue s'effectue par la propriété DataContext de la Vue que l'on fait pointer vers une instance du Modèle de Vue. Ce sont là les principes de base de MVVM dont j'ai longuement parlé dans l'article indiqué en introduction. Tout cela est du ressort de la méthodologie, il n'y a besoin, à la base, de rien de spécial pour suivre cette partie du pattern.

Aider à la séparation code / visuel

Dans les faits tout ce qui peut amener à une meilleure isolation entre Vue et Modèle de Vue est une bonne chose. Si on applique le pattern « tel quel », il suffit donc de créer une classe quelconque exposant des propriétés (le Modèle de Vue) et de coder dans la partie Xaml de la Vue une référence vers une instance de cette classe via le DataContext de la Vue.

C'est bien, c'est déjà du MVVM mais ce n'est pas très satisfaisant. Cela créé en effet un lien « en dur » entre la Vue et son Modèle de Vue. Il est préférable de rendre ce lien plus flexible, ce qui renforcera le découplage et simplifiera la mise en place des tests par exemple.

Inversion de contrôle

MVVM Light propose à ce niveau d'ajouter une **inversion de contrôle** (*Inversion Of Control*, IoC) pour renforcer plus encore l'isolation entre Vue et Modèle de Vue. Parmi les deux solutions envisageables, le localisateur de services (*Service Locator*) et l'injection de dépendance (*Dependency Injection*), MVVM Light a opté pour la plus simple à comprendre et à mettre en œuvre, le localisateur de services.

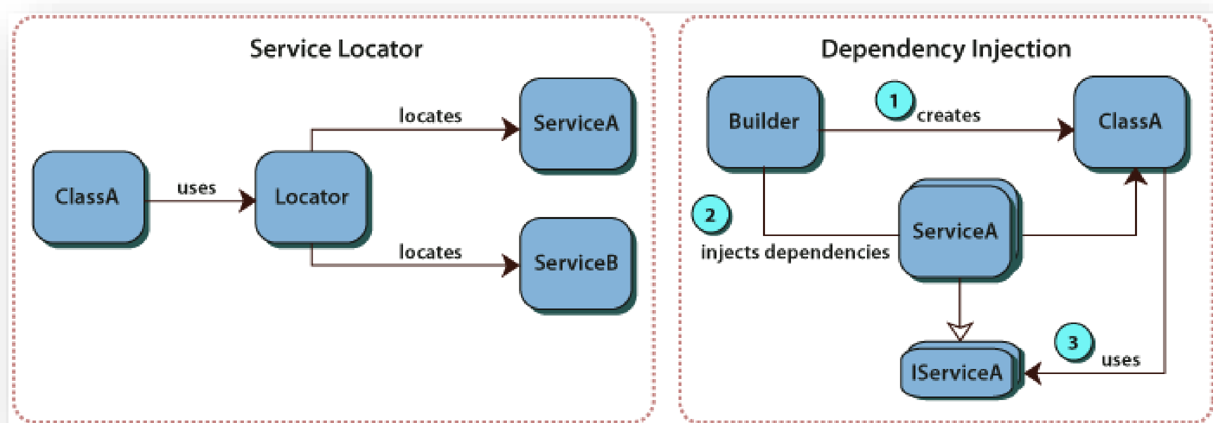


Figure 41 - Les patterns de l'Inversion de Contrôle

Nota : L'inversion de contrôle est aussi présentée en détail dans l'article de référence sur MVVM.

Le but est en fait de construire une application qui, plutôt qu'un monstre monolithique, se présente sous la forme de **blocs de plus petite taille n'ayant aucune dépendance directe entre eux**.

On comprend bien que l'esprit de l'IoC est avant tout de donner un moyen de maîtriser l'extravagante complexité galopante des logiciels modernes. On ne peut plus développer aujourd'hui un logiciel WPF ou Silverlight comme on développait un utilitaire console en C il y a 20 ans. *Quand on a compris cette évidente motivation, on a compris l'IoC*.

Bref, l'IoC vise un but pratique : séparer le code en petites unités non dépendantes directement les unes des autres. *C'est un pattern générique, applicable à tout langage dans toute circonstance et n'entretient pas de liens particuliers avec MVVM, c'est juste un pattern*

qu'on peut appliquer, entre autre, à certains problèmes soulevés par la mise en œuvre de MVVM.

Quand on replace tout cela dans le cadre de la problématique bien spécifique abordée ici, à savoir la séparation entre la Vue et son Modèle de Vue, ce qui est déjà une pattern (MVVM ici mais qui pourrait être MVC, MVP ou tout autre pattern ayant cet objectif), **cela signifie tout simplement qu'on veut éviter que le DataContext de la Vue ne pointe directement sur l'instance du Modèle de Vue.** C'est tout ☺

Pour en revenir à Mvvm Light, la librairie a donc opté pour l'Inversion de Contrôle appelée Localisateur de Service (*Service Locator*) afin de séparer plus encore la Vue de son Modèle de Vue.

Un Localisateur de services est une stratégie simple consistant à créer un catalogue des services existants et à leur donner un nom (une clé par exemple). Quand un code veut accéder à un service, plutôt que de se lier directement à ce dernier à la compilation (technique dite de *early binding*, ligature précoce) il va demander, à l'exécution, au localisateur de services de lui fournir le service en question en présentant son nom (sa clé). C'est une technique de *late binding* (ligature différée).

Dans le cadre de MVVM Light, le Localisateur de Services se résume à sa plus simple expression : une classe qui expose des propriétés statiques (jouant le rôle de clé d'accès), chacune retournant une instance de ViewModel différent. Chaque Vue se lie à son Modèle de Vue en liant sa propriété DataContext non pas directement à une instance de ce Modèle de Vue mais créant un binding avec la classe Localisateur de Service sur l'une de ses propriétés : le Modèle de Vue qu'on cherche à atteindre. Le Localisateur de services créé ainsi une indirection entre la Vue et son Modèle de Vue.

Ce montage particulier réclame de modifier la classe Localisateur de Services à chaque fois qu'on ajoute un Modèle de Vue (en tout cas dans l'implémentation choisie par MVVM Light).

De ce fait, tout ce mécanisme est totalement géré par l'application elle-même, MVVM Light ne jouant absolument aucun rôle ici, ce qui est paradoxal !

Mais si le Localisateur de Service n'est pas intégré à MVVM Light en tant qu'objet fini c'est parce qu'il doit être modifié en fonction des nouveaux Modèles de Vue ajoutés à l'application et que le principe même retenu (une classe exposant des propriétés statiques) est tellement simple qu'il n'y a aucun besoin d'un Framework quelconque pour atteindre facilement l'objectif.

Toutefois, MVVM Light fournit des *templates* de projets qui, justement, contiennent déjà l'ébauche de cette architecture qu'il n'y a plus qu'à faire évoluer… L'unité de code créée s'appelle ViewModelLocator.cs, elle est placée dans le sous-répertoire ViewModel de l'application. La classe exposée s'appelle ViewModelLocator et contient, au départ, une propriété pour le Modèle de Vue de la fiche principale, propriété portant le nom de Main (et doublée d'une MainStatic, déclarée de façon static, mais nous y reviendrons).

Je reviendrai sur le localisateur de services plus loin lorsque nous verrons des exemples plus concrets. Pour l'instant continuons le tour des différentes facettes de MVVM Light et *rappelons simplement que pour nous aider à séparer l'interface visuelle du code, MVVM Light nous propose de gérer une indirection entre Vues et Modèles de Vue en passant par un Localisateur de Services.*

La gestion des commandes

Un autre des points essentiels du pattern MVVM est **le mécanisme par lequel les commandes données par l'utilisateur sont acheminées jusqu'au Modèle de Vue**, puisque la Vue ne contient plus aucun code fonctionnel (elle peut en revanche posséder du code de gestion de l'interface).

ICommand

Le Framework .NET de Silverlight 4 a introduit une notion déjà présente dans WPF : l'interface ICommand.

Cette interface, comme toutes les interfaces, est un contrat que certaines classes peuvent accepter de remplir. L'interface ICommand est un contrat très simple qui représente une action à exécuter.

Cette interface propose deux méthodes et un événement :

- CanExecute() qui doit retourner true ou false selon que la commande est disponible ou non
- Execute() qui exécute réellement la commande
- CanExecuteChanged, un événement que l'objet commande peut lever si l'état de CanExecute() change, ce qui, en cascade permet à l'IHM de se mettre à jour.

Dans l'idéal, si Silverlight était WPF, ce que je viens d'écrire concernant l'événement CanExecuteChanged serait vrai. Or, comme le CommandManager n'existe pas sous Silverlight, ce retour d'information de la commande vers l'IHM ne se fait pas tout seul et le développeur devra utiliser d'autres méthodes pour maintenir son interface à jour (état Enabled des boutons par exemple). Notons que MVVM Light nous permet de contourner ce manque en utilisant des messages ce que nous allons aborder dans quelques paragraphes.

Une commande = une propriété bindable

Les actions (ou commandes) ainsi représentées par des interfaces peuvent être pointées par des variables (de type ICommand). Qui dit variable, dit binding. Les commandes contenues dans le Modèle de Vue sont ainsi exposées sous la forme de propriétés de type ICommand. La Vue étant déjà liée à son Modèle de Vue par son DataContext, les éléments d'interface peuvent donc se lier par binding aux variables représentant les commandes. Le code des actions a ainsi disparu de la Vue pour être placé dans le Modèle de Vue. C'est un peu magique, et c'est l'un des points clé de la mise en œuvre du pattern MVVM.

Bien entendu tout n'est pas aussi simple. Comment un élément d'interface peut-il se lier à une variable de type ICommand ?

Le support limité de ICommand

Il est évident que l'ajout de ICommand seul ne règlerait rien. C'est pourquoi la classe ButtonBase de Silverlight a été modifiée pour exposer une propriété Command, de type ICommand. ButtonBase est la classe dont dérive par exemple Button (sous WPF on retrouve la même logique). C'est donc en liant par data binding la propriété Command d'un dérivé de ButtonBase à une propriété ICommand d'un Modèle de Vue qu'on complète la mise en place du mécanisme. La classe ButtonBase, lorsqu'elle doit déclencher l'action qu'elle représente, sait reconnaître la présence de la commande qu'on lui a fournie et sait l'invoquer. Cela remplace la gestion de l'événement Click pour le bouton par exemple. On notera que sous WPF la prise en charge des commandes est plus complètes (notamment par la présence de CommandManager) mais n'est pas aussi évoluée qu'on pourrait l'espérer.

Arrivé à ce point on s'aperçoit que tout est déjà dans « la boîte ». Où intervient MVVM Light alors ?

Simplifier la création des commandes avec RelayCommand

La création des commandes, puisqu'il s'agit de supporter une interface, ICommand, réclame donc la *création d'une classe*. Et comme une classe ne peut fournir qu'une seule implémentation d'une même interface, on en conclue facilement *que chaque commande exposée par le Modèle de Vue est en réalité une instance d'une classe spécifique*. S'il y a cent commandes dans l'application, il y aura donc cent classes à écrire.

Certes ces classes sont simples puisqu'elles n'ont pour obligation que de supporter ICommand et ses méthodes (et aussi le code fonctionnel de la commande !). Mais il faut avouer qu'au niveau de la productivité cela pèse lourd dans l'écriture d'une application, car des commandes, une application en expose souvent beaucoup.

C'est là qu'entre en scène RelayCommand. L'astuce tient au fait que RelayCommand est déjà une classe implémentant ICommand, mais qu'au lieu de coder en dur chacune des méthodes

de cette interface elle laisse la porte ouverte en quelque sorte. Cette porte, ces portes pour être exact, ce sont des délégués (*delegates*).

Ainsi, plutôt que de créer une classe pour chaque commande, ce qui est un peu lourd il faut en convenir, il suffit de créer une instance de RelayCommand en lui passant les méthodes nécessaires à son fonctionnement.

On comprend dès lors le nom de la classe, RelayCommand : elle *relaye* des commandes.

Une commande peut avoir besoin de paramètres, l'implémentation de RelayCommand est trop simple pour prendre en compte la diversité de ces derniers. C'est pourquoi MVVM Light propose une seconde implémentation, générique, RelayCommand<T>, qui permet d'indiquer le type du paramètre attendu par la commande.

Echapper à la limitation du support de ICommand

Comme on le voit le support de ICommand, même en le simplifiant avec RelayCommand, ne règle pas tous les besoins, loin s'en faut ! Car, de base, seules les classes dérivant de ButtonBase possèdent une propriété Command déclenchée généralement par le clic sur l'objet visuel. Il y a souvent des boutons ou dérivés dans une application, mais pas que ça !

Prenons le simple exemple d'un Slider. Ce qu'il envoie n'est pas un « clic » mais un événement de changement de valeur lorsque le curseur est déplacé. Pas de support de ICommand ici… Comment le Modèle de Vue pourra-t-il réagir à ce changement s'il n'existe pas de lien entre le changement de valeur du Slider et le code du Modèle de Vue ?

A ce stade on a donc réglé qu'une partie de la problématique posée par les commandes sous MVVM ce qui impliquerait de faire du bricolage pour le reste. Une situation qui n'est vraiment pas satisfaisante !

Si WPF propose des mécanismes plus subtils pour la gestion des commandes, il n'en est pas de même sous Silverlight…

Transformer les événements en commandes

Heureusement, dans ces versions récentes, MVVM Light nous propose une solution générique et plutôt élégante : EventToCommand. Il s'agit d'un TriggerAction. Ce behavior se comporte donc comme un Trigger, un déclencheur. Il suffit de l'ajouter à un objet, par exemple le Slider, et d'indiquer que l'on souhaite détourner l'événement ValueChanged. A chaque fois que ce dernier sera déclenché par le Slider, le Trigger du *behavior* le sera. C'est la moitié du travail d'un TriggerAction… Car une fois le Trigger déclenché que faire ? Déclencher en cascade l'Action enregistrée : une commande supportant ICommand ! On peut même récupérer les arguments de l'événement original ce qui est souvent essentiel.

L'astuce est simple et facile à mettre en œuvre : pour tout ce qui n'est pas dérivé de ButtonBase et qui ne possède pas de propriété Command il suffit d'utiliser un behavior EventToCommand pour relier un événement de l'objet à une commande du Modèle de Vue. Et le tour est joué.

On notera que le behavior en question est proposé dans la seconde Dlls de MVVM Light (« Extra »).

Enfin on appréciera que cette solution très utile sous Silverlight peut parfaitement être utilisée sous WPF ce qui permet de produire un code très portable entre ces deux saveurs de Xaml.

La communication

Dans un environnement fragmenté, conçu sous la forme de blocs n'ayant pas de dépendances directes, se pose très vite la question de savoir comment synchroniser les actions entre ces blocs…

En effet, la vue est liée à son Modèle de Vue (au travers du localisateur de services) par sa propriété DataContext. La Vue peut ainsi lier certains de ces éléments à des propriétés exposées par le Modèle de Vue. De même, le Modèle de Vue peut directement (ou par le biais d'un autre localisateur de services), se lier à un ou plusieurs Modèles (données).

Ces liaisons sont à sens unique et uniquement verticales. A *sens unique* car le Modèle ne doit rien savoir des Modèles de Vue qui l'utilisent de même que les Modèles de Vue ne doivent rien connaître des Vues qui les exploitent[21]. Et *verticale* car on voit bien que ses relations suivent une « flèche de connaissance » qui part de la Vue pour aller au Modèle. La vue peut connaître son Modèle de Vue (par force puisque son contenu en dépend), le Modèle de Vue peut connaître certains Modèles (par force puisqu'il en exploite les données). *Qu'en est-il des Vues entre elles ou des Modèles de Vues les uns vis-à-vis des autres ? Plus délicat encore, comment un Modèle de vue peut-il dialoguer avec sa Vue alors qu'il est censé n'en rien savoir ?*

En fait, le pattern MVVM implique une telle segmentation et des barrières tellement hautes entre les différents tiers (Modèle, Vue, Modèle de Vue) que la seule communication qui existe est à sens unique et uniquement verticale et encore ne repose-t-elle que sur le mécanisme de data binding, donc de propriétés liées à d'autres propriétés ce qui ne peut répondre à tous les besoins.

---

[21] Attention : le « sens unique » ne porte que sur le sens du lien de « connaissance » entre les blocs, les data bindings établis entre la Vue et son Modèle de vue peuvent être « two way » bien entendu, ce qui ne change rien au fait que la relation Vue/Modèle de Vue est à sens unique…

Le besoin d'un mécanisme de messagerie

Un cadre aussi rigide ne serait guère réaliste.

Comment ne pas le casser pour en conserver tous les avantages tout en introduisant la flexibilité nécessaire pour que les communications horizontales et verticales puissent s'établir et en tous sens ?

La réponse tient en un mot : **message**.

Windows est un bon exemple d'un type proche de cette problématique de communication entre blocs distincts : un bouton avertit son code qu'il a été cliqué en plaçant un message dans une file d'attente, file qui est traitée par la boucle principale du programme. La souris bouge-t-elle de quelques centimètres ? Ce sont des centaines de messages d'un type particulier (c'est-à-dire véhiculant des informations particulières) qui sont émis dans la grande file d'attente de l'OS pour être distribués ensuite aux applications qui en ont fait la demande.

Le problème est donc loin d'être récent et la solution existe de longue date : *dans un système (au sens large) composés d'objets n'ayant que peu ou pas de connaissance sur la nature des autres objets (par design ou par obligation), le seul moyen simple de leur permettre de se synchroniser pour effectuer une tâche demandant leur collaboration est de mettre en place une messagerie.*

C'est aussi comme cela que les humains ont réglé un problème identique entre individus ne se connaissant pas mais pouvant collaborer à un projet commun : le mail !

Messenger, le messager

MVVM Light propose ainsi un **service de messagerie**. Le gestionnaire de cette messagerie est la classe Messenger qui supporte l'interface IMessenger. Par défaut, Messenger expose une propriété statique fournissant une instance de messagerie directement utilisable. Le développeur utilise généralement Messenger et ses mécanismes internes mais il peut très bien écrire son propre gestionnaire de messagerie en supportant IMessenger et en fournissant une instance valide de ce nouveau service de messagerie à la classe Messenger qui l'utilisera en place et lieu de son propre code.

Dans un environnement objet, les messages sont aussi des objets. Et ils peuvent être de nature très différente les uns des autres, c'est-à-dire que les informations qu'ils véhiculent sont elles-mêmes de nature très variée. Si nous revenons sur l'exemple de l'OS, un message de déplacement de la souris ne contiendra pas les mêmes informations qu'un message de ShutDown avertissant toutes les applications que Windows va se fermer par exemple.

Il en va de même au sein d'une application : Les messages, du moins leur contenu, vont dépendre de la communication spécifique qui sera établie. C'est pourquoi chaque type de message donne lieu à l'écriture d'une **classe message spécifique**, exactement comme les arguments des événements. Ceux d'un clic souris sont d'une classe différente de ceux d'un MouseMove.

On se retrouve ici dans le même cas que celui des commandes : l'application *stricto sensu* du pattern MVVM risque de faire exploser le nombre de classes à écrire. Pour les commandes MVVM Light nous offre une solution plus pratique en fournissant une classe qu'il n'y a plus qu'à instancier (RelayCommand). Pour les messages il va en être de même. C'est peu de chose (d'un point vue conceptuel et même de celui de la complexité du code fourni) mais c'est tout ce qui fait l'avantage de MVVM Light : simplifier la mise œuvre de MVVM !

Par exemple, il existe de nombreux cas qui reviennent souvent, il y a même de nombreux autres cas où le message pourrait être symbolisé par une simple valeur (un nombre, une chaîne de caractères). Pour tous ces cas il serait réellement fastidieux d'avoir à créer une classe spécifique.

MVVM Light prend en compte ces cas particuliers et fournit une arborescence de classes dérivant de MessageBase. Cette arborescence n'est pas très complexe mais elle permet de couvrir la plupart des besoins :

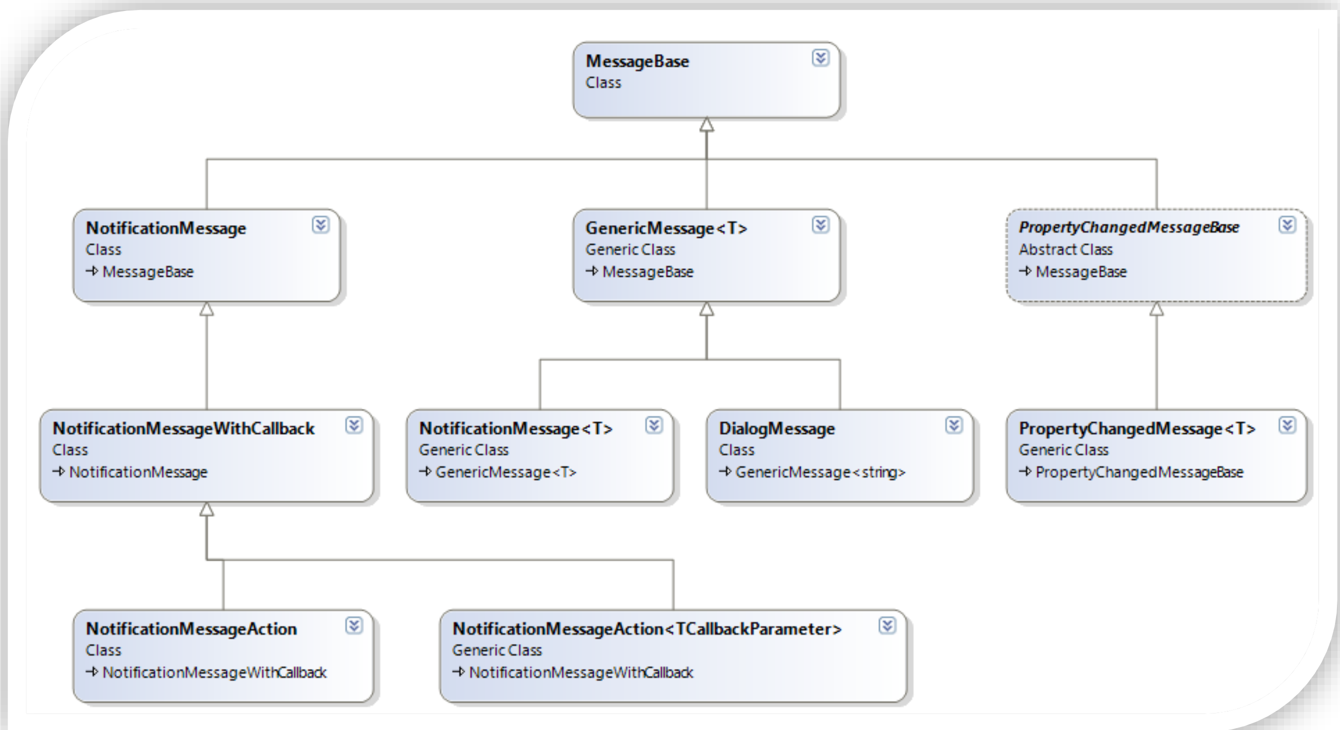


Figure 42 - Les différentes classes de messages de MVVM Light

Le diagramme ci-dessus montre l'arborescence née de la classe MessageBase.

Dans le principe, tout le monde peut envoyer un message, mais seuls ceux qui se sont abonnés à certains messages les recevront.

Ce modèle n'est pas une invention non plus et il tombe sous le sens parce qu'il est le plus souple et le plus simple. Tout le monde peut créer un blog et proposer un flux RSS, seuls ceux qui sont intéressés par tel ou tel blogueur s'abonnent à son flux.

Le Messenger de MVVM Light joue les rôles de gestionnaire et de distributeur de messages. Un objet envoie un message, le Messenger le place dans sa file d'attente et le distribue à tous les autres objets qui se sont abonnés à ce type message. Nous verrons qu'en pratique il existe plusieurs moyens de s'abonner à des messages avec différents niveaux de filtrage ce qui rend le procédé encore plus souple.

Les notifications

Mais revenons un instant sur les messages eux-mêmes. Je parlais plus haut des messages simples pouvant être symbolisés par une chaîne de caractères ou un simple code numérique. Il s'agit de messages dits de « *notification* ». C'est-à-dire qu'un émetteur veut (en général) avertir ses abonnés qu'il « *vient de se passer quelque chose* ». Le « quelque chose » étant

symbolisé par un nom, une clé, etc. Dans un tel cas assez courant il serait lassant d'avoir à écrire une classe message spécifique.

C'est la solution à ce besoin que propose la classe NotificationMessage. Elle prend un paramètre de type string qui peut symboliser ce qu'on veut tant que l'émetteur et le (ou les) récepteur(s) se sont mis d'accord sur la signification de cette chaîne.

Obtenir une réponse à un message

Parfois il est nécessaire d'obtenir une réponse à un message que l'on émet. On entre ici dans les choses un peu compliquées non pas à cause du code lui-même mais du raisonnement qui devient plus difficile à suivre en raison de **la nature asynchrone des messages**.

En fait, dans un tel système il n'est pas possible, simplement, d'émettre un message « bloquant » et d'attente la réponse tranquillement pour la traiter à la ligne de code suivante…

Lorsqu'on émet un message il part pour un voyage dont on ignore le trajet exact et le temps qu'il prendra pour arriver à son destinataire (un peu comme avec la Poste☺). On n'a pas même l'assurance qu'il y aura « quelqu'un au bout du fil » ! Si des messages pouvaient être bloquants, il se pourrait bien que parfois l'émetteur se retrouve à attendre sans fin une réponse qui ne viendrait jamais. Une situation qui n'est pas acceptable. Tout système informatique et *a fortiori* dans des environnements de type WPF ou Silverlight se doit d'être **fluide et réactif**. Laisser *by design* des possibilités non nulles de blocage serait totalement déraisonnable. On peut bien entendu interdire la notion de message ou obliger que le récepteur existe et qu'il réponde et ainsi revenir à la programmation structurée monotâche sous MS-DOS. Mais je ne crois pas que cela soit beaucoup plus raisonnable…

Ainsi, lorsqu'un message réclame une réponse il doit en réalité fournir un CallBack (l'adresse d'une méthode) qui sera appelé par le récepteur une fois qu'il aura traité le message, s'il le traite. La tâche que l'émetteur doit effectuer doit ainsi supporter d'être *découpée en deux méthodes* (celle qui prépare le travail et émet le message et celle qui reçoit la réponse et termine le travail). Cela n'est pas sans poser certains problèmes, mais nous y reviendrons plus tard. *L'asynchronisme impose des raisonnements nouveaux avec lesquels beaucoup de développeurs ont des difficultés*.

Messages avec ticket réponse

Dans ces cas particuliers MVVM Light nous offre aussi une solution prête à l'emploi : la classe de message NotificationMessageWithCallback. On peut voir cela comme un courrier qui contient une enveloppe pré-timbrée pour la réponse.

Message avec action

Parfois, la réponse à un message est une action à exécuter. L'émetteur se moque de recevoir la réponse mais en revanche il demande au récepteur « faire quelque chose » pour lui (l'action).

Il s'agit en réalité d'une variante du cas précédent mais qui accepte un délégué en plus d'un paramètre de notification. La variante s'appelle NotificationMessageAction et, naturellement, prend un paramètre de type Action plutôt qu'un délégué quelconque. L'utilisation reste sensiblement la même : le récepteur peut invoquer l'action une fois qu'il a reçu et traité le message. *Il peut aussi n'y avoir aucun récepteur, ou plein, et il se peut que la réponse ne vienne jamais. Tout cela doit être pris en compte dans la façon de programmer avec des messages*.

La généricité

Ce système de messagerie est simple et fournit déjà de base des classes suffisamment proches des besoins courants. Mais il manque un peu de souplesse à l'édifice. MVVM Light a ainsi évolué en proposant une seconde voie dérivée de MessageBase, une voie générique.

On trouve ainsi un GenericMessage<T> qu'on pourrait comparer au NotificationMessage, mais au lieu d'imposer un paramètre de type string la généricité autorise l'emploi de n'importe quel objet pour le paramètre (un double, un entier, mais aussi une classe complexe ou simple, selon les besoins). Et cela toujours sans avoir à coder de descendant de la classe MessageBase.

GenerericMessage<T> donne naissance à NotificationMessage<T> qui fait un peu la fusion de tout cela : il gère un paramètre chaîne (la notification) et une valeur quelconque (le paramètre). Cela peut être pratique pour signifier qu'un événement vient de se produire et, en même temps, encapsuler dans le message les informations nécessaires au traitement de cet événement.

Le cas … délicat des dialogues

Dérivant de GenericMessage<T> nous trouvons un autre message qui au lieu d'étendre encore plus la généricité se spécialise au contraire pour une tâche particulière, c'est DialogMessage. Cette classe permet de gérer des messages destinés à afficher des dialogues. Et cela règle (en partie) un problème épineux.

Tout ne se passe pas dans l'interface utilisateur, je veux dire par là que l'interaction qui s'instaure entre l'utilisateur et l'application est bien une « inter – action » et non une suite de commandes à sens unique venant exclusivement de l'utilisateur. Le programme peut lui aussi avoir besoin de réagir, de passer des informations à l'utilisateur, lui poser des questions.

Mais le programme est contenu, pour sa partie fonctionnelle, dans le Modèle de Vue. Comment le Modèle de Vue qui ne sait rien de la ou des vues qui l'utilisent pourraient-il afficher quelque chose dans l'interface alors qu'il n'a aucun accès, *by design*, à tout ce qui concerne l'interface ?

Certes un Modèle de Vue n'est que du code qui pourrait créer une fenêtre, l'afficher par-dessus l'affichage existant et attendre la répondre. Ceux qui opteraient pour une telle solution, outre d'être des barbares saccageant MVVM deviendraient aussi de grands producteurs de code spaghetti !

Dilemme… Que MVVM Light résout par le type de message DialogMessage. L'émetteur envoie un message dans lequel il peut aussi indiquer quels boutons doivent être affichés ou non, et il précise l'adresse d'un Callback. Le récepteur (généralement une Vue) va pouvoir réagir à ce message en créant une boite de dialogue (ou tout autre procédé) qui affichera le message transmis par l'émetteur. Une fois que l'utilisateur aura répondu, le récepteur retransmettra cette réponse à l'émetteur en utilisant le Callback contenu dans le message original.

Ici encore nous touchons à cette difficulté qui consiste à écrire des traitements asynchrones pouvant accomplir une tâche en plusieurs parties. L'émetteur pourra aller jusqu'à l'émission du message, mais il devra laisser la fin du travail à un CallBack. Ce découpage est loin d'être aisé dans de nombreux cas. J'y reviendrais au travers d'un exemple de code.

> *Notons au passage que la classe DialogMessage n'est qu'une aide pré-formatée avec, en tête de son concepteur, l'affichage d'un dialogue de type ok/cancel. Si on désire afficher des messages plus sophistiqués (avec des ChildWindow par exemple), il sera plus intéressant de créer sa propre classe message contenant l'ensemble des paramètres nécessaires ou d'utiliser un message générique qui passera ces paramètres au récepteur. DialogMessage n'est qu'un prétexte permettant d'introduire une problématique ardue en MVVM pour mieux y réfléchir et non une réponse globale, unique et définitive.*

Les changements de propriétés

Enfin, parmi les messages spécialisés déjà écrits car le besoin est fréquent, on trouve PropertyChangedMessageBase donnant naissance à PropertyChangedMessage<T> en version générique.

Bien entendu, une classe « *bindable* » se doit de gérer l'interface INotifyPropertyChanged, mais dans un système aux blocs fortement découplés, les tiers ne se connaissant pas directement, il n'est pas possible de poser des *listeners* sur l'événement PropertyChanged d'un autre bloc. Par exemple un Modèle de Vue A ne pourra pas écouter le changement de

valeur de la propriété P du Modèle de Vue B car en aucun cas ces deux Modèles de Vue ne peuvent ni ne doivent se connaitre. Il est donc nécessaire parfois de « doubler » l'événement PropertyChanged de l'émission d'un message ayant même sens et qui lui pourra être traité « ailleurs » sans qu'un lien figé n'existe à la compilation entre l'émetteur et le récepteur.

MVVM Light propose, au moins pour les Modèles de Vue, une classe de base qui supporte INotifyPropertyChanged ainsi qu'une propriété RaisePropertyChanged qui activera à la fois l'événement PropertyChanged et l'envoi d'un messsage PropertyChangedMessage (il s'agit d'une option de fonctionnement, de base RaisePropertyChanged n'envoie pas de message).

C'est pour cela que MVVM Light propose une classe de base pour les Modèles de Vue. Alors que les Vues et les Modèles sont des classes sans héritage particulier (du point de vue de la librairie).

> *On notera ici que MVVM Light respecte son côté « light ». La librairie pourrait très bien se compléter de mécanismes identiques pour les classes Modèles, au même titre qu'elle pourrait proposer un localisateur de services à placer entre les Modèles de Vue et les Modèles. L'avantage de MVVM Light est de mettre tout cela en évidence en ne répondant qu'au strict minimum nécessaire pour alléger la mise en œuvre de MVVM. Le développeur est libre de généraliser certaines solutions ponctuelles, voire d'en faire intervenir d'autres, en complément, développées par lui ou provenant d'autres Frameworks.*

Séquence de ShutDown pilotée par message

Laurent Bugnion, sur son site, offre une démonstration intéressante mettant en œuvre la messagerie de MVVM Light pour créer une séquence de « shutdown » dans une application complexe.

En effet, dans une application un peu large constituée de blocs distincts, il peut être intéressant, au moment de quitter l'application, de faire le tour de tous les blocs pour laisser la possibilité à chaque partie de finir ce qu'elle était en train de faire, voire d'annuler la séquence de fin en cours ou la relancer. Les messages peuvent être utilisés pour simplifier l'implémentation d'un tel mécanisme.

Plutôt que de réinventer cette démonstration, je vous laisse consulter cette page du site de GalaSoft :

http://blog.galasoft.ch/archive/2009/10/18/clean-shutdown-in-silverlight-and-wpf-applications.aspx

Une vidéo agrémente le propos et le code source de la démonstration est téléchargeable.

C'est un cas d'utilisation de la messagerie assez intéressant qui pourra certainement vous donner des idées même dans un autre contexte que l'extinction d'un programme.

Le Modèle de Vue

C'est un tiers essentiel du pattern MVVM qui joue un rôle de pivot entre la Vue et les Modèles.

De cette position particulière naissent des contraintes particulières. On a évoqué plus haut celles concernant les communications transversales ou celles portant sur les dialogues qu'un Modèle de Vue peut être amené à déclencher.

Une autre contrainte tombe sous le sens, celle de la notification des changements de valeurs des propriétés. Les propriétés d'un Modèle de Vue étant liées à une Vue via data binding il est indispensable que le Modèle de Vue avertisse la ou les Vues qui lui sont connectées dès que l'une de ses propriétés change de valeur, ce qui permet aux Vue de se rafraîchir.

Le simple support de INotifyPropertyChanged est suffisant. Mais puisqu'il est obligatoire autant créer les Modèles de Vue à partir d'une classe qui implémente déjà le code nécessaire à la gestion de l'événement PropertyChanged. Et puisque nous sommes dans un environnement cloisonné il peut être judicieux de doubler cet événement de l'émission d'un message de signification équivalente mais qui pourra être traité par d'autres blocs de code que la Vue accrochée au Modèle de Vue.

De même, un Modèle de Vue, pour supporter la « blendalibilité », se doit de fournir des données directement visibles en mode conception. Mais parfois il est impossible de fournir de telles données car le reste du logiciel est en cours de conception ou bien parce que les données n'existent pas encore. Pouvoir détecter si le Modèle de Vue est manipulé sous un éditeur (VS ou Blend) pour prendre la décision de fournir les données réelles ou des données simulées est donc quelque chose de très important.

Pour tous ces besoins, MVVM Light propose de faire descendre vos Modèles de Vue d'une classe de base ViewModelBase.

D'une part elle implémente tous ces mécanismes, d'autre part elle en ajoute d'autres comme le support de l'interface ICleanup permettant à un Modèle de Vue de « faire le ménage » sans pour autant nécessiter d'être disposé (ViewModelBase supporte aussi IDisposable).

ViewModelBase expose ainsi certaines méthodes ou propriétés qui simplifient la mise en œuvre d'un Modèle de Vue :

- La propriété IsInDesignMode qui permet de savoir si le Modèle de Vue fonctionne normalement ou bien s'il est en cours de modification sous Blend ou Visual Studio ;
- Cleanup qui supprime les références du Modèle de Vue dans le système de messagerie et peut servir aussi à relâcher les ressources possédées par le Modèle de Vue ;
- RaisePropertyChanged qui invoque PropertyChanged et qui, de façon optionnelle, peut émettre un message de changement de propriété via la messagerie ;

ViewModelBase intègre aussi un mécanisme de contrôle très intéressant : uniquement en mode Debug, l'appel à RaisePropertyChanged se double d'une vérification du nom de la propriété passée en paramètre pour vérifier son existence dans la classe.

PropertyChanged utilise en effet une chaîne de caractère devant contenir le nom de la propriété. Un refactoring incomplet, la correction d'une faute d'orthographe dans le nom d'une propriété et c'est tout un logiciel qui peut se mettre à boguer si ces changements ne sont pas répercutés dans les chaînes passées à PropertyChanged ! Disposer d'un moyen de contrôle est ainsi un avantage, une garantie contre ce type de bogue. Comme ce contrôle nécessite l'utilisation de la réflexion, procédé coûteux, il n'est actif qu'en mode Debug. Quand le logiciel est compilé en mode Release les contrôles sont ignorés grâce à l'attribut [Conditional("DEBUG")] qui décore la méthode de vérification.

Faisons le point !

Nous venons de faire le tour de la librairie MVVM Light en partant de l'angle de vue de la pattern MVVM et de ce qu'elle implique.

Cette promenade nous a amené à évoquer :

- La séparation de l'interface utilisateur et du code fonctionnel
- La gestion du système de commande
- La gestion du système de messagerie
- Les contraintes des Modèles de Vue

En face de chacun de ces problèmes concrets posés par la mise en œuvre de la pattern MVVM nous avons vu quels outils la librairie MVVM Light mettait à notre disposition :

- Inversion de contrôle et localisateur de services
- RelayCommand, le behavior EventToCommand
- Messenger et les classes de messages spécialisées ou génériques
- La classe ViewModelBase

Arrivé ici vous devez commencer à avoir une bonne compréhension globale de ce que MVVM Light permet et comment cette librairie répond aux problèmes pratiques de la mise en œuvre de MVVM sous WPF et Silverlight.

Mais tout cela est « global » justement. Comme une image figée ne pourra jamais rendre la complexité des pas d'une danseuse, nous avons des photos mais il nous manque la vidéo et le son pour apprécier vraiment le ballet !

N'ayez crainte, je ne vais pas enfiler un tutu et esquisser quelques pas sur le Lac des Cygnes… Je ne veux aucun mal à Tchaïkovski et encore moins à mes lecteurs (et je n'ai pas non plus un goût particulier pour les tutus !)

En revanche nous allons maintenant aborder la partie pratique de cet article en voyant comment chaque outil de MVVM Light s'insère dans la mise en place d'une application respectant la pattern MVVM.

Je vous conseille maintenant, si ce n'est pas déjà fait, de télécharger et d'installer MVVM Light, sinon vous ne pourrez pas reproduire les exemples ni même faire tourner les solutions livrées avec l'article…

MVVM LIGHT EN PRATIQUE

Il est temps de passer le tutu et d'envoyer la musique… Non ! Ne vous sauvez pas, je plaisante ☺

Mais le moment est venu de faire fonctionner tout cela !

La création d'un projet MVVM Light

Un projet MVVM Light est un projet « normal », la seule différence réside dans le fait que les templates fournis avec la librairie contiennent une mise en place bien pratique. Il est tout à fait possible de se passer des templates MVVM Light de même qu'il est possible d'ajouter MVVM Light à un projet existant. Bien entendu dans ce cas il faudra certainement *refactorer* un peu le code. Il est donc bien plus pratique d'utiliser d'emblée les templates MVVM Light pour tout nouveau projet !

MVVM installe des templates pour toutes les saveurs de Xaml :

Silverlight

Dans la version actuelle, MVVM supporte directement Silverlight 3 et 4 et propose un template pour chacun.

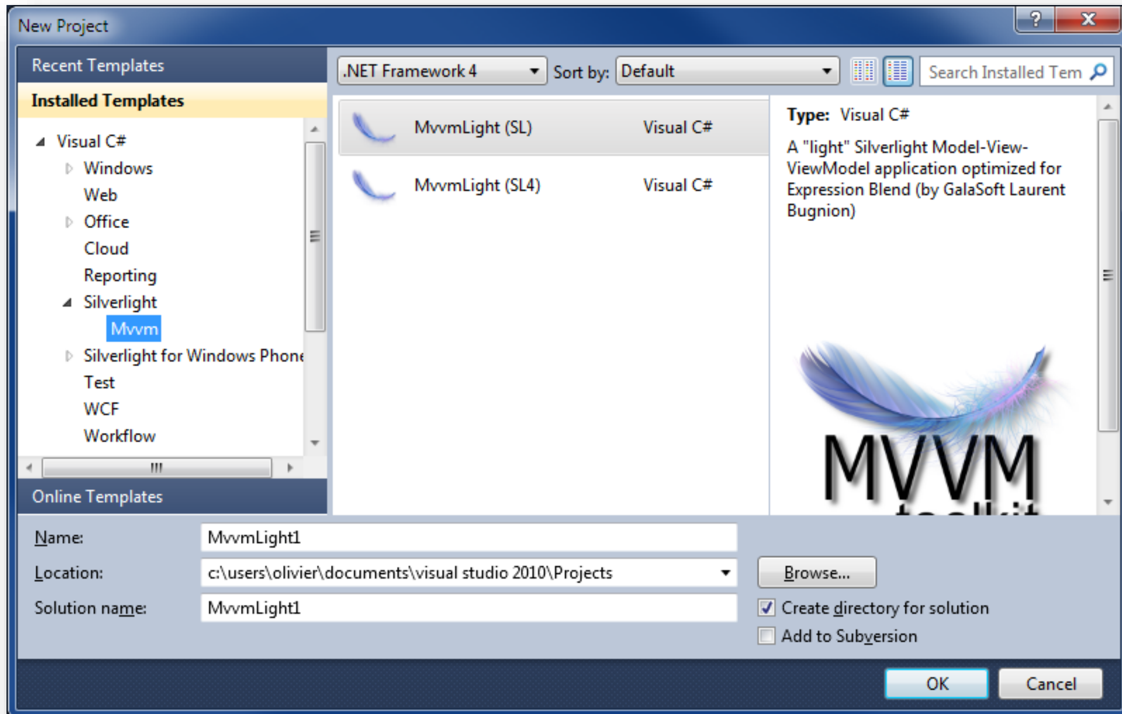


Figure 43 - Templates Silverlight

Silverlight for Windows Phone

Et oui… MVVM Light est utilisable aussi pour les applications Windows Phone 7. Rappelez-vous que la librairie ne pèse que 25Ko environ, ce qui n'est rien face au moindre PNG utilisé par l'application. Il n'y a donc aucune raison de se priver des services de MVVM Light et de la pattern MVVM sous Windows Phone 7.

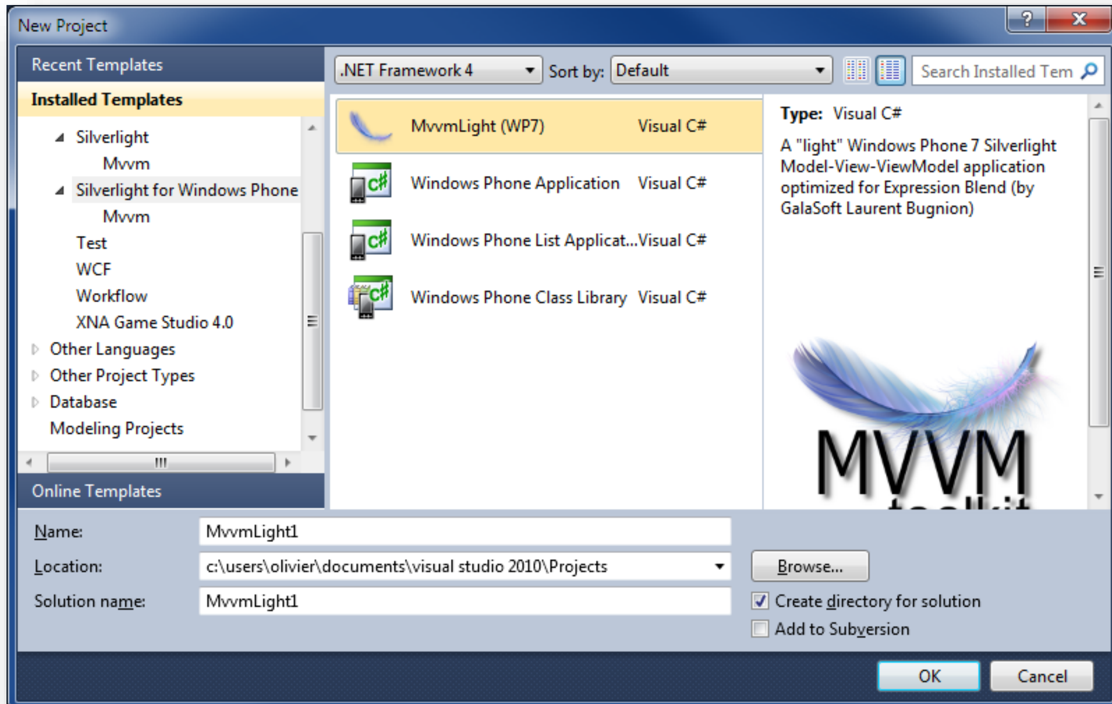


Figure 44 - Les templates pour Windows Phone

WPF

Bien entendu on trouve aussi des templates pour WPF 3.x et WPF 4.

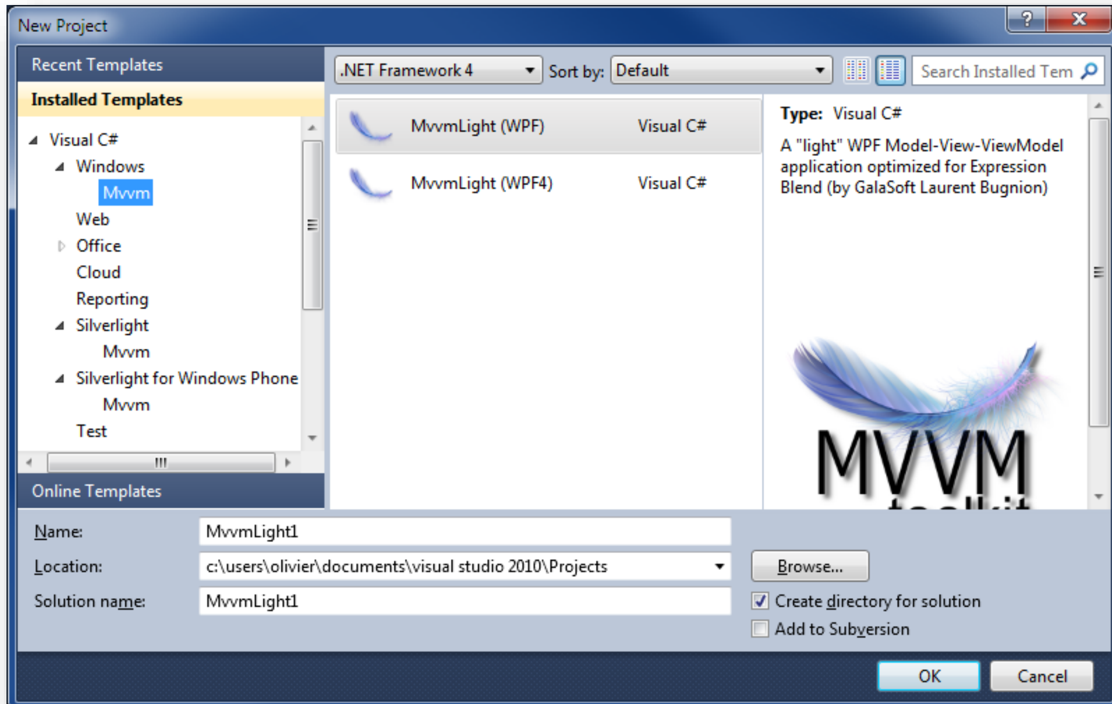


Figure 45 - Les templates pour WPF

Expression Blend

MVVM Light n'oublie pas Expression Blend qui se trouve doté, lui aussi, de templates pour les projets Silverlight et WPF.

On notera que pour Windows Phone, actuellement toujours en bêta, c'est une version spéciale de Blend 4 qu'il faut utiliser, du coup Blend 4 « normal » ne contient pas de templates et il semble que Laurent Bugnion n'a pas trouvé le moyen d'installer des templates dans le Blend 4 spécial Windows Phone. Dans l'attente que ce petit problème soit réglé (car il le sera) il suffit de créer les projets Phone 7 en utilisant le template MVVM Light depuis Visual Studio puis d'ouvrir la solution sous Blend 4 for Windows Phone, et le tour est joué…

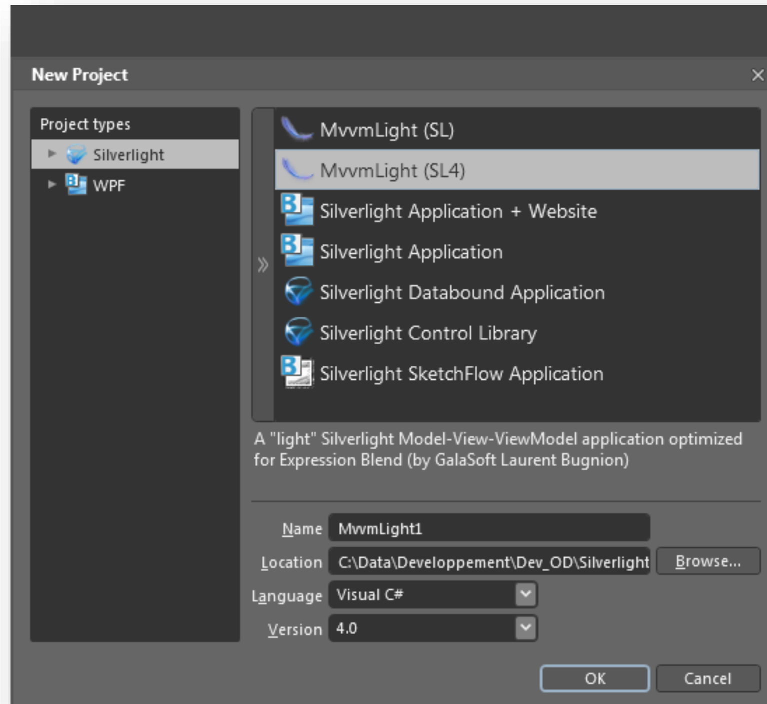


Figure 46 - Les templates Silverlight pour Blend

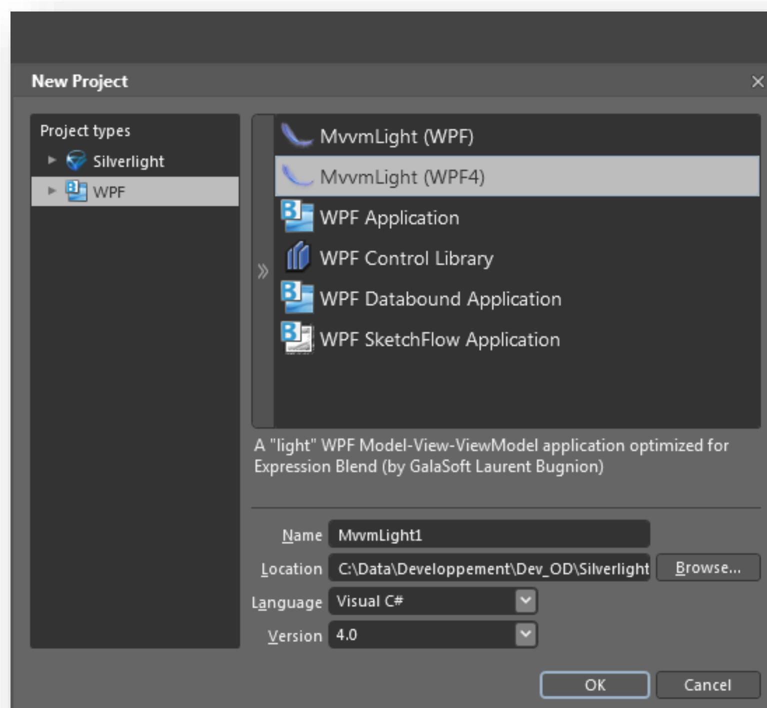


Figure 47 - Les templates WPF pour Blend

Le contenu d'un projet MVVM Light

Ayant une petite préférence pour Silverlight, c'est un projet de ce type qui va nous servir d'exemple (il n'y a pas vraiment de différence avec le template WPF). De même, lorsqu'il s'agit de code ou de principes simples, je vais rester sous Visual Studio même si je préfère (et vous conseille fortement) d'utiliser Expression Blend avec Silverlight et WPF. Mais comme je sais que tout le monde ne possède pas Blend, le choix de VS semble plus évident pour qu'un maximum de lecteurs puisse s'y retrouver facilement.

Le projet exemple s'appelle « Exemple1 » en voici le contenu une fois le template MVVM Light sélectionné et validé :

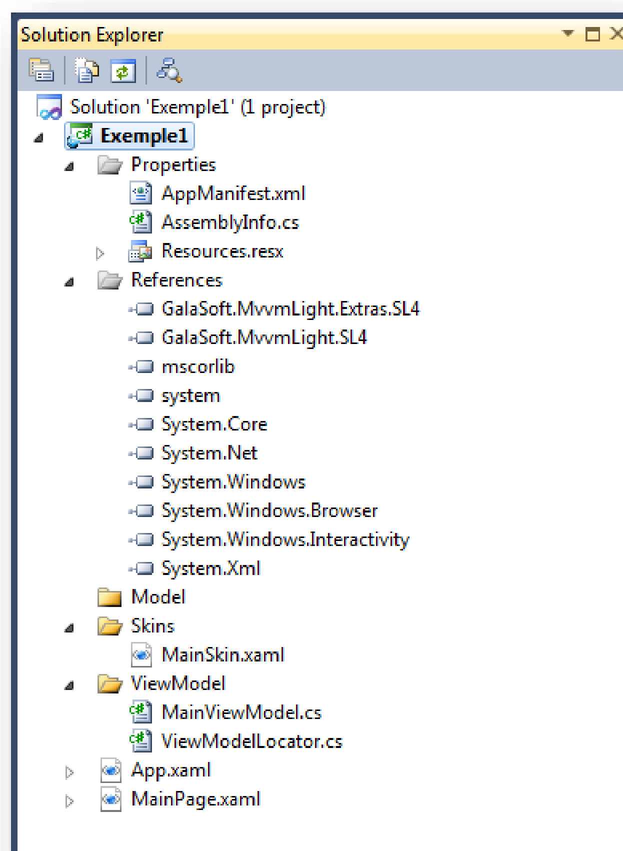


Figure 48 - Squelette d'un template de projet MVVM Light

Comme on peut le voir dans la capture ci-dessus, un projet MVVM Light ressemble beaucoup à un projet standard aux différences suivantes :

- Les références comportent un lien vers les deux Dlls de MVVM Light
- Des sous-répertoires sont ajoutés :
    - Model, qui est vide, et qui contiendra les modèles (données)

- Skins, qui contient MainSkin.xaml, répertoire dédié aux styles et templates visuels (à créer et à modifier de préférence sous Blend)
- ViewModel près à recevoir vos Modèles de Vue et qui contient MainViewModel.cs, le Modèle de Vue de la page principale et ViewModelLocator.cs, le localisateur de services.

- Une première page Mainpage.xaml reliée à MainViewModel via le localisateur de services.

C'est peu de chose, mais il faut dire que la librairie est justement étudiée pour être légère et impliquer le moins de contraintes possibles.

La compilation du projet donne le résultat suivant :

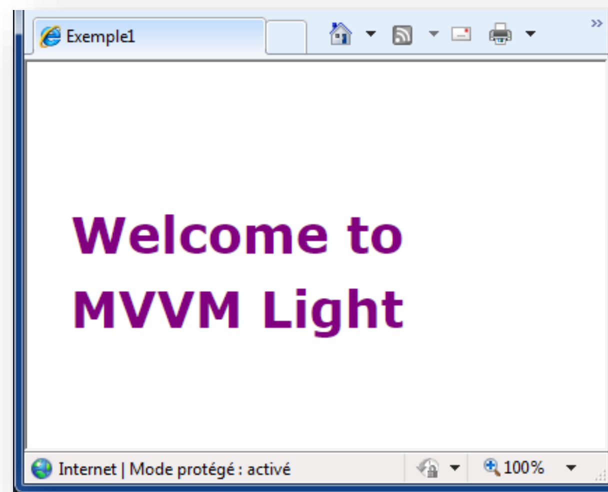


Figure 49 - Compilation du template (sous Silverlight)

Magnifique ☺

On ne se moque pas car il s'agit déjà d'une application MVVM ! Décortiquons-là justement…

Le répertoire pour les modèles est vide, on va donc passer très vite sur le sujet que nous aborderons plus loin. Le répertoire Skins contient un dictionnaire de ressource qui est vide, ne reste plus qu'à y ajouter vos styles, passons aussi, donc.

Commençons plutôt par la page principale pour démêler l'écheveau MVVM.

MainPage, le début du voyage

Sous Visual Studio la page se présente comme suit :

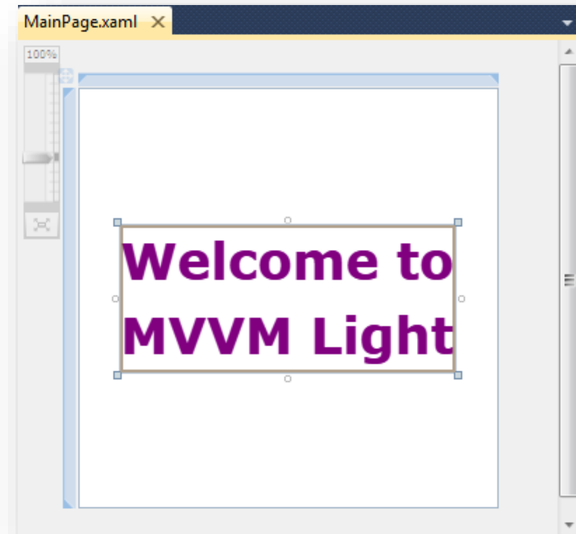


Figure 50 - MainPage sous Visual Studio

Comme on peut le noter il n'y a rien d'autre qu'un TextBlock affichant le même texte que celui que nous avons pu voir lors de l'exécution du projet.

Toutefois, à y regarder de plus près nous voyons que la propriété Text de ce TextBlock est déjà bindée à « quelque chose », ouvrons l'éditeur de data binding :

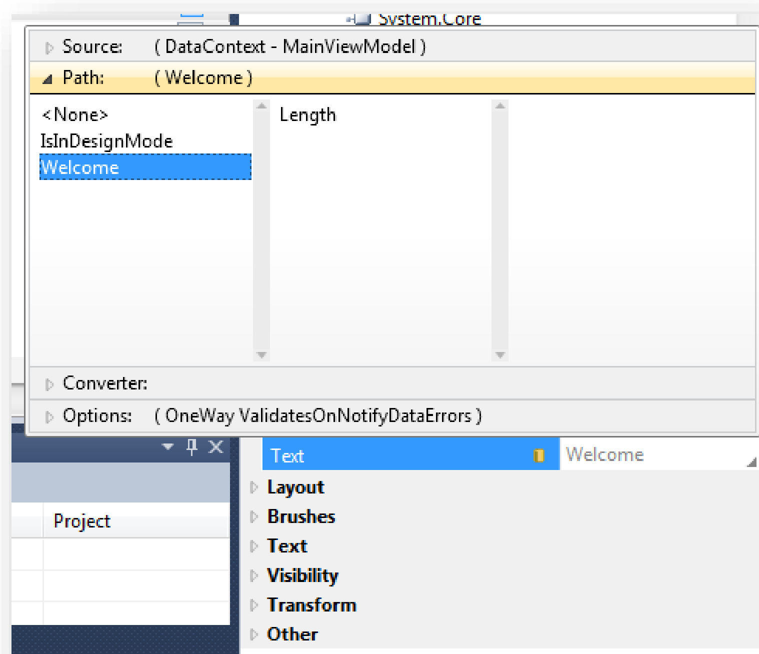


Figure 51 - Le binding de la zone de titre

L’éditeur de data binding de VS révèle ce à quoi est liée la propriété : elle est connectée au chemin (Path) « Welcome » de la source « MainViewModel » vue grâce au DataContext de la page.

Le code Xaml du TextBlock nous montre tout cela :

```
<TextBlock FontSize="36"
           FontWeight="Bold"
           Foreground="Purple"
           Text="{Binding Welcome}"
           VerticalAlignment="Center"
           HorizontalAlignment="Center"
           TextWrapping="Wrap" />
```

En effet, la propriété Text est bindée à Welcome. D’où vient cette variable ? …du DataContext, nous venons le voir (il faut suivre hein !). Mais un DataContext c’est vide… du moins tant qu’on ne le fait pas pointer sur quelque chose. Et quel est le code derrière ce DataContext ? Celui de MainViewModel (on vient de le voir juste sur l’image précédente… on ne s’endort pas !).

Donc regardons le DataContext de la page :

```
<UserControl x:Class="Exemple1.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"
        mc:Ignorable="d"
        Height="300"
        Width="300"
        DataContext="{Binding Main, Source={StaticResource Locator}}">
```

Le DataContext du UserControl est ainsi lui-même bindé à quelque chose d’autre. Une ressource statique s’appelant Locator et à la propriété Main de celui-ci.

Mais qui est donc ce Locator et de quoi est faite sa propriété Main ?

De la MainPage à App.xaml

Puisqu’il s’agit d’une ressource statique et qu’elle n’est pas déclarée dans l’objet en cours, c’est qu’elle est globale, donc déclarée dans App.xaml. Ouvrons ce fichier :

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Exemple1.App"
```

```
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:vm="clr-namespace:Exemple1.ViewModel"
        mc:Ignorable="d">
    <Application.Resources>
        <!--Global View Model Locator-->
        <vm:ViewModelLocator x:Key="Locator"
                     d:IsDataSource="True" />
    </Application.Resources>
</Application>
```

C'est là, en effet. La section est même commentée « *Global View Model Locator* ». C'est ici, dans le dictionnaire de ressource principale de l'application qu'est créée une instance de ViewModelLocaltor à laquelle est attribuée la clé « *Locator* ». Le localisateur de services devient donc une ressource globale de l'application, visible et accessible depuis toutes les Vues sous le nom de « *Locator* ».

Tels des saumons excités par la fraie, nous remontons la rivière bouillonnante des binding et des ressources…

Il ne nous reste plus qu'à comprendre quelle est cette classe « ViewModelLocator » pour continuer notre course vers la source.

De App.xaml au Localisateur de Services

Des saumons excités par la fraie, je veux bien, mais pas amnésiques pour autant … Ce nom là nous l'avons vu il n'y a pas longtemps. En effet, parmi les choses ajoutées par le template MVVM Light au projet se trouve justement et comme par un heureux hasard une unité de code s'appelant ViewModelLocator.cs située dans le sous-répertoire ViewModel.

Ouvrons ce fichier…

Comme le fichier est abondamment commenté je ne fais pas le publier en entier ici (vous pouvez l'ouvrir sur VS ou Blend, ça sera bien plus pratique !). Mais on y découvre la déclaration de la classe ViewModelLocator.

Que contient cette classe ?

Pour l'instant peu de choses :

```
private static MainViewModel _main;
```

Une variable statique de type MainViewModel, le Modèle de Vue, c'est son nom que nous voyions apparaître dans l'éditeur de binding de VS. Cette variable se retrouve exposée sous la forme de deux propriétés :

```
public static MainViewModel MainStatic
{
   get
   {
      if (_main == null)
      {
         CreateMain();
      }

      return _main;
   }
}
```

et

```
public MainViewModel Main
{
   get
   {
      return MainStatic;
   }
}
```

Pourquoi le champ _main se trouve-t-il ainsi exposé deux fois ?

La raisons est simple : la propriété MainStatic est le reflet du champ _main qui est static aussi. MainStatic donne ainsi naturellement accès de façon statique au champ qui l'est lui aussi.

Mais le binding ne fonctionne pas sur les propriétés statiques, il faut donc une propriété d'instance pour créer un binding. D'où la seconde déclaration qui sera visible au travers de l'instance du localisateur de service « *Locator* » créée dans App.Xaml.

Pour être plus exact, sous WPF le data binding sur une propriété statique fonctionne, mais pas sous Silverlight. De plus, si on veut conserver la blendabilité, même avec WPF il est conseillé d'utiliser une propriété d'instance. C'est pourquoi MVVM Light a opté pour cette façon de faire.

Il faut noter que la stratégie elle-même adoptée par MVVM Light n'est pas un absolue. Le fait que le champ _main soit statique est assez logique puisque la fiche principale vivra tout le temps de l'application et qu'il ne peut y avoir qu'une fiche principale dans l'application. Toutefois appliquer la même logique à tous les autres Modèles de Vue n'est qu'une possibilité. Dans une grosse application ou dans une application choisissant de charger des Dlls dynamiquement il sera peut-être plus habile de pratique autrement.

On notera que la propriété statique n'est réellement instanciée que lorsqu'elle est utilisée. Donc tous les Modèles de Vue ne sont pas chargés en mémoire à la création de l'application, juste quand ils sont référencés par une autre instance.

D'ailleurs on trouve dans le localisateur de services d'autres méthodes comme :

```
public static void ClearMain()
{
    _main.Cleanup();
    _main = null;
}
```

Cette méthode, qui est censée exister pour chaque Modèle de Vue, effectue justement le ménage et doit être appelée dès qu'une Vue est devenue inutile et que son Modèle de Vue l'est devenu aussi. Ces deux conditions pouvant être découplées : par exemple lorsque l'application change de page, elle peut « perdre » la Vue, cela fait de la place en mémoire, mais elle peut décider de conserver le Modèle de Vue. Si l'utilisateur retourne sur la page, la nouvelle Vue créée, en se liant au Modèle de Vue qui existe déjà, sera affichée dans le même état (à quelques détails près que je vous laisse imaginer, mais qui peuvent se régler en stockant justement des informations d'état dans le Modèle de Vue).

```
public static void CreateMain()
{
    if (_main == null)
    {
        _main = new MainViewModel();
    }
}
```

De la même façon si on veut s'assurer qu'un Modèle de Vue existe bien avant de s'en servir (par code par exemple) on peut appeler CreateMain. Chaque Modèle de Vue est là aussi censé posséder une méthode identique. Mais l'accès à la propriété (d'instance ou statique) du localisateur de services créée l'instance du Modèle de Vue si elle n'existe pas. On peut se

demander alors quel est l'intérêt de cette méthode. En réalité elle n'a de sens que pour la page principale de l'application. La méthode est d'ailleurs appelée dans le constructeur de la classe ViewModelLocator. Il s'agit là certainement de s'assurer que le Modèle de Vue de la page principale existe au plus tôt, dès que l'instance du *locator* est créée dans App.xaml. Il ne me semble donc guère intéressant de reproduire une telle méthode pour les autres Modèles de Vue.

Enfin, le localisateur de services expose

```
public static void Cleanup()
{
   ClearMain();
}
```

Une méthode de nettoyage qui ne contient pour le moment que l'appel à ClearMain mais qui devra être agrémentée de tous les Clearxxx des Modèles de Vue qu'on ajoutera au projet. Le « clear » de chaque Modèle de Vue s'assure notamment que les abonnements à la messagerie sont révoqués, ce qui évite d'éventuels *memory leaks* (perte de mémoire).

MVVM Light est simple, peut-être simpliste dans cette gestion du localisateur de services. Avec ce procédé il est par exemple impossible d'avoir deux fiches de détail « article » en même temps. Les propriétés Modèles de Vue étant statiques on aurait deux fois le même affichage au lieu d'avoir celui de deux articles différents…

Sous Silverlight, pour le Web ou pour Phone 7, cela n'est pas très grave car il est rare d'ouvrir des fenêtres les unes sur les autres. En revanche pour une application WPF cela peut devenir plus gênant car ce cas d'utilisation et loin d'être rare. Il existe d'ailleurs d'autres modèles d'applications, notamment ceux faisant intervenir la composition, la découverte dynamique de plugins, etc, qui ne pourraient pas se satisfaire d'une vision aussi simple du Localisateur de services.

Mais cette partie de MVVM Light est totalement découplée des autres fonctions offertes par la librairie. De fait, rien ne vous interdit, sans perdre les autres avantages de MVVM Light, d'écrire votre propre Localisateur de services, voire d'utiliser celui d'un autre framework. De nombreux développeur penchent aujourd'hui vers Unity, un petit Framework Microsoft spécialisé dans l'inversion de contrôle et utilisant la variante de l'injection de dépendances plutôt que celle du localisateur de services.

C'est ici que les choses se compliquent méthodologiquement. Faire cohabiter plusieurs Framework et ne pas se retrouver avec un sac de nœuds dont on ne maîtrise plus les effets de bord réclame de grandes compétences et un temps de réflexion et de test non nul…

Donc sauf à disposer de telles compétences et du temps pour vous assurer que tout marchera bien ensemble, le plus sage est de rester avec quelque chose de simple, MVVM Light par exemple, et de vous en contenter quitte à écrire un peu de code pour améliorer telle ou telle partie. Garder la maîtrise de son ouvrage est autrement plus important que vouloir associer d'énormes briques sans être sûr de pouvoir en supporter le poids… Rappelez-vous de la fable de La Fontaine sur la grenouille qui voulait se faire plus grosse que le bœuf. Ca finit mal pour la grenouille…

Fin du premier voyage

De la fiche principale, en passant par son DataContext, nous sommes remontés à la ressource ayant pour clé Locator, ressource définie dans App.xaml comme une instance du localisateur de services. Dans ce dernier nous avons découvert les propriétés qui pointent les instances des Modèles de Vue.

Nous avons ainsi pu comprendre comment le découplage Vue / Modèle de Vue propre à MVVM est mis en œuvre avec MVVM Light, notamment par le biais d'une inversion de contrôle utilisant la pattern *Service Locator*.

C'est grâce à l'ensemble des mécanismes que nous venons de voir que la propriété Text du TextBlock de la MainPage peut afficher un texte qui n'apparaît pour l'instant nulle part mais qu'on sait provenir du Modèle de Vue, de la propriété Welcome plus exactement, au travers du Locator connecté au DataContext de MainPage et qui aiguille le contexte vers le bon Modèle de Vue.

Ce n'est qu'un premier voyage, d'autres choses restent à découvrir. Mais les saumons sont arrivés à la source, ne restent plus qu'à trouver un partenaire, le Modèle de Vue.

Le Modèle de Vue

Dans notre voyage depuis la MainPage nous avons croisé différents éléments qui composent le template de projet MVVM Light. Nous sommes arrivés à un premier palier, là se trouve une porte. Derrière celle-ci se cache le Modèle de Vue MainViewModel qui est associé à MainPage. Poussons la porte…

…et nous découvrons un code très court (j'en supprime certains commentaires pour une meilleure lisibilité) :

```
public class MainViewModel : ViewModelBase
  {
     public string Welcome
     {
        get
        {
           return "Welcome to MVVM Light";
        }
     }

     public MainViewModel()
     {
        if (IsInDesignMode)
        {
           // Code runs in Blend --> create design time data.
        }
        else
        {
           // Code runs "for real"
        }
     }

     ////public override void Cleanup()
     ////{
     ////    // Clean up if needed

     ////    base.Cleanup();
     ////}
  }
}
```

La classe MainViewModel ne contient presque rien, juste une propriété « Welcome » qui contient justement le texte que nous avons vu s'afficher à l'exécution. C'est la propriété qui est bindée à la propriété Text du TextBlock de la MainPage.

Pour le reste tout est virtuellement en place : la possibilité de surcharger la méthode Cleanup() si nécessaire, la possibilité d'agir différemment selon qu'on se trouve en mode conception ou en mode exécution du projet.

Rien dans cette classe n'est vraiment spécial, sauf son héritage : ViewModelBase.

C'est par ce dernier que le Modèle de Vue se voit doté de certaines capacités dont pour ainsi dire aucune n'est d'ailleurs exploitée dans la classe MainViewModel dans son état actuel.

Pour le lecteur distrait je rappelle que la classe ViewModelBase a été présentée page 185.

On comprend maintenant qu'il suffit d'ajouter des propriétés à cette classe puis de les binder à d'autres propriétés de la MainPage pour bénéficier de données réelles ou simulées, même en conception.

Fin du second voyage

Créer une application suivant MVVM ne se limite toutefois pas à créer des propriétés dans des Modèles de Vue et à les lier à des Vues !

Mais ce voyage en deux parties n'était qu'une mise en bouche, un apéritif pour s'ouvrir l'appétit !

N'oublions pas que notre but dans cette section était de faire le tour de ce qu'il y a à l'intérieur d'un template de projet MVVM Light. Ce tour d'horizon est terminé, nous avons vu chaque classe importante, chaque mécanisme utilisé.

Néanmoins un « *hello world* », aussi intéressant soit-il, est à mille lieux d'une application réelle, on le sait bien. Et notre template, une fois compilé n'est jamais qu'un « *hello world* » sophistiqué.

Il nous reste à voir *comment utiliser en pratique le potentiel de MVVM Light* et du pattern MVVM pour concevoir des applications. C'est ce que nous verrons plus loin avec des exemples.

Gérer les changements de valeurs

Le template MVVM Light propose une page principale reliée à son Modèle de Vue, dans ce dernier une seule propriété est définie, et encore est-elle en lecture seule, juste un message de bienvenue.

Dans la réalité les propriétés sont souvent des valeurs disponibles aussi en écriture. De même, le Modèle de Vue, réagissant à certains événements ou messages, à des changements d'état dans les données (le Modèle), va-t-il lui-même modifier la valeur de certaines propriétés, ce que l'interface utilisateur doit refléter.

Nous allons faire quelques modifications simples au Modèle de Vue existant.

- Nous allons supprimer la propriété la propriété Welcome
- Ajouter une propriété Title (titre)
- Ajouter les propriétés CurrentTime (heure courante) et CurrentDate
- Modifier la Vue pour qu'elle affiche notre titre et qu'elle prenne aussi en charge l'affichage de l'heure.
- Accessoirement donner une taille de 800x600 à la page principale.

Nota : Dans le code source fourni avec l'article il s'agit de « Exemple2 »

Créer les propriétés

Ajouter des propriétés c'est facile, la preuve, voici celles que je viens d'ajouter à MainViewModel, le Modèle de Vue de la page principale MainPage :

```
public string Title
{
   get { return "Exemple 2 - MVVM Light en pratique"; }
}

public string CurrentDate
{
   get { return DateTime.Now.ToShortDateString(); }
}

public string CurrentTime
{
   get { return DateTime.Now.ToShortTimeString(); }
}
```

Rien de bien fantastique donc. J'ai aussi créé les trois TextBlock qui reçoivent ces valeurs dans l'écran principal.

Pour le binding des Textblock voici comment cela se passe sous Blend :

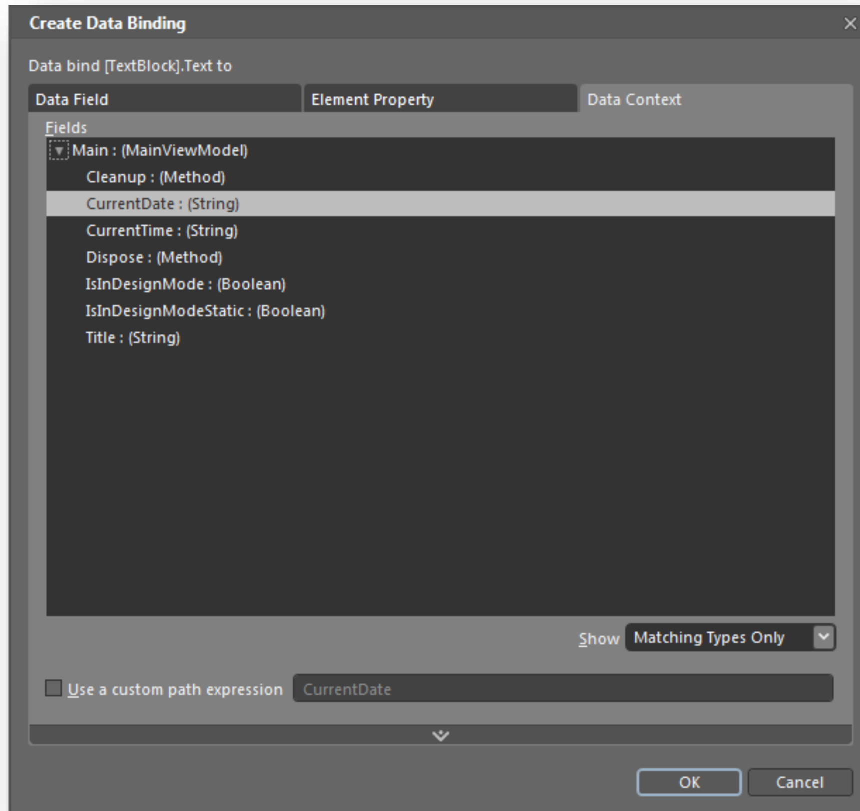


Figure 52 - Binding sous Blend

En affichant l'onglet *Data Context* on voit toutes les propriétés mises en place par le Modèle de Vue. Il suffit de choisir celle qu'on veut lier. On notera que sous Blend ou sous Visual Studio il est nécessaire de construire le projet pour que les modifications du Modèle de Vue soient visibles.

> *Attention : Blend accepte parfaitement que le DataContext de la vue soit déclaré soit dans la balise d'ouverture du UserControl, soit sous la forme d'une balise de ressources un peu plus loin dans le code Xaml. Hélas le concepteur visuel de Visual Studio est plus « chatouilleux » et lorsqu'on utilise la seconde forme le dialogue de binding ne marche pas correctement. Si vous tombez sur ce problème, au lieu de vous arracher les cheveux, déplacer l'initialisation du DataContext dans la balise d'ouverture du UserControl et tout marchera (sous VS et sous Blend). Une astuce à garder sous le coude !*

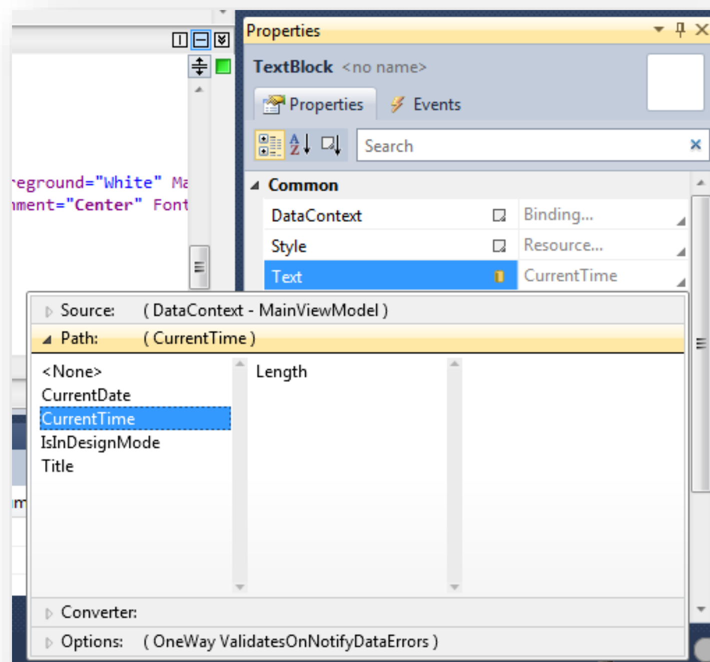


Figure 53 - binding sous Visual Studio

Lorsqu'on exécute l'application on obtient un affichage de ce type :

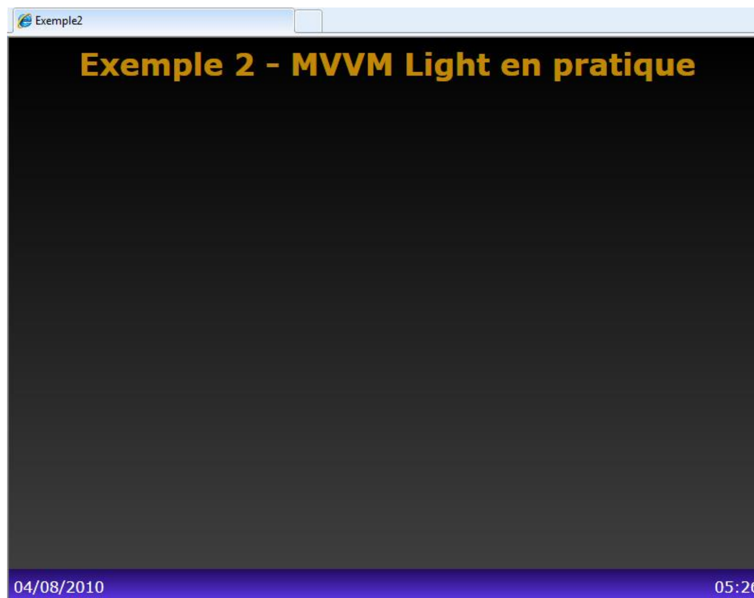


Figure 54 - Application exemple 2 - phase 1

Peu importe la mise en page, ce qui nous importe ce sont les trois zones de texte. En haut le titre, en bas à gauche la date, en bas à droite, l'heure.

Avertir l'interface du changement

Tout est parfait sauf que… L'heure ne se met pas à jour. Telle qu'affichée il faut attendre au moins une minute pour s'en apercevoir, nous allons ajouter les secondes à l'affichage pour mieux se rendre compte, mais croyez-moi, ça ne changera rien.

En effet, telle qu'elle est définie dans le Modèle de Vue, la propriété CurrentTime est une propriété en lecture seule, donc elle ne possède pas de *setter*, donc à aucun moment PropertyChanged n'est invoqué par quelque procédé que ce soit.

De fait, le binding existant entre cette propriété et la zone d'affichage n'est jamais prévenu qu'il faut rafraichir l'affichage.

Dans un cas comme celui-là il nous faut un autre moyen pour prévenir le binding que les données ont changé. Ici, un timer va être utilisé.

Pour cela il ne faut pas créer n'importe quel timer. Un System.Threading.Timer tournera dans un thread différent de celui de l'interface ce qui nous obligera à utiliser Dispatcher.Invoke pour créer la mise à jour de l'interface sinon on risque fort d'obtenir une exception. Cela complique un peu trop les choses d'autant qu'il existe un autre timer dans System.Windows.Threading. Il s'agit de DispatcherTimer. Ce dernier a la particularité de se placer dans la boucle de job du thread de l'UI, nous ne risquons donc pas de problèmes (en

revanche ce type de timer n'est pas très précis et ne garantit qu'un intervalle de temps minimum sans maximum, si la machine est occupée le timer n'aura pas la main).

Ensuite il est inutile de faire marcher le timer lorsque le projet est en mode conception. MVVM Light nous offre un point d'entrée intéressant dans le constructeur du Modèle de Vue avec un « if » permettant de savoir dans quel mode l'application se trouve. C'est donc naturellement dans le seconde partie (le « else »), c'est-à-dire quand le projet est exécuté, que nous placerons le code de création et d'activation du timer.

Enfin, dans le gestionnaire de l'événement, nous nous contenterons d'appeler la méthode RaisePropertyChanged du Modèle de Vue, méthode héritée de ViewModelBase qui effectuera pour nous le reste du travail de notification.

Assurer le nettoyage

Une dernière chose, nous allons supprimer les commentaires de la surcharge de la méthode Cleanup() car si le timer est non nul, nous appellerons sa méthode Stop(). Ainsi, si l'application demande au Modèle de Vue de faire le ménage, le timer sera arrêté et supprimé. Dans le cas présent, s'agissant de la fiche principale de l'application cela n'aura aucun impact réel, mais dans un autre Modèle de Vue cela peut en avoir beaucoup, donc plutôt que de réfléchir et faire des bêtises autant utiliser des méthodes de développement identiques qui fonctionnent dans toutes les situations.

Le code

La création du timer :

```
private DispatcherTimer timer;

public string CurrentTime
{
   get { return DateTime.Now.ToString("HH:mm:ss"); }
}
```

L'activation du timer uniquement au runtime :

```
public MainViewModel()
{
   if (IsInDesignMode)
   {
      // Code runs in Blend --> create design time data.
   }
   else
   {
      timer = new DispatcherTimer {Interval = TimeSpan.FromSeconds(1)};
```

```
            timer.Tick += timer_Tick;
            timer.Start();
        }
    }

    void timer_Tick(object sender, EventArgs e)
    {
        RaisePropertyChanged("CurrentTime");
    }
```

La méthode de nettoyage :

```
    public override void Cleanup()
    {
        // Clean up if needed
        if (timer!=null)
        {
            timer.Stop();
            timer.Tick -= timer_Tick;
            timer = null;
        }
        base.Cleanup();
    }
```

Méthodes anonymes, délégués et gestionnaires d'événement

Certains lecteurs se demanderont pourquoi je n'ai pas utilisé un *delegate* anonyme ou une expression lambda pour définir le gestionnaire d'événement de Tick. En effet, il serait bien plus élégant, à première vue, d'écrire :

```
timer.Tick += delegate { RaisePropertyChanged("CurrentTime"); };
```

ou même :

```
timer.Tick += (s, e) => RaisePropertyChanged("CurrentTime");
```

Le mieux est souvent l'ennemi du bien, et vouloir absolument utiliser toutes les subtilités d'un langage implique d'avoir une bonne maîtrise de celui-ci, sinon gare à tout ce qu'on ignore et où se cachera un problème bien difficile à déboguer !

Ici nous voulons être sûr de faire le nettoyage du timer, de ce fait, le stopper n'est pas suffisant puisqu'il conserve une référence vers le code du gestionnaire d'événement du Tick et que ce code référence une méthode de l'instance du Modèle de Vue (de l'instance puisque RaisePropertyChanged n'est pas une méthode statique, elle est protected virtual).

Si nous utilisons un délégué anonyme ou une expression lambda nous ne serons pas en mesure de supprimer le gestionnaire d'événement. Certains pensent qu'en faisant « Tick -= …delegate/expression… » cela va fonctionner (en recopiant le code du délégué ou de l'expression lambda). Cela compile oui. Mais en réalité chaque délégué et chaque expression lambda sont uniques, le compilateur ne s'amuse pas à comparer le code de chacun pour voir s'ils sont équivalents ou non, cela impliquerait de contrôler tout l'arbre de code qui ici est trivial mais qui peut fort bien être plus complexe. Un tel comportement magique n'est pas implémenté. De fait le code compile mais il ne supprimera pas le gestionnaire ajouté à Tick, il supprimer un nouveau gestionnaire qui n'a jamais été enregistré, donc il ne fera rien, et l'ancien gestionnaire restera sagement en mémoire !

Du coup il faut conserver une référence vers le délégué ou l'expression lambda, ce qui est facile à faire en déclarant une variable.

Mais arriver à ce stade de complication juste pour éviter d'écrire un gestionnaire d'événement traditionnel est totalement surfait et inutile. D'où l'écriture qui a été retenue qui permet simplement de supprimer le gestionnaire de Tick, sans manière, et de façon sûre.

Le cas de la date

L'heure oui, mais on se dit que la date ça ne change pas en cours de journée, du coup à quoi bon la rafraichir… Le bug de l'an 2000 a existé parce que des programmeurs ont raisonné de cette façon, « l'an 2000 c'est loin, gagnons deux octets et codons l'année sur 2 chiffres… »

Mais celui qui travaillera tard le soir sur votre application ? À 0h10 du matin il verra toujours la date de la veille ?

Il est donc indispensable (je dis bien indispensable) de coder un rafraichissement de la date comme on le fait pour l'heure, même dans une application modeste.

Ce genre de détail ne doit pas vous échapper, ne lésinez jamais sur ces aspects trop souvent jugés annexes. C'est ce qui fait la différence entre du code professionnel et du code de programmeur à la petite semaine.

Faisons le point

Nous avons vu ici comme ajouter des propriétés dans le Modèle de Vue et comment lier ces propriétés à celles des éléments d'interface de la Vue via le DataContext de cette dernière (le code Xaml des bindings est consultable dans les sources des projets exemples fournis avec l'article).

Nous avons vu aussi à quel point il était important de gérer les RaisePropertyChanged convenablement pour assurer le rafraichissement de la Vue.

Bien entendu nous ne pouvons ici faire le tour de tous les types possibles de propriétés. Le principe reste ensuite rigoureusement le même que celui que nous avons étudié, qu'il s'agisse de propriété en lecture seule, en lecture et écriture, de string, de dates, ou de collection d'objets complexes.

Gérer la communication

Attachons-nous maintenant à la compréhension du système de messagerie.

Les messages de notification de changement de valeur

Je pourrais vous proposer d'animer le changement de date, mais il faudrait attendre 24h pour voir une fois l'effet… Alors même si animer l'heure toute les secondes n'a pas un intérêt énorme cela va nous permettre de voir le phénomène étudié.

Pourquoi dans cette section sur les communications ajouter une animation sur l'heure ? Parce que, si vous l'avez remarqué, la classe TextBlock ne possède pas de TextChanged. Du coup toutes les techniques simples et habituelles pour ajouter un effet sur le changement de valeur du texte (qui sortiraient d'ailleurs du cadre de cet article) ne sont pas applicables. Aucun Trigger ne peut être utilisé, aucun *behavior* (de base en tout cas).

C'est ainsi que cet exemple un peu stupide va en réalité se révéler riche en enseignement !

Puisque par construction, TextBlock ne gère pas d'événement de changement de valeur, et puisque la Vue a besoin d'être prévenue de ce changement, c'est le Modèle de Vue qui va être obligé de fournir à la Vue ce dont elle a besoin pour fonctionner. Nous sommes donc bien en plein dans la problématique générale de MVVM.

La première chose à faire consiste à créer une petite animation sur le TextBlock de l'heure, si vous n'avez pas Blend ne vous inquiétez pas, elle se trouvera déjà dans le projet Exemple2. Cela va être minimaliste puisque les animations ne sont pas mon propos ici. L'animation s'appellera « TimeHasChangedAnim ». L'effet rendu sera plutôt celui auquel on s'attend pour un compte à rebours par exemple. L'animation consiste à faire passer immédiatement l'opacité du texte à zéro puis de le refaire monter à 100% en 400 millisecondes, ce qui laisse 60% du temps à pleine intensité. A la longue c'est un peu énervant car cela attire vraiment l'œil ! Le procédé n'est donc pas à conseiller dans le cadre d'une horloge discrète qui tourne en bas de l'écran comme dans notre démonstration !

Bref, nous avons l'animation, la vraie question est de savoir comment diable la déclencher !

Ceux qui crient « la messagerie ! la messagerie ! », c'est bien, vous avez suivi, mais quelle messagerie et par quel procédé ?

MVVM Light nous offre une classe, Messenger, offrant de base tous les services nécessaires à la création d'une messagerie. Nous aussi vu que la classe ViewModelBase servant à créer des Modèles de Vue support la méthode RaisePropertyChanged qui prend en charge INotifyPropertyChanged.

Mais à y regarder de plus près (dans le code ViewModelBase de MVVM Light) on s'aperçoit que RaisePropertyChanged existe en plusieurs variantes dont l'une appelle une méthode appelée BroadCast et que, dans ce cas, un message bien spécial est émis, PropertyChangedMessage (visible sur le schéma **Figure 42 - Les différentes classes de messages de MVVM Light**, page 180).

Il suffit donc de modifier nos appels à RaisePropertyChanged pour utiliser la variante émettant ce message puis de récupérer ce message dans la Vue.

Broadcaster les notifications de changement de valeur

La méthode qui gère l'événement Tick du timer devient la suivante :

```
void timer_Tick(object sender, EventArgs e)
     {
       RaisePropertyChanged("CurrentTime","","",true);
       RaisePropertyChanged("CurrentDate","","",true);
     }
```

Vous noterez que maintenant nous utilisons une variante à 4 paramètres de RaisePropertyChanged. Le premier ne change pas, c'est le nom de la propriété, les deux seconds représentent l'ancienne et la nouvelle valeur, le quatrième indique enfin si on souhaite ou non que BroadCast soit appelé.

Tel que fonctionne notre application seule la notification de changement de valeur a un intérêt, il est détecté par le binding entre la Vue et le Modèle de Vue et permet le rafraichissement des textes affichés par la première. Nous passons ainsi des chaînes vides pour le couple ancienne / nouvelle valeur car nous n'utilisons pas ces informations ici.

Recevoir le message dans la vue

A chaque changement des propriétés CurrentTime et CurrentDate un PropertyChanged sera déclenché (comportement par défaut de INotifyPropertyChanged) mais, en plus, un PropertyChangedMessage sera envoyé selon un autre canal, celui de Messenger, la messagerie de MVVM.

Envoyer des messages, c'est comme lancer des bouteilles à la mer. A la différence d'un Robinson Crusoé qui ne peut compter que sur la chance nous allons créer un récepteur pour notre bouteille afin d'être sûr que quelqu'un la trouvera…

Ce qui nous intéresse c'est que la Vue qui affiche l'heure et la date puisse être avertie du changement, c'est donc dans le code de cette vue que nous devons gérer le message.

Cela s'effectue en deux étapes :

1. Déclarer que nous sommes intéressés par les messages de type PropertyChangedMessage et le signifier à la messagerie ;
2. Fournir une méthode qui sera effectivement appelée lorsque le message en question se présentera et sera distribué par Messenger.

Ainsi, dans le code MainPage (MainPage.xaml.cs), nous allons commencer par nous enregistrer auprès de la messagerie :

```
public MainPage()
 {
   InitializeComponent();
   Messenger.Default.Register<PropertyChangedMessage<string>>(this, dataHasChanged);
  }
```

C'est dans le constructeur de la page que nous appelons la méthode Register de l'instance par défaut (Default) de la messagerie (Messenger). Cela faisant, nous indiquons le type de message que nous souhaitons recevoir, PropertyChangedMessage<string> (Register est une méthode générique). Dans les paramètres de la méthode nous passons le destinataire (this car c'est la page elle-même qui va gérer la réponse) ainsi qu'une méthode de signature Action<T>, le type étant PropertyChangedMessage<string>.

Notre page est maintenant enregistrée auprès de la messagerie qui sait désormais que les messages de type PropertyChangedMessage devront être routés vers la méthode dataHasChanged.

La seconde phase consiste à fournir l'Action, ici appelée dataHasChanged :

```
private void dataHasChanged(PropertyChangedMessage<string> message)
 {
   if (message.PropertyName == "CurrentTime") TimeHasChanged.Begin();
 }
```

Bien que nous émettions un message pour la date et pour l'heure, notre application ne gère pour l'instant que le message concernant l'heure. Lorsque dataHasChanged reçoit le message de changement de la propriété CurrentTime elle déclenche l'animation TimeHasChanged. Cette animation est prévue pour durer moins d'une seconde bien entendu

(puisque le message sera envoyé toutes les secondes par le Tick du timer du Modèle de Vue).

Un PDF étant ce qu'il est, je ne pourrais pas vous faire voir ici le résultat, le mieux est d'exécuter le projet Exemple2 fourni avec l'article.

Le point sur les notifications de changement de valeur

Comme nous venons de le voir les Modèles de Vue comportent déjà, par héritage de ViewModelBase, certains comportement agissant directement sur la messagerie, comme l'émission de messages PropertyChangedMessage. Cela n'est pas automatique, pour éviter de gaspiller mémoire et CPU puisque généralement la prise en compte du changement de valeur est automatique grâce au binding entre la Vue et son Modèle de Vue.

Mais nous avons vu que certains éléments d'interface courants, comme le TextBlock, ne possèdent pas d'événement PropertyChanged. De fait la Vue, bien qu'elle soit mise à jour automatiquement par le binding de son DataContext ne « sait » pas que tel ou tel texte a changé. Fonctionnellement cela ne pose généralement que peu de problèmes, mais dans un environnement comme Silverlight ou WPF où le visuel compte beaucoup, un tel changement de valeur peut être l'occasion de déclencher un son, une animation, de changer quelque chose qui relève de la Vue et non du Modèle de Vue.

L'utilisation du système de *broadcasting* des messages PropertyChangedMessage de MVVM Light peut nous apporter une solution simple à mettre en œuvre.

L'exemple choisi est forcément réducteur. Il faut donc noter qu'il est possible d'une part de recevoir ces messages de changement de valeur ailleurs que dans la Vue (il suffit de s'enregistrer comme la Vue le fait), dans un autre Modèle de Vue pourquoi pas, voire un Modèle, et, que, d'autre part, il est bien entendu possible d'émettre de tels messages en dehors de la facilité mise à notre disposition par RaisePropertyChanged qui est spécifique à ViewModelBase. Ainsi, un Modèle peut-il émettre un tel message en utilisant directement les services de Messenger. La façon de recevoir le message restant identique.

Les notifications

La messagerie de MVVM met à notre disposition d'autres types de messages que ceux portant sur le changement d'une valeur. En réalité le système de messagerie est très générique et se moque totalement du type des messages véhiculés et de leur contenu. Toutes les variantes qu'on peut voir dans l'arbre des classes de messagerie de MVVM relève plus de l'historique que d'un besoin réel. Pour tout ce qui est notification une classe générique aurait très bien pu faire l'affaire, c'est le cas d'ailleurs de GenericMessage<T>. Les messages non génériques sont issus des premières versions de MVVM Light et ne subsistent

que par souci de compatibilité ascendante, de préférence ne les utilisez pas. Quant aux spécialisations comme PropertyChangedMessage elles sont plutôt à voir comme des facilités pratiques. Rien ne vous interdit de n'utiliser que des notifications génériques. Et d'écrire vos propres types de message. Il s'agit d'une position théorique bien entendu. Dans la pratique MVVM Light propose différents types de messages, autant les utiliser, sauf peut-être les versions non génériques.

Une fois cela posé, voyons justement la classe la plus simple de notification qui descend de GenericMessage<T>, NotificationMessage<T>.

Un message de notification c'est avant tout une façon de *signaler que quelque chose vient de se passer (en général) et de le faire savoir* à tous ceux que cela intéresse.

Une notification MVVM Light est un message composé de deux composants : une string de notification, qui indique en général la nature de cette notification, et un paramètre dont le type est totalement ouvert grâce à la généricité.

Exemple de notification entre la Vue et le Modèle

Ici, nous allons créer un Modèle simple qui génère une suite d'entiers, de 1 à l'infini. Un nombre par seconde.

Ce Modèle est utilisé par le Modèle de Vue de la page principale qui expose une propriété correspondant au numéro en cours de tirage et une autre propriété de type liste qui contient les dix derniers numéros. Le modèle ne fournissant qu'un nombre par seconde, mais la Vue ayant besoin aussi d'une liste, *c'est le rôle du Modèle de Vue d'adapter les données pour satisfaire les besoins de la Vue*.

La Vue affichera donc d'une part le numéro en cours ainsi qu'une liste reflétant les derniers tirages.

Ce que nous souhaitons ici c'est que depuis la Vue l'utilisateur puisse remettre à zéro le générateur (le Modèle).

La chaîne de commandement classique passerait par une commande exposée par le Modèle de Vue, cette commande serait liée à un bouton de la Vue, et comme le Modèle de Vue a le droit de connaître les Modèles qu'il utilise il pourrait, sur l'activation de cette commande, demander au Modèle de reprendre le comptage depuis zéro. Ça, c'est la « bonne » méthode. Bonne au sens où elle utilise les procédés habituels et qu'elle produit donc un code compréhensible et maintenable.

Mais comme les commandes seront abordées plus loin, nous allons utiliser un raccourci : c'est directement la Vue qui va donner l'ordre au Modèle de se remettre à zéro.

Scandale ! Violation du pattern ! s'écrient déjà certains…

Non. Car bien entendu la Vue ne va pas appeler une méthode du Modèle. Elle va juste se contenter d'émettre un message de notification particulier sur le clic d'un bouton, c'est tout. Le Modèle écoutera ce type de notification et agira en conséquence sans même savoir d'où émane le message. MVVM est totalement respecté, même si, dans pareil cas la solution passant par une commande dans le Modèle de Vue serait certainement meilleure (tout simplement parce qu'elle serait plus « standard » vis-à-vis de MVVM).

Nous allons même ajouter un petit niveau de complexité supplémentaire : la Vue pourra indiquer à partir de quelle valeur reprendre le comptage. Tout cela sans passer par le Modèle de Vue et sans violer MVVM, juste en utilisant la messagerie.

L'application Exemple3

Il est toujours plus facile de comprendre quand on voit le résultat.

Figure 55 - Notification Vue/Modèle

Le titre, « *Exemple 3* » est issu d'une propriété Title du ViewModel. Comme je fais toujours tout en anglais, en toute logique le texte devrait être *Sample 3*, mais n'ergotons pas…

La colonne de gauche, la liste de nombres, évolue sans cesse (une fois par seconde), elle montre en permanence les dix derniers tirages. Elle est issue d'une propriété du Modèle de Vue, History (historique).

Le nombre écrit en gros et en vert est la valeur courante, proposée elle aussi bien entendu par le Modèle de Vue, propriété CurrentNumber.

Quant au cadre gris « *Reset counter* » il permet de saisir un entier (*New Seed*, nouvelle graine) et d'appliquer le changement en cliquant sur le bouton *Reset*. Ici tout sera géré par la Vue (gestion du clic du bouton, collecte des informations et émission du message).

Le snapshot présenté plus haut a été pris peu après le lancement, on voit dans la liste de gauche les nombres 9 à 15 encore présents, et juste après avoir cliqué sur *Reset*, c'est-à-dire après avoir demandé d'initialiser la graine du compteur avec la valeur 500. Cela se voit dans la liste, après 15 apparaissent 500, 501, 502 …

Le Modèle

Il est bien modeste notre modèle… un petit générateur de nombre qui utilise une valeur courante ainsi qu'un timer qui incrémente cette dernière toutes les secondes. On peut démarrer et arrêter le compteur, on peut aussi forcer la valeur courante. Tout cela peut être vu dans le code fourni je ne vais pas m'y attarder (fichier NumberGenerator.cs dans le sous-répertoire Model du projet).

En revanche, cette classe a une particularité qui nous intéresse : on peut changer la valeur courante depuis « l'extérieur » sans même connaitre la classe NumberGenerator ni posséder le moindre pointeur sur elle. On pourrait, sur le même mode, accepter le démarrage ou l'arrêt du compteur, mais dans cet exemple seule la valeur courante est ainsi manipulable.

C'est bien entendu par le biais de la messagerie de MVVM Light que cette valeur va pouvoir être changée. Notre petit générateur s'est tout simplement **abonné à un message de notification générique**.

> *Que la Vue collecte et traite des données n'est pas très conforme à MVVM je l'ai fait remarquer. En revanche qu'une classe quelconque, même issu du modèle, puisse accepter des commandes venant de l'extérieur via des messages peut se révéler une solution originale et digne d'intérêt dans certaines situations.*

Le message contient toutes les données nécessaires à l'exécution de la commande. Ici nous trouvons une mini-classe ne contenant qu'une propriété, la nouvelle valeur du compteur. Dans la réalité cette classe peut contenir tout ce qu'on veut, pourquoi pas du fonctionnel avec des méthodes.

> *J'ai choisi d'implémenter une classe pour le contenu du message afin d'illustrer au mieux l'exemple, il est évident qu'ici nous aurions pu nous contenter d'un type de contenu int sans créer de classe pour contenir un int…*

Cette classe a été créée à part, dans le sous-répertoire Message (ajouté pour ce projet), fichier ResetGeneratorMessageContent.cs contenant le code suivant :

```
namespace Exemple3.Message
{
   public class ResetGeneratorMessageContent
   {
      public static string NotificationString = "NEWSEED";

      public int NewSeed { get; set; }
   }
}
```

La propriété servant à la réinitialisation du compteur est NewSeed, un entier. La propriété statique NotificationString servira de constante pour que l'émetteur et le récepteur soit d'accord sur la nature du message (même si, dans notre exemple, le simple type du message est suffisant pour lever toute ambiguïté).

En effet, on pourrait très bien utiliser la même structure pour passer non pas une nouvelle graine mais un pas d'incrémentation par exemple. Dans ce cas, comme le contenu du message serait de même classe (l'information est la même), il faudrait bien faire la différence entre la notification de changement de la valeur du compteur et celle concernant le changement du pas d'incrémentation. Cela peut se régler en créant des messages différents, ce qui est souvent plus clair, mais parfois le contexte fait qu'un même contenu de message est parfaitement adapté à plusieurs fonctions. La chaine de caractères de **notification devient ainsi un marqueur** utilisé pour différencier les différentes interprétations d'un même message (et d'un contenu de même nature).

Dans notre exemple cela est inutile, mais je voulais vous montrer les nombreuses possibilités de filtrage des messages de MVVM Light.

Lorsqu'il existe plusieurs chaines de notifications différentes (ici nous n'en avons qu'une seule) c'est une bonne idée de les créer sous forme de propriétés statiques string de la classe de contenu de message qui gèrera ces différentes notifications. Ainsi, émetteur et récepteur peuvent utiliser ces constantes et minimiser les inévitables erreurs de saisie si ces chaînes sont retapées à la main à chaque fois.

Cette façon de procéder créé un couplage fort entre la classe message et les deux tiers l'utilisant (l'émetteur et le récepteur). Mais de toute façon l'émetteur devra créer une instance de cette classe, donc la connaître, et le récepteur devra posséder une méthode Action<TMessage>, c'est-à-dire ayant le type du message dans sa signature. On le voit, avec ou sans les constantes statiques le couplage fort existe de fait et on voit mal comment le supprimer. Il n'y a donc aucun danger à utiliser cette guideline.

Le modèle (la classe NumberGenerator) s'enregistre auprès de la messagerie dans son constructeur, de la façon suivante :

```
public NumberGenerator()
{

Messenger.Default.Register<NotificationMessage<ResetGeneratorMessageContent>>(
            this, "Channel0", resetSeed);
}
```

J'ai un peu compliqué les choses ici aussi puisque j'ai utilisé l'une des méthodes d'enregistrement les plus longues. Dans notre cas **d'autres surcharges plus simples étaient suffisantes**. Mais ce choix nous permet de voir toutes les possibilités de la messagerie d'un seul coup.

L'enregistrement, en tant que récepteur, s'effectue ainsi par un appel à Register de l'objet par défaut Default (singleton) proposé par la classe Messenger.

S'agissant d'une méthode acceptant les génériques, nous lui passons les informations de type nécessaire : le message est un NotificationMessage de MVVM Light, lui-même générique, supportant comme contenu de message une instance de ResetGeneratorMessageContent.

Ensuite viennent les paramètres de Register.

En premier l'instance du récepteur, très généralement c'est this, l'instance en cours. Il existe d'ailleurs des surcharges simples de Register qui évitent de passer cette information.

Le second paramètre est appelé Token, de type object. On peut voir cela comme un jeton, ou plutôt comme un canal d'émission/réception. Il est évident qu'ici cela n'est pas nécessaire et que d'autres surcharges de Register éviteraient d'avoir à fixer la valeur du Token. Mais comment en parler si on ne le voit pas… Ici je lui ai donné le nom de « Channel0 », canal zéro, pour bien signifier qu'il s'agit d'un filtre, d'un canal de

communication virtuel. Tout message de même type émis sur un autre canal ou émis *sans précision* de canal *ne sera pas distribué* à notre récepteur.

> *Gérer du fonctionnel en jonglant avec les messages et les canaux d'émission peut devenir assez sportif… Soyez vigilant et créez une documentation claire et sans cesse à jour… sinon les bogues incompréhensibles et difficiles à attraper vont se multiplier !*

Le dernier paramètre de Register est la méthode Action<TMessage> qui sera activée par Messenger le moment venu.

La signature est donc celle d'un délégué de type Action, mais en version générique acceptant un type ouvert pour le contenu du message reçu.

Cela donne dans notre cas :

```
private void resetSeed(NotificationMessage<ResetGeneratorMessageContent> e)
{
  if (e.Notification != ResetGeneratorMessage.NotificationString) return;
  CurrentValue = e.Content.NewSeed;
}
```

Vous voyez que ce n'est pas si compliqué ! La méthode est void et elle supporte un paramètre qui est de type NotificationMessage<ResetGeneratorMessageContent>, c'est un message de type NotificationMessage de MVVM Light, mais dont la propriété Content (le contenu du message) est d'un type ResetGeneratorMessageContent (le type que nous avons déclaré).

> *Si ce cas était réel, et donc écrit avec un souci d'économie, il n'y aurait pas de classe pour le contenu du message et la signature de l'action serait directement NotificationMessage<int> puisque le message sert à véhiculer un unique entier. Pour la démonstration j'ai utilisé la complexité maximale qui n'est pas représentative de l'écriture économe et efficace d'une vraie application.*

S'agissant d'une notification, le message contient une propriété NotificationString qui est initialisée par l'émetteur. Ce sont les « sous commandes » qui seraient acceptées par le message. Ici nous n'avons qu'une valeur, mais pour la démonstration nous vérifions que c'est celle que nous attendons, sinon.. return !

Enfin, la véritable action est exécutée : modifier la propriété CurrentValue du générateur de nombre (le Modèle). La nouvelle valeur est contenue dans une instance de ResetGeneratorMessageContent, instance accessible via la propriété générique Content de l'instance de NotificationMessage de MVVM Light….

La Vue

Ecouter des messages c'est bien, mais encore faut-il que quelqu'un en émettent !

Dans notre exemple c'est la Vue qui sur le clic du bouton *Reset* va avoir la charge de broadcaster la notification.

```
private void btnReset_Click(object sender, RoutedEventArgs e)
{
  int ns;
  ns = int.TryParse(txtSeed.Text, out ns) ? ns : 0;
  Messenger.Default.Send(new NotificationMessage<ResetGeneratorMessageContent>
          (new ResetGeneratorMessage{NewSeed = ns},
           ResetGeneratorMessage.NotificationString )
          ,"Channel0");
}
```

Passons les deux premières lignes qui n'ont pour autre objectif que de récupérer la valeur du TextBox et de la convertir en entier (*ns* pour *new seed*).

> *Pour envoyer le message, comme j'ai bien compliqué la réception avec beaucoup d'options inutiles dans l'exemple mais nécessaires pour vous les faire voir, je suis un peu coincé et obligé de réutiliser la même complexité et les mêmes paramètres à l'émission. Des surcharges de Send existent pour satisfaire les besoins plus courants et plus simples, exactement comme pour la méthode Register.*

Reprenons ainsi l'envoi : On utilise la méthode Send de l'instance par défaut du Messenger. Dans la surcharge de Send utilisée on passe :

- La nouvelle instance de NotificationMessage<ResetGeneratorMessageContent>, instance de NotificationMessage dont la propriété Content sera donc chargée avec une instance de ResetGeneratorMessageContent dont nous initialisons la propriété NewSeed à la valeur calculée plus haut (issue du TextBox saisi par l'utilisateur) ;
- L'instance de NotificationMessage voit son contenu complété par la *clé de notification*. Nous réutilisons la constante statique discutée plus haut et stockée dans la classe de notre contenu de message personnalisé ;
- Enfin, le dernier paramètre de la méthode Send est le Token « Channel0 », le même que celui déclaré par le récepteur quand il s'est enregistré auprès de la messagerie. Le Token n'est pas utile dans notre cas, mais comme le récepteur le gère il faut

absolument que l'émetteur le précise sinon le message n'arrivera jamais. Ce Token pourrait d'ailleurs faire l'objet d'une déclaration de constante, comme la *string* de notification, pour éviter toute erreur de saisie.

Le point sur la notification

Et voilà… en réalité il n'y a que peu de lignes de codes et le système de notification de la messagerie de MVVM Light est très simple. Malgré tout, la généricité, parfois imbriquée, les Token, les clés de notification, les signatures mêmes de certaines méthodes peuvent donner une impression de complexité. Il s'agit là juste d'une question d'entraînement, quand vous aurez écrit votre premier message vous-mêmes vous verrez qu'on ne peut plus s'en passer ensuite ! Attention toutefois aux paramétrages de Register et de Send, il existe beaucoup de surcharges et si l'émetteur et le récepteur n'utilisent pas des méthodes compatibles vous risquez la frustration de ne pas comprendre pourquoi votre message n'arrive jamais à destination !

Exemple de notification de Dialogue

Dans l'arborescence des messages que nous propose MVVM Light se trouve DialogMessage. Cette *enveloppe* est conçue pour contenir un message demandant l'affichage d'un dialogue et obtenir la réponse à la question posée.

Si la Vue peut afficher ce qu'elle veut, c'est bien la seule sous MVVM ! Mais il arrive souvent qu'un Modèle de Vue ait besoin d'une confirmation de type *ok/cancel*. Par exemple pour confirmer la suppression d'une donnée.

Parfois il est possible pour la Vue d'intercaler un tel dialogue avant que la commande ne soit exécutée, mais souvent, surtout si des règles de validation sont appliquées par le Modèle de Vue ou le Modèle, seul ces derniers peuvent prendre la décision d'afficher ou non le dialogue.

Sachant que sous MVVM le Modèle de Vue et le Modèle n'ont aucun accès à l'interface utilisateur, comme faire pour ouvrir une boîte de dialogue et recueillir la réponse ?

C'est là que les messages de notification de type DialogMessage vont être utiles.

> *Un fois encore cette arborescence de messages proposée par MVVM Light n'a rien de définitif, un type générique, d'ailleurs disponible, pourrait servir à tout. Les types spécialisés de MVVM Light ne sont que des aides pratiques simplifiant l'utilisation de la librairie au quotidien. N'oubliez jamais que vous n'êtes en aucun cas contraint à ces types et que vous pouvez écrire les vôtres ou vous contenter des versions génériques de base.*

L'application Exemple 4

L'application exemple numéro 4 a pour but de montrer comment mettre en œuvre une notification de type DialogMessage, c'est-à-dire un type de message généralement envoyé par le Modèle de Vue lorsqu'il a besoin d'une confirmation pour continuer un travail.

> *DialogMessage a été conçu pour fonctionner avec les boîtes de dialogue standard MessageBox, les paramètres du message sont « taillés » pour correspondre aux paramètres nécessaires à l'utilisation d'un tel dialogue. Cela ne signifie pas que vous êtes obligé d'utiliser DialogMessage pour afficher une dialogue, un message générique passant les paramètres nécessaires pourrait faire l'affaire. D'autre part, vous n'êtes pas même obligé d'utiliser un MessageBox.Show pour afficher le message, vous avez le droit d'exploiter le principe en utilisant par exemple une ChildWindow ou même un panel caché qui peut surgir dans votre interface. Dans un tel cas il sera certainement plus intéressant d'utiliser votre propre classe de message en s'inspirant de DialogMessage mais véhiculant des informations plus pertinentes et adaptées à votre besoin.*

Le principe de l'application est le suivant :

Un Modèle est créé, il s'agit d'un article, classe Product. Dans une application réelle nous aurions certainement une collection d'articles ainsi qu'une liaison à une base de données. Ici nous nous satisferons d'un article unique.

Cet article comporte un nom, un stock, et une quantité en commande. Il fournit une information sur la quantité disponible définie comme « stock – commandes en cours ».

Il n'est pas possible de modifier directement le stock ou la quantité en commande, pour cela l'article propose deux méthodes AddToStock(int quantity) et Order(int quantity). La première permet d'aliment le stock, la seconde d'enregistrer une quantité commandée. La classe n'est pas thread safe, ce qui n'a pas d'importance ici mais pourrait en avoir dans une vraie application.

Les entrées en stock ne peuvent pas être négatives. Si tel est le cas la classe Product va utiliser une *notification d'erreur* pour le signaler et refusera l'entrée. Notre exemple implémente ce cas mais par souci de simplicité la Vue ne permet pas de faire une saisie de stock, le Modèle de Vue créera automatiquement une instance de Product initialisée avec un stock de 100 et un nom quelconque.

De même les saisies de commandes doivent correspondre à certains critères. Le premier est qu'il n'est pas possible de saisir une commande négative. Dans un tel cas une notification d'erreur sera émise et l'entrée sera refusée. Ce cas de figure est implémenté dans l'exemple

et comme il ressemble au cas du stock négatif vous comprenez pourquoi le premier n’est pas codé.

Il existe une seconde contrainte sur la saisie d’une commande : si la quantité commandée dépasse la quantité disponible (stock moins les commandes en cours) nous ne refusons pas la commande mais nous souhaitons avertir l’utilisateur de cette situation ainsi qu’obtenir de sa part une confirmation.

C’est là que les choses se compliquent…

Mais prenons-les plutôt dans l’ordre pour démêler la pelote calmement !

Le Modèle

Il s’agit de la classe Product.

Une classe très simple qui expose quatre propriétés :

- Name, le nom du produit
- CurrentStock, la quantité actuellement dans l’entrepôt
- OnOrder, la quantité en commande (qu’on suppose non livrées, nous n’allons pas implémenter une gestion commerciale complète…)
- Available, la quantité disponible, c'est-à-dire le stock courant diminué des commandes en cours non livrées.

Le code est le suivant :

```
/// <summary>
/// Gets or sets the name.
/// </summary>
/// <value>The name.</value>
public string Name
{
   get { return name; }
   set
   {
      if (name == value) return;
      name = value ?? string.Empty;
      RaisePropertyChanged("Name");
   }
}

/// <summary>
/// Gets the current stock.
/// </summary>
/// <value>The current stock.</value>
public int CurrentStock
{
   get { return currentStock; }
}

/// <summary>
/// Gets the quantity on order.
/// </summary>
/// <value>The on order.</value>
public int OnOrder
{
   get { return onOrder; }
}

/// <summary>
/// Gets the available quantity (stock - On Order).
/// </summary>
/// <value>The available.</value>
public int Available
{
   get { return currentStock - onOrder; }
}
```

Pour saisir le stock on ne peut pas manipuler directement CurrentStock (notre cahier des charges), Product propose ainsi une méthode AddToStock qui permet d’alimenter le stock :

```
/// <summary>
/// Adds products to stock.
/// </summary>
/// <param name="quantity">The quantity.</param>
public void AddToStock(int quantity)
{
  if (quantity < 0)
  {
    doError(ProductError.AddToStockCantBeNegative);
    return;
  }
  currentStock += quantity;
  RaisePropertyChanged("CurrentStock");
  RaisePropertyChanged("Available");
}
```

Si la quantité passée est inférieure à zéro la saisie est refusée, *mais la méthode émet une erreur* par le biais de doError, méthode privée qui prend en paramètre un type d'erreur (une énumération).

Si la quantité est valide, le stock est incrémenté et bien entendu nous levons les PropertyChanged qui correspondent aux propriétés qui sont modifiées par la manipulation.

La méthode doError se présente comme suit :

```
private void doError(ProductError error)
{
  Messenger.Default.Send(
    new NotificationMessage<ProductError>( error, ErrorNotification));
}
```

Elle utilise l'énumération ProductError :

```
namespace Exemple4.Model
{
  public enum ProductError
  {
    AddToStockCantBeNegative,
    OrderCantBeNegative
  }
}
```

doError utilise la messagerie de MVVM Light pour émettre une notification particulière : elle possède, en plus d'une *string* de notification, un contenu qui est de type ProductError

(l'énumération). La *string* de notification est une constante, et comme dans les exemples précédents elle est définie comme un champ statique de la classe en cours (la chaine est définie comme "ProductError").

Ici, une notification simple sera donc émise. Elle possédera un marqueur, la *string* de notification qui permettra aux éventuels récepteurs de filtrer les notifications pour ne traiter que celles qui sont des erreurs de la classe Product. Le message sera complété par le type d'erreur.

Qui va traiter ce message ? S'agissant d'une simple notification ayant pour but de prévenir l'utilisateur c'est la Vue qui sera en charge de l'affichage. Un simple MessageBox.Show sera suffisant, mais nous verrons cela en regardant le code de la Vue.

La seconde méthode de la classe Product est Order :

```
/// <summary>
/// Records an order of a given quantity
/// </summary>
/// <param name="quantity">The quantity.</param>
public void Order(int quantity)
{
   if (quantity < 0)
   {
      doError(ProductError.OrderCantBeNegative);
      return;
   }
   if (quantity > Available)
   {
      askForConfirmation(quantity, Available - quantity);
      return;
   }
   onOrder += quantity;
   RaisePropertyChanged("OnOrder");
   RaisePropertyChanged("Available");
}
```

Au début la méthode fait comme la précédente : les entrées négatives sont interdites. Si une quantité inférieure à zéro se présente elle est tout simplement ignorée mais l'erreur est signalée par un appel à doError (avec un type d'erreur différent du cas précédent). Il y aura donc une *notification simple*.

Là où les choses se compliquent c'est dans le cas où la quantité devient supérieure à la disponibilité physique du produit. Dans ce cas nous ne refusons pas la saisie mais en raison

de ses conséquences (disponibilité négative) nous devons **obtenir une confirmation de la part de l'utilisateur**.

Quel utilisateur ? Nous sommes dans une classe du Modèle ! Entre ce Modèle et l'interface il y a tout un monde… en aucun cas une classe du Modèle, pas plus qu'un Modèle de Vue ne peut se permettre d'accéder à l'interface utilisateur.

La notification, comme nous l'avons vu jusqu'à maintenant, ne règle qu'une partie du problème : ici il ne s'agit pas seulement d'afficher un message, mais **il faut attendre et traiter la réponse**.

Or, comme nous n'avons vraiment pas de chance, l'asynchronisme règne en maître dans les environnements modernes. Impossible de lancer une notification et d'attendre gentiment la réponse à la ligne de code suivante. Rien n'est bloquant, pas même dans ce cas.

Comment la méthode Order va-t-elle s'en sortir ?

C'est une grosse maline elle transfert le problème à une méthode qui s'appelle AskForConfirmation en lui passant en paramètre la quantité demandée ainsi que ce que serait le stock si la commande était validée. Et puis c'est tout ! Un simple return et on oublie le problème !

En revanche si aucun problème n'est détecté, Order se comporte comme on s'y attend un peu : la quantité OnOrder est incrémentée et les PropertyChanged des propriétés qui sont impactées par cette modification sont invoqués.

Joli tour de passe-passe… Et la commande qui rend la disponibilité négative, elle devient quoi ?

Regardons le code de AskForConfirmation :

```
private int quantityToConfirm;
private void askForConfirmation(int quantity, int newStock)
{
  var msg = string.Format("An order of {0} items will create a negative availability ({1}).\n Do you confirm your order ?", quantity, newStock);
  quantityToConfirm = quantity;
  Messenger.Default.Send(new DialogMessage(msg, manageConfirmation)
              {Button=MessageBoxButton.OKCancel,Caption = "Confirmation"});
}
```

La méthode commence par formater un message indiquant le problème, elle utilise pour cela les paramètres qui lui ont été passés. Le message indique « *une commande de xxx items créera une disponibilité négative. Confirmez-vous cette commande ?* ».

Ensuite la méthode mémorise la quantité demandée dans une variable privée de l'instance de Product, quantityToConfirm.

Puis elle utilise la messagerie de MVVM Light pour émettre un message particulier, une notification de type DialogMessage. Ce message contient de nombreux éléments bien pratiques pour afficher une boîte de dialogue sans que la Vue qui va s'en charger n'ait besoin de savoir quoi que ce soit. Son rôle sera juste d'afficher (interface utilisateur), pas de comprendre à quoi cela sert (fonctionnel, donc pas du ressort de la Vue). On trouve bien entendu dans les paramètres du message MVVM Light le message à afficher (msg) ainsi qu'un *Callback*, manageConfirmation, la méthode qui sera appelée lorsque le dialogue aura été validé par l'utilisateur. D'autres paramètres permettent d'indiquer quels sont les boutons qui seront disponibles ou bien de fixer le titre de la boîte de dialogue.

Donc, **pour résumer**, lorsqu'une commande risque de faire passer la disponibilité en négatif, la méthode Order se décharge du problème en passant la quantité demandée à une autre méthode, AskForConfirmation. Cette dernière formate un message de type DialogMessage qu'elle émet, en ayant pris soin de **mémoriser la quantité réclamée** et de fournir au message une méthode *Callback* qui traitera la réponse.

Le message de dialogue va ainsi se propager et atteindre son destinataire, destinataire totalement inconnu de Product rappelons-le. Ce récepteur aura pour charge **d'afficher le message et de recueillir la réponse de l'utilisateur puis d'appeler le Callback** fourni en précisant quelle est la réponse de l'utilisateur.

Si aucun récepteur ne répond à l'appel le message sera oublié… et la commande ignorée. La classe Product respecte ainsi le paradigme de la programmation objet et notamment l'une de ses règles les plus précieuses : l'encapsulation qui permet entre autre de protéger l'état de l'objet vis-à-vis de l'extérieur (les deux autres règles étant, pour rappel, le polymorphisme et l'héritage).

> *Si cela répond aux canons de la programmation moderne, fonctionnellement on peut se demander si ignorer totalement le cas de « non réponse » est fiable et solide. Ma réponse est non, bien sûr, puisqu'ici on touche à du fonctionnel. Dans la réalité il faudrait créer une liste des demandes en attente de réponse au sein de Product et mettre en place un mécanisme (lors de la validation de la commande ou du panier dans une application Web) qui vérifie que toutes les questions ont obtenu réponse. Cela serait bien plus satisfaisant. Mais vous comprendrez qu'ici je n'ai pas implémenté un tel mécanisme. A vous d'y penser lors de la conception de vos logiciels !*

Pour terminer la chaine de ce traitement asynchrone, regardons maintenant le code du *Callback* manageConfirmation :

```
private void manageConfirmation(MessageBoxResult result)
{
   if (result == MessageBoxResult.OK)
   {
      onOrder += quantityToConfirm;
      RaisePropertyChanged("OnOrder");
      RaisePropertyChanged("Available");
   }
   quantityToConfirm = 0;
}
```

En raison du type d'Action choisi, la signature de la méthode comporte un paramètre de type MessageBoxResult. Elle ne fait qu'exploiter ce résultat. Si l'utilisateur a validé le dialogue par l'appui sur le bouton Ok, alors **seulement maintenant la quantité en commande du produit est incrémentée** de la quantité demandée sauvegardée précédemment. Suivent les notifications de PropertyChanged des propriétés impactées afin que la Vue puisse refléter les changements.

La zone de mémorisation est remise à zéro par sécurité. C'est là que la demande en attente serait supprimée de la liste évoquées plus haut, ce qui permettrait à la partie gérant la commande en cours de s'assurer que toutes les questions en suspens on bien reçu réponse.

AMELIORER LA GESTION DE DIALOGMESSAGE

Peut-être un jour Laurent Bugnion ajoutera-t-il cette fonctionnalité, mais pour l'instant il y a un manque cruel à la classe DialogMessage, une propriété « Context » de type object. Une sorte de Tag si on veut.

En effet, comme on vient d'ailleurs de le voir, DialogMessage va être utilisé dans des situations où un traitement reste « suspendu » jusqu'à ce que la réponse arrive. D'une part plusieurs requêtes peuvent partir avant même que la première réponse ne soit arrivée,

ensuite, l'ordre de ces réponses n'est pas garantie. Il en résulte que conserver le contexte sous la forme d'une variable locale n'est vraiment pas raisonnable. Pour traiter correctement une telle situation il faudrait accrocher au message le contexte nécessaire au traitement de la réponse et que ce contexte soit retourné en même temps que la réponse. Les problèmes que je viens de citer disparaissent alors… C'est pourquoi j'ai écrit la classe DialogMessageWithContext que vous trouverez dans le projet Enaxos.MvvmLight.Extensions. Cette classe est volontairement placée dans le même espace de nom que DialogMessage, c'est-à-dire le namespace GalaSoft.MvvmLight.Messaging pour éviter le foisonnement des « using » dans les applications. L'exemple 4 montré ici a été réécrit en utilisant cette nouvelle classe, le projet exemple s'appelle Exemple4Context. Pour simplifier la compilation de cette démonstration l'unité de code contenant l'extension a été intégrée au projet au lieu de faire référence aux librairies. Il est bien entendu préférable d'utiliser ces dernières, mais ici cela rend la démonstration plus simple à utiliser.

Vous trouverez ces sources avec celles fournies dans le zip de l'article. L'exemple 4 modifié est rigoureusement identique à l'original sauf qu'il utilise DialogMessageWithContext et que « quantityToConfirm » n'existe plus dans le Modèle, la valeur est envoyée dans le message en tant que contexte de celui-ci et est ré-exploitée à l'arrivée du message en transtypant le contexte (de type object) en int. Le mieux étant d'ouvrir le projet et de regarder le code !

A noter aussi : le répertoire Extensions contient deux sous-répertoires : Enaxos.MvvmLight.Extensions, et Enaxos.MvvmLight.Extensions (SL). Le premier contient le code original sous la forme d'une librairie de classe WPF4, le second est un projet se liant aux sources du précédent pour créer une librairie de classe Silverlight 4.

Vous pouvez sur le même principe créer des versions ciblant d'autres releases de ces frameworks.

Le modèle de Vue

Il ne faut pas l'oublier, c'est lui qui permet à la Vue d'avoir accès aux données notamment. Le Modèle de Vue de notre exemple comporte une propriété Title déjà utilisée dans les exercices précédents, une commande (nous verrons cela plus loin), et une instance de Product qu'elle a créée et initialisée. Dans la réalité il s'agirait d'une collection de produits, ou d'un produit retourné par une requête Linq par exemple.

Les commandes n'étant pas encore notre propos, disons juste que la commande implémentée est liée au bouton de validation ce qui déclenche une méthode du Modèle de Vue appelant « product.Order(quantité) », méthode décrite dans le Modèle un peu plus haut.

Le Modèle de Vue joue parfaitement son rôle ici, mais pour ce qui nous intéresse (les messages de type DialogMessage) ce rôle ne nous concerne pas. Passons plutôt à la Vue.

La Vue

Avant son relookage sous Blend, voici à quoi elle correspond sous Visual Studio :

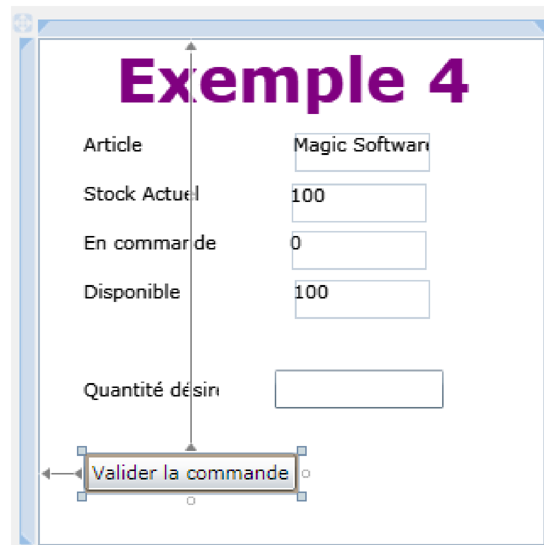


Figure 56 - la Vue de l'exemple DialogMessage

On trouve une série de libellés et de données, celles de l'instance de la classe Product proposée par le Modèle de Vue. On voit que grâce aux mécanismes de MVVM Light nous voyons les données en conception, ce qui est bien pratique.

Ensuite la Vue propose un champ de saisie (quantité désirée) assorti d'un bouton « Valider la commande ».

L'aspect ne nous intéresse pas ici, pas plus que le binding de tous ces éléments via le DataContext de la Vue. Ces sujets sont censés être maîtrisés.

En revanche, le code de la Vue va nous fournir **la dernière pièce du puzzle** de la gestion de DialogMessage : le code qui répond aux diverses notifications…

Commençons par les choses simples, les notifications d'erreur. Dans le constructeur de la Vue on trouve :

```
Messenger.Default.Register<NotificationMessage<ProductError>>(this,
displayProductError);
```

Ce code enregistre la Vue comme étant récepteur des messages de type NotificationMessage<ProductError>. Lorsqu'un message de ce type arrive à la Vue, la méthode displayProductError est invoquée. Elle est définie comme suit :

```
private static void displayProductError(NotificationMessage<ProductError> msg)
{
  if (msg.Notification != Product.ErrorNotification) return;
  MessageBox.Show(msg.Content.ToString(), "Error", MessageBoxButton.OK);
}
```

La méthode vérifie qu'elle a bien à faire à une notification portant sur une notification d'erreur d'un produit. Dans notre exemple le doute n'est pas permis mais dans la réalité la *string* de notification joue un rôle important pour « verrouiller » les échanges de messages et être certain de ne traiter que ceux auxquels on s'attend.

Ensuite la méthode se contente d'utiliser MessageBox.Show pour afficher le message d'erreur. Dans notre exemple il s'agira simplement de la traduction en string de la valeur de l'énumération. En réalité on aurait un switch qui, en fonction de la valeur reçue, construirait un message un peu plus « *user friendly* », qui d'ailleurs serait construit avec un string.Format à l'aide d'une chaîne de caractère issue d'une ressource conforme à la culture en cours (la localisation c'est essentiel !)… On comprend bien qu'ici tout cela ne peut pas être implémenté.

Passons maintenant à la notification particulière de DialogMessage.

Le constructeur de la Vue s'enregistre de la façon suivante :

```
Messenger.Default.Register<DialogMessage>(this,
 msg=>
    {
      var result = MessageBox.Show(msg.Content, msg.Caption, msg.Button);
              msg.ProcessCallback(result);
    });
```

Et voici la feinte dite de l'expression Lambda… Nous aurions pu l'utiliser pour la notification d'erreur mais il faut varier les plaisirs… Ici, en même temps que la Vue s'enregistre auprès de la messagerie pour réceptionner les messages de type DialogMessage, elle enregistre non pas l'adresse de la méthode qui gérera l'action mais directement le code de cette action sous la forme d'une expression Lambda.

Cette façon de faire particulièrement élégante supprime une méthode qui n'aurait guère d'intérêt en tant que telle. De plus, le traitement des messages de type DialogMessage ne nécessite aucune « réflexion » de la part de la Vue, aucun traitement. Elle ne fait que relayer la demande d'affichage qui lui arrive, affiche le message servilement, et tout aussi docilement elle appelle le *Callback* en fournissant la réponse de l'utilisateur. Quel que soit

l'émetteur et la nature du message, la Vue le traitera toujours de la même façon « sans réfléchir » car ce n'est pas son rôle. Plusieurs classes peuvent ainsi envoyer des notifications de type DialogMessage, bien que leur contenu, leur sens, et leur traitement soient radicalement différent, du point de vue de la Vue, tous seront gérés identiquement.

La partie intelligente du traitement du message est réalisée dans cet exemple par le Modèle, la classe Product. En aucun cas la Vue ne doit interférer dans ce traitement, son rôle est bien de présenter des données à l'utilisateur et de répercuter ses actions sur le Modèle de Vue (ou bien le *Callback* dans le cas d'un message DialogMessage).

L'application en action

Voici quelques snapshots de l'application en action pour bien comprendre son fonctionnement.

Figure 57 - Exemple 4 : Etat initial

Figure 58 - Exemple 4 : Après la validation d'une commande de 50 unités

Figure 59 - Exemple 4 : Tentative d'une commande négative

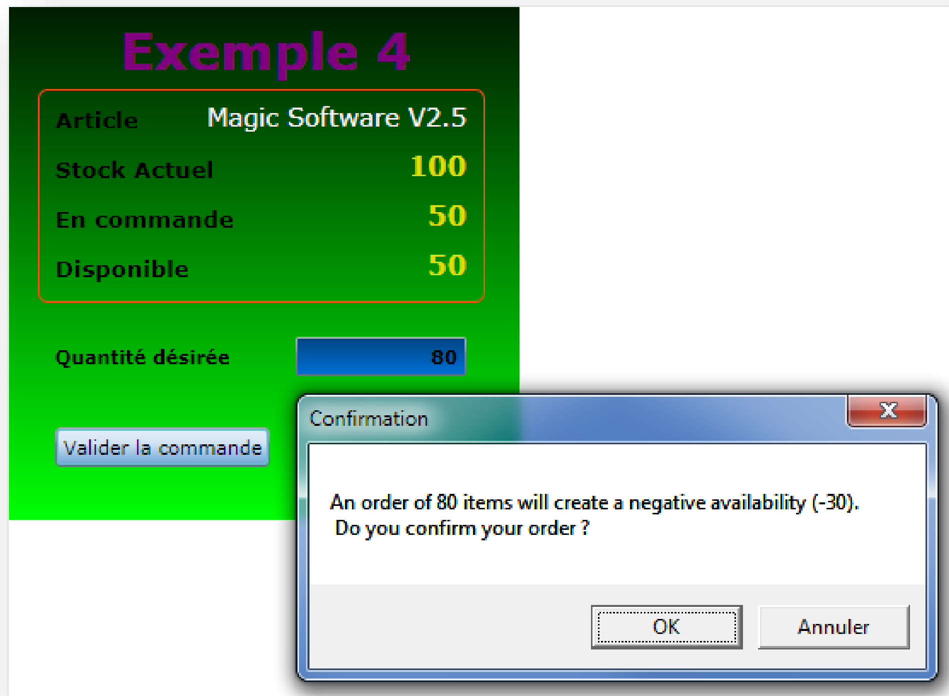


Figure 60 - Exemple 4 : Saisie supérieure à la validité, affichage du message

Rien ne se passe si on répond « annuler », la commande est enregistrée et le stock disponible passe à moins 30 si on répond « Ok »…

Le point sur les notifications DialogMessage

Vous l'avez compris, la difficulté ici n'est pas dans l'utilisation de MVVM light pour envoyer ou recevoir des messages, mais bien **dans la gestion asynchrone de ces messages** au sein d'une application qui doit réaliser **une tâche précise** bien qu'elle soit **interrompue pour un temps indéterminé**.

La solution consiste à découper le traitement en deux parties :

- Préparation et envoi du message de dialogue
- Traitement de la réponse lorsqu'elle arrive

Cela implique que le traitement qui se trouve ainsi tronçonné doit être capable de se poursuivre après l'interruption.

Il n'y aucune magie, aussi contraignant que cela soit, i**l est nécessaire de mémoriser l'état du traitement avant l'envoi du message** et **d'être capable de le récupérer** une fois le message arrivé.

Dans notre exemple cette mémorisation est simplement assurée par une variable de l'instance.

La méthode n'est pas mauvaise en soi mais possèdent tout de même quelques désavantages sérieux…

Le code actuel de MVVM Light ne permet pas de répondre aux problèmes soulevés qui seraient réglés par l'ajout d'un contexte transmis dans le message et retourné avec la réponse. Ainsi il n'y aurait plus de stockage local (la variable quantityToConfirm) donc plus de besoin de mémoriser en interne l'état du traitement stoppé par l'envoi du message, il n'y aurait plus, non plus, à se soucier des messages multiples ni de leur ordre d'arrivée. Bref cela serait le paradis… Après écriture de cet article j'ai finalement décidé d'implémenter cette solution comme indiqué quelques pages plus haut.

De fait, la solution existe, il suffit d'utiliser DialogMessageWithContext, classe que je vous offre avec l'article (assortie de sa démonstration)…

La gestion des commandes

La gestion des commandes est quelque chose d'essentiel en programmation, en général, et prend une dimension bien plus grande sous la pattern MVVM qui implique le découplage entre la Vue qui enverra des commandes et le Modèle de Vue qui est censé les recevoir et les traiter.

Malgré tout la gestion des commandes est le parent pauvre de Xaml, aussi étonnant que cela puisse paraître. Et si c'est pauvre dans WPF, ça l'est encore plus sous Silverlight hélas (même si ce dernier implémente depuis peu une partie de la « plomberie »).

De fait, les librairies qui tentent d'aider le développeur à mettre en œuvre MVVM Light proposent toutes une solution à ce problème. Sous **Prism** par exemple, cela prend la forme de la classe DelegateCommand. Sous MVVM Light, Laurent Bugnion ré-exploite une classe créée par *Josh Smith*, RelayCommand.

Pour rappel, l'idée consiste à éviter d'avoir à créer une classe supportant ICommand pour chaque commande du projet. RelayCommand est ainsi une de ces classes, elle implémente ICommand, mais au lieu d'implémenter cette interface « en dur » elle en remplace chaque partie par la gestion d'un délégué (ou une expression lambda). D'où le nom de la classe, *Relay Command*, *Relayeur de Commande*.

Comme RelayCommand implémente ICommand, il suffit pour le Modèle de Vue d'exposer des propriétés de type ICommand (ou directement RelayCommand) qui sont initialisées (généralement dans le constructeur) par des instances de RelayCommand pointant vers les méthodes du Modèle de Vue (ou autre objet) en charge des impératifs du contrat que représente ICommand.

Une fois les propriétés ICommand exposées comme les autres, il devient possible de les binder à des éléments de la Vue, des composants visuels qui gèrent une propriété Command.

Hélas, de base, WPF et plus encore Silverlight ne proposent que peu de composants ayant cette facilité. Pour Silverlight il s'agit pour l'instant uniquement de la classe ButtonBase et de ses descendants (la classe Button par exemple).

Nous allons voir que MVVM Light offre une solution complémentaire à RelayCommand pour contourner ce (très gros) problème qui limite dans la pratique la mise en œuvre de MVVM. Mais commençons par le commencement…

RelayCommand (version non générique)

La version la plus basique, non générique, de RelayCommand s'utilise de façon étonnement simple.

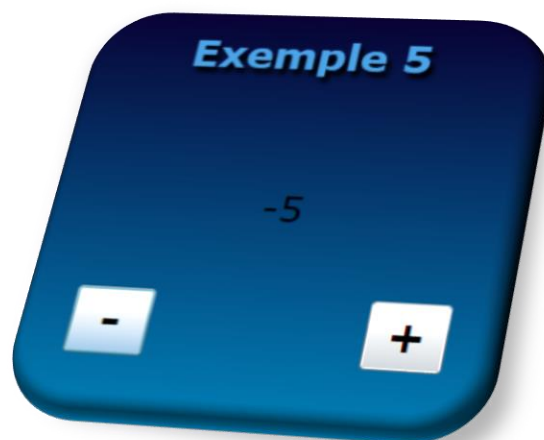


Figure 61 - Exemple 5 - RelayCommand simple

L'application exemple numéro 5 comporte un titre, une zone centrale présentant un entier et deux boutons. L'un sert à incrémenter la valeur, l'autre à la décrémenter.

Pour cela, le Modèle de Vue expose une propriété entière CurrentValue (initialisée à zéro) et deux propriétés de type RelayCommand : Increment et Decrement.

> *Certains aiment ajouter le suffixe Command aux commandes, ici il n'y a pas d'ambiguïté mais il est vrai que lorsque le Modèle de Vue devient plus gros cette convention aide l'intégrateur lorsqu'il affiche le dialogue de binding, sous Blend ou VS. A vous d'adopter ou non cette convention.*

Les boutons voient leur propriété Command être bindée à l'une ou l'autre de ces commandes.

Commençons par les commandes implémentées dans le Modèle de Vue :

```
public RelayCommand Increment { get; private set; }
public RelayCommand Decrement { get; private set; }
```

Il s'agit donc bien de propriétés “normales”, avec un getter public et un *setter* privé. Il n'y a pas de champ sous-jacent (*backing field*).

C'est dans le constructeur du Modèle de Vue que sont créées les instances de RelayCommand (et que sont initialisées les Action à réaliser) :

```
public MainViewModel()
{
   Increment = new RelayCommand(incrementValue);
   Decrement = new RelayCommand(() => CurrentValue--);
}

private void incrementValue()
{
   CurrentValue++;
}
```

J'ai volontairement ici utilisé les deux stratégies les plus courantes. Pour la première commande j'ai utilisé la méthode « classique » qui consiste à fournir en paramètre de RelayCommand le nom d'une méthode privée qui possède une signature compatible avec Action. Ici il s'agit de incrementValue() dont le seul rôle est d'incrémenter la propriété CurrentValue (le *setter* de cette propriété effectue un RaisePropertyChanged pour notifier la Vue du changement).

Pour la seconde commande j'ai utilisé la méthode la plus économe en code, l'utilisation d'une *expression Lamba*. Ici nul besoin de créer une méthode pour si peu. RelayCommand enregistre directement le code à exécuter.

Côté Xaml les binding sont réalisés comme suit (j'ai supprimé tout ce qui concerne la présentation des balises) :

```
<TextBlock Text="{Binding Path=CurrentValue}" />
<Button Content="-" Command="{Binding Path=Decrement}" />
<Button Content="+" Command="{Binding Path=Increment}" />
```

On peut difficilement faire plus simple…

Et ça marche, bien entendu (voir le projet Exemple5 fourni avec l'article).

Les autres éléments de ICommand

ICommand ne comporte pas seulement une méthode Execute, cette interface propose aussi

- CanExecute, une méthode retournant un booléen qui indique à tout moment si la commande représentée par ICommand est utilisable ou non. Les composants qui gèrent ICommand comme la classe Button exploitent cette propriété pour basculer automatiquement leur propriété Enabled à true ou false.
- CanExecuteChanged, un événement doit être levé à chaque fois que la valeur de CanExecute change (en raison de condition internes ou externes à l'application).

RelayCommand tient compte de toutes ces contraintes de l'interface ICommand et propose des surcharges à son constructeur prenant en compte les paramètres nécessaires à l'initialisation de ces éléments supplémentaires.

CanExecute est définie comme une signature Func<bool>, tout délégué ou expression Lambda correspondant à cette signature peut être utilisé.

Nous pouvons améliorer notre programme Exemple 5 en complémentant les définitions des commandes. Ainsi, nous allons limiter l'incrémentation et la décrémentation à la fourchette [-5,5].

Une première version serait la suivante :

```
Increment = new RelayCommand(() => CurrentValue++, () => CurrentValue < 5);
Decrement = new RelayCommand(() => CurrentValue--, () => CurrentValue > -5);
```

Les commandes possèdent désormais, en plus de l'action, une fonction évaluant la validité de la commande. Comme le mécanisme interne des commandes fonctionne sur le mode « if (CanExecute) Execute() » l'utilisateur ne pourra désormais plus dépasser les valeurs -5 et 5.

Toutefois il manque quelque chose d'important, le fait que l'exécution soit contrôlée est déjà une bonne chose, mais au niveau de l'interface cela ne suit pas : les boutons restent actifs alors que les commandes deviennent inactives (quand les valeurs bornes sont atteintes).

A cela une bonne raison : Silverlight n'implémente pas de CommandManager comme WPF qui se charge de vérifier automatiquement si les commandes sont disponibles.

Pour pallier ce manque, MVVM Light ajoute à RelayCommand la méthode RaiseCanExecuteChanged qui force le rafraîchissement de l'interface utilisateur. Cette méthode est utilisable aussi sous WPF si c'est le code qui change la valeur de CanExecute.

Telles que nos commandes sont écrites il n’est pas possible d’exploiter cette possibilité, nous allons ainsi revenir à une écriture plus classique en codant les méthodes à passer aux deux instances de RelayCommand.

De même nous allons utiliser le RaiseCanExecuteChanged de chaque commande à chaque fois que la valeur change.

On notera dans le code qui suit :

- Le retour à l’utilisation de délégués
- La création d’une méthode pour chaque action (incrément ou décrément)
- La création d’une méthode de contrôle pour chaque action
- L’appel à la méthode RaiseCanExecuteChanged des deux commandes dans chaque action (sinon les boutons resteraient « coincés » en mode disabled une fois basculés dans cet état)

Ce n’est pas très compliqué, et cela fait déjà beaucoup de chose, le tout en conformité avec MVVM, c'est-à-dire que la Vue ne sait absolument rien des traitements et des contrôles qui sont du ressort exclusif du Modèle de Vue.

> *On appréciera aussi que le respect du pattern MVVM nous permettrait maintenant d’écrire un programme de test qui appellerait les commandes du Modèles du Vue et qui pourrait vérifier si elles fonctionnent sans que jamais nous n’ayons besoin de l’interface utilisateur ! Cette dernière peut d’ailleurs fort bien ne pas encore exister, être entre les mains des designers alors même que le logiciel peut déjà être testé… C’est un gros avantage de MVVM bien entendu.*

```
public MainViewModel()
{
 Increment = new RelayCommand(incrementValue, canIncrementValue);
 Decrement = new RelayCommand(decrementValue, canDecrementValue);
}

        private bool canDecrementValue()
        {
            return CurrentValue > -5;
        }

        private void decrementValue()
        {
            CurrentValue--;
            Decrement.RaiseCanExecuteChanged();
            Increment.RaiseCanExecuteChanged();
        }

        private bool canIncrementValue()
        {
            return CurrentValue < 5;
        }

        private void incrementValue()
        {
            CurrentValue++;
            Increment.RaiseCanExecuteChanged();
            Decrement.RaiseCanExecuteChanged();
        }
```

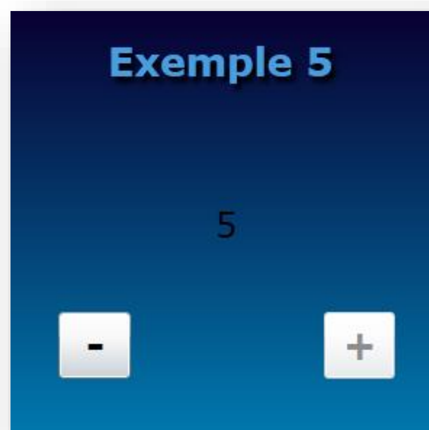


Figure 62 - Exemple 5 : le bouton + est "disabled"

RelayCommand avec paramètre générique

Si la plupart des commandes sont des actions simples cela ne veut pas dire qu'elles ne dépendent pas d'une façon ou d'une autre du « contexte » (au sens large).

Le plus souvent ce contexte est celui du Modèle de Vue lui-même. Les méthodes d'action peuvent ainsi se référer aux autres propriétés ou appeler des méthodes du Modèle de Vue. C'est d'ailleurs un autre gros avantage de RelayCommand, car sans cette classe, il faudrait créer des classes (une par commande) et il faudrait implémenter des mécanismes permettant à la multitude des classes de commande d'avoir accès aux propriétés et méthodes de la classe du Modèle de Vue. Avec RelayCommand tout le code est intégré à la classe Modèle de Vue ce qui simplifie énormément les choses.

Tout cela fait que généralement un commande peut se suffire de ce que nous en avons vu jusqu'à maintenant.

Mais il existe des cas où l'on peut avoir besoin de passer un paramètre à la commande.

Ces cas sont finalement assez rares.

On pourrait penser par exemple à la valeur d'un Slider. Mais si ce dernier a une importance sa valeur devrait être bindée avec une propriété du Modèle de Vue, dès lors la commande pourrait se référer à cette dernière plutôt que de tenter d'extraire la valeur du Slider et de la passer en paramètre à la commande…

Il en va de même avec le texte d'un TextBox. Si sa valeur compte, elle doit être bindée avec une propriété string du Modèle de Vue. Du coup, une action qui dépendrait de la valeur saisie dans le TextBox n'aurait nul besoin d'extraire celle-ci et de la passer en paramètre à la commande dont le code pourrait directement consulter la valeur de la propriété correspondante dans le Modèle de Vue…

Plus on y réfléchit plus on se dit que cette option, ce paramètre à passer à la commande, ne fait que mettre en évidence un certain *laxisme* dans l'implémentation du Modèle de Vue… On évite de coder certaines propriétés alors qu'elles sont nécessaires (la preuve) au bon fonctionnement du Modèle de Vue.

Personnellement j'adhère à cette vision et je conseille plutôt de toujours implémenter des « *valeurs miroir* » dans le Modèle de Vue, valeur reflétant celles de certains éléments d'interface dont les commandes peuvent avoir besoin. Et je ne vois guère de raisons, en dehors de la paresse, de contourner cette règle.

Mais par sagesse je sais que j'oublie peut-être (et certainement) certains cas où la gestion d'un paramètre est un « plus », une simplification. J'admets bien volontiers que de tels cas puissent exister.

Par exemple, on pourrait avoir une commande « *Print* » qui selon le paramètre passé se transformerait en méthode *Print* ou *Preview*. Pourquoi pas.

Mais même ici encore je préfère largement l'implémentation de *deux commandes distinctes* (*Print* et *Preview*), même si ces commandes utilisent un même code interne auquel elles passent un paramètre pour demander le *Print* plutôt que le *Preview* (et cela serait dans l'idéal la valeur d'une énumération et non un texte).

Même en faisant des efforts (j'essaye honnêtement) je ne vois derrière les paramètres de commande que les stigmates d'une *implémentation paresseuse ou mal planifiée*.

Mais soit ! Soyons bons joueurs et admettons que cela puisse être utile quelques fois. Comme MVVM Light l'implémente je ne vais pas faire le censeur, et je vais vous montrer comment cela fonctionne.

MVVM Light propose ainsi RelayCommand<T>.

Le type générique qui doit être précisé est celui du paramètre. Cela permet de passer un string, un double, un entier, ou même une instance complexe provenant du Modèle ou d'une de ses abstractions.

Le cas le plus classique où l'on voit RelayCommand utilisé avec un paramètre est par exemple le passage de la valeur d'un TextBox (pour un login par exemple). Quand bien même j'ai montré qu'on pouvait le faire « plus proprement » (à mon sens), nous allons voir comment mettre en œuvre ce cas précis.

Pour cela nous allons agrémenter l'exemple 5 d'un TextBox qui servira à fixer la fourchette de valeurs acceptées. Pour faire simple nous attendrons un seul nombre, la fourchette sera toujours égale à [-nombre,+nombre].

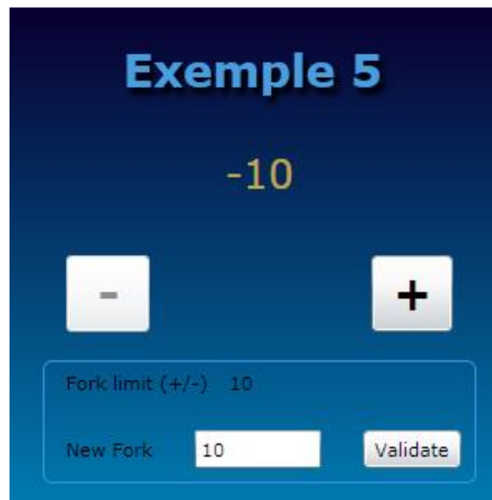


Figure 63 - RelayCommand avec paramètre générique

L'exemple 5 ressemble maintenant à ce qui est montré sur la figure ci-dessus. Le snapshot a été pris après que le changement de fourchette ait été validé (-10/+10) et après avoir cliqué autant de fois que nécessaire sur le bouton de décrémentation jusqu'à ce qu'il se mette dans l'état *disabled*.

Je passerai sur le fait que j'ai déclaré une propriété entière ForkValue qui est notamment bindée au TextBlock indiquant la limite de la fourchette en cours. Rien de savant ici.

Regardons plutôt la définition de la commande :

```
public RelayCommand<string> SetFork { get; private set; }
```

Comme nous allons passer le texte contenu dans le TextBox, notre paramètre sera de type string. Pour le reste cette commande ressemble aux autres.

> *Vouloir traduire dans la Vue le texte du TextBox en entier avant de le transmettre ou écrire un convertisseur me semble des solutions lourdes, voire scabreuses. Quitte à ce que la vue transmette des données par message, autant qu'elle en fasse le minimum. Si conversion il doit y avoir (comme dans cet exemple) c'est au Modèle de Vue de s'en charger. Adapter les données pour la Vue est déjà dans ses attributions, lui ajouter l'adaptation des données venant de la Vue m'apparait une raisonnable réciproque alors qu'écrire du code dans la Vue pour adapter dans celle-ci des données pour le Modèle de Vue me semble une aberration.*

Dans le constructeur du Modèle de Vue nous trouvons son initialisation :

```
SetFork = new RelayCommand<string>(setFork);
```

Initialisation qui nous renvoie à la définition de la méthode setFork :

```
private void setFork(string fork)
{
   int nf;
   nf = int.TryParse(fork, out nf) ? nf : forkValue;
   ForkValue = nf;
}
```

Son travail est fort simple : transformer la chaine reçue en un entier, puis modifier la propriété ForkValue en conséquence.

Le setter de cette dernière mérite tout de même un coup d'œil :

```
public int ForkValue
{
   get { return forkValue; }
   set
   {
      if (forkValue == value) return;
      forkValue = value;
      RaisePropertyChanged("ForkValue");
      Decrement.RaiseCanExecuteChanged();
      Increment.RaiseCanExecuteChanged();
   }
}
```

On voit qu'en cas de changement de valeur, non seulement le PropertyChanged de la propriété est déclenché, mais que, dans le même temps nous levons les RaiseCanExecuteChanged des commandes d'incrémentation et décrémentation afin que leur état *enabled/disabled* soit mis à jour.

On comprend que les méthodes de contrôle des commandes ont été modifiées pour utiliser forkValue en place et lieu de la valeur 5 qui était codée en dur dans les premières versions de l'exemple. De fait, ces contrôles s'adaptent automatiquement à la valeur courante de la fourchette.

Il nous reste à regarder comment le bouton a été programmé en Xaml :

```
<Button Content="Validate"
   Command="{Binding Path=SetFork}"
   CommandParameter="{Binding ElementName=txtNewFork, Path=Text}" />
```

Le binding à la commande SetFork est de même nature que les actions des autres commandes. On trouve juste, en plus, CommandParameter qui ici est bindée non pas à un

élément du Modèle de Vue mais à un élément de l'interface utilisateur, la propriété Text du TextBox que l'utilisateur peut modifier. La technique utilisée est dite « *element binding* » et se trouve dans Silverlight depuis peu de temps en fait. Elle permet de lier par binding les propriétés de deux éléments de l'interface.

EventToCommand

EventToCommand est un behavior proposé dans la librairie « Extras » de MVVM Light. Son utilité : permettre de dépasser les limitations assez drastiques de l'implémentation actuelle de ICommand. Ce qui est la faute du Framework et non de MVVM Light.

En effet, sous Silverlight par exemple, seules les classes héritant de ButtonBase possèdent une propriété Command.

Cela fait tout de même quelques classes puisqu'on compte parmi celles-ci : Button, GridViewColumnHeader, DataGridColumnHeader, DataGridRowHeader, RepeatButton et ToggleButton.

De ces classes en descendent d'autres. De Button naissent EditModeSwitchButton, CalendarButton ou CalendarDayButton. De même, ToggleButton est le père de CheckBox et de RadioButton.

Il y a quand même de quoi s'amuser un peu…

Mais tout cela n'est au final que variante autour d'un même thème : le bouton, et son événement fétiche, le clic.

Désirez-vous déclencher des actions dans votre Modèle de Vue sur le changement de valeur d'un Slider par exemple ? Impossible. Sur le changement de taille d'un objet ? Impossible aussi. Bref il serait bien trop long d'énumérer les centaines d'événements qui existent dans les contrôles disponibles sous WPF et Silverlight et qui diffèrent du Click !

Pour tous ces cas il n'y a pas de réponse « out of the box ».

Heureusement Xaml et ses enfants WPF et Silverlight sont riches de bonnes idées. Sans oublier le travail essentiel de l'équipe qui s'occupe de Expression Blend et qui œuvre depuis toujours pour une plus grande simplicité, pour la possibilité qu'un non informaticien, le Designer, puisse créer des interfaces sans avoir un informaticien assis à ses côtés en permanence ! L'invention géniale qui nous intéresse ici ce sont les **Behavior** (*comportement*). Je ne vais pas entrer dans une présentation des behaviors, cela sortirait largement du cadre de cet article. Mais vous pouvez lire certains de mes billets sur Dot.Blog (faites une recherche du mot « behavior » pour obtenir la liste des billets tournant autour de ce thème).

Si je parle de l'équipe de Blend c'est qu'ils sont aussi à l'origine de EventToCommand, Laurent expliquant qu'il s'est largement inspiré de l'exemple InvokeDataCommand. Cet exemple fait partie d'une suite mis en ligne par l'équipe de Blend (voir mon billet « *Silverlight/Blend : bien plus que des exemples* », à cette adresse : http://www.e-naxos.com/Blog/post.aspx?id=d962d91d-76ab-4eee-9d2e-256eb6c62411)

Donc allons directement au but et regardons en quoi EventToCommand est si fantastique.

Comme c'est un behavior, il est manipulable directement sous Blend en le posant sur l'élément qu'on souhaite ainsi décorer. Il est possible de tout taper à la main si on a une tendance masochiste affirmée ou bien si on n'utilise que Visual Studio puisque ce dernier ne gère pas les behavior (il faut taper le code Xaml à la main). Utilisez Expression Blend, c'est étudié pour…

EventToCommand peut être ajouté à tout objet de type FrameworElement, donc globalement à peu près tout ce qui peut se manipuler en Xaml. Une image, une forme, un Slider, un descendant de ButtonBase (CheckBox, Button…), tout.

EventToCommand se lie à un **événement de l'objet**, n'importe lequel, sous Blend on le choisit par une liste déroulante, en Xaml il faut taper son nom. Chaque fois que cet événement sera déclenché par l'objet source, il sera **transformé en un appel à une commande** ICommand (ou RelayCommand) généralement exposée par le Modèle de Vue.

EventToCommand autorise aussi le passage d'un paramètre comme on l'a vu à la section précédente avec les commandes classiques. Ce paramètre est généralement un binding (le contenu d'un TextBox dans notre précédent exemple) et on créée ce binding avec la propriété CommandParameter de EventToCommand. Si on souhaite passer une valeur « en dur » on utilisera plutôt CommandParameterValue.

La propriété MustToggleIsEnabled peut être bindée à toute propriété booléenne, comme par exemple le IsChecked d'un CheckBox ou plus logiquement le IsEnabled d'un Control. En fonction de la valeur de CanExecute de la commande l'élément d'interface sera ou non actif. Le comportement varie ici entre Silverlight et WPF pour les raisons déjà évoquées dans cet article à propos de la gestion des commandes. Par exemple le IsEnabled fonctionne sur tous les FrameworkElement sous WPF, mais uniquement sur les descendants de Control sous Silverlight.

Enfin, une propriété ajoutée dernièrement est plus qu'essentielle : PassEventArgsToCommand. Quand elle est mise à true la commande bindée à EventToCommand recevra en paramètre les EventArgs de l'événement détourné ce qui permet réellement de traiter ce dernier en bénéficiant de toutes les informations (pour un

MouseMove on peut récupérer la position de la souris par exemple). Si vous laissez cette propriété à false, EventToCommand fonctionne plus sur le mode d'une notification que d'un gestionnaire d'événement. Bien entendu, pour que cela fonctionne le behavior ne doit pas avoir déjà de binding sur CommandParameter ni de valeur dans CommandParameterValue (à vérifier lorsqu'on modifie un code existant). Côté déclaration de la commande dans le Modèle de Vue, on utilisera une signature de type RelayCommand<EventArgs> (en utilisant la classe exacte de l'argument bien entendu).

Avec tout ce que nous avons déjà vu ici, le fonctionnement de EventToCommand ne devrait pas vous poser de problème. Toutefois un petit exemple améliore toujours la compréhension…

Le projet exemple

Le projet exemple numéro 6 montre comment utiliser EventToCommand pour gérer le changement de valeur d'un Slider qui est lié à une commande du Modèle de Vue.

Le Modèle de Vue va ainsi exposer trois propriétés :

- Une commande compatible avec la signature d'un argument d'événement ValueChanged d'un Slider ;
- Une propriété SliderValue qui retournera la valeur entière de la position du Slider (mais qui restera un double) ;
- Une propriété SliderMessage qui retournera un message construit selon la valeur du Slider.

Je passe sur la déclaration des deux propriétés pour arriver directement à celle de la commande dans le Modèle de Vue :

```
private RelayCommand<RoutedPropertyChangedEventArgs<double>> sliderMoveCommand;

public RelayCommand<RoutedPropertyChangedEventArgs<double>> SliderMoveCommand
{ get { return sliderMoveCommand; } }
```

> *Vous pouvez vous demander pourquoi j'utilise un champ privé doublé d'une propriété alors même qu'un champ public ferait l'affaire. C'est une bonne question. Qui trouve une réponse simple : tout simplement parce que le dialogue de binding de Blend (et certainement celui de VS aussi) se base sur les propriétés et non sur les champs. Ainsi, pour voir notre commande sous Blend il est préférable qu'elle soit déclarée en tant que propriété plutôt qu'en simple champ. J'ajouterai aussi que pour respecter le strict minimum du paradigme objet, la moindre des choses est d'honorer le principe d'encapsulation et que jamais un champ ne devrait pouvoir être public. A mon sens c'est même une erreur du langage que de l'autoriser.*

La signature d'un événement est parfois difficile à deviner. Dans le cas présent, comme je ne savais pas quelle classe de EventArgs était utilisée par l'événement ValueChanged du Slider, j'ai utilisé une astuce toute bête : j'ai d'abord créé un événement « normal », j'ai récupéré la signature créée automatiquement par Blend (ou VS) et j'ai effacé le gestionnaire inutile que je venais de créer. Simple mais efficace.

La création de la commande dans le Modèle de Vue est :

```
public MainViewModel()
{
 sliderMoveCommand =
    new RelayCommand<RoutedPropertyChangedEventArgs<double>>(slideMove);
}
```

La méthode slideMove effectue le travail :

```
private void slideMove(RoutedPropertyChangedEventArgs<double> e)
{
  sliderValue =  (int)e.NewValue;
  RaisePropertyChanged("SliderValue");
 if (e.NewValue < 0d) sliderMessage = "Negative Value";
 if (e.NewValue > 0d) sliderMessage = "Positive Value";
 if (Math.Abs((e.NewValue)) > 5) sliderMessage = "Very High " + sliderMessage;
  else if (Math.Abs((e.NewValue)) > 3) sliderMessage = "High " + sliderMessage;
  else if (Math.Abs((e.NewValue)) > 2) sliderMessage = "Normal " + sliderMessage;
  else if (Math.Abs((e.NewValue)) > 1) sliderMessage = "Low " + sliderMessage;
  else sliderMessage = "Very Low " + sliderMessage;
 RaisePropertyChanged("SliderMessage");
 }
```

Ça ne donne pas dans le subtil… mais ce n'est qu'un exemple, ce qui compte c'est que cette méthode du Modèle de Vue, grâce à EventToCommand, va recevoir les arguments de l'événement ValueChanged du Slider *tout en respectant les principes de MVVM*.

On note les RaisePropertyChanged qui permettent le rafraîchissement de la Vue.

Au passage on notera aussi l'efficacité du système de contrôle de cet événement intégré à ViewModelBase. A la première exécution j'ai pu voir ceci :

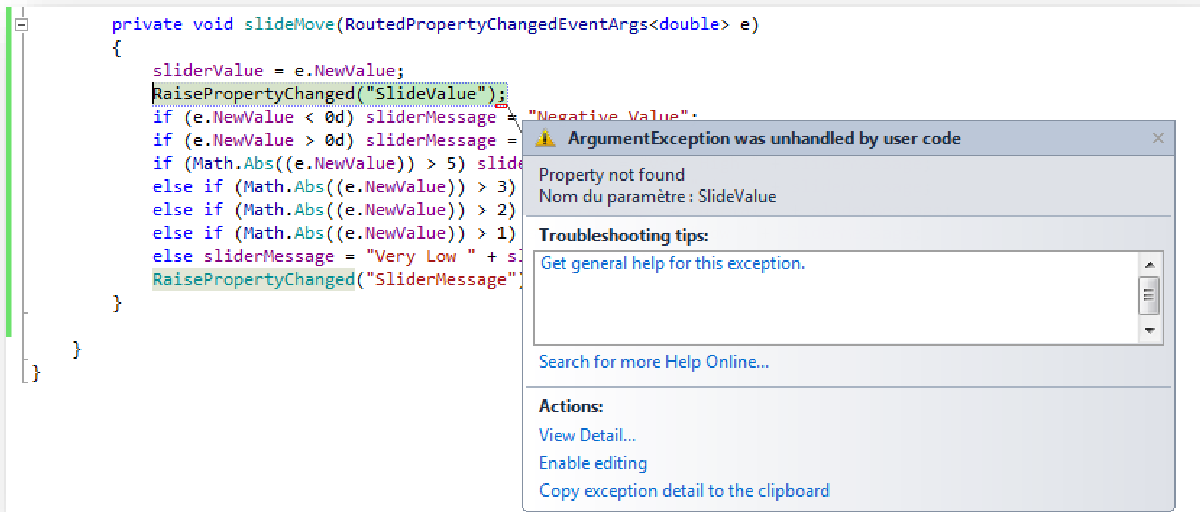


Figure 64 - Le contrôle automatique des noms de propriétés de ViewModelBase

Si… regardez-bien ! Il y a une coquille dans le nom de la propriété passée à RaisePropertyChanged, j'ai écrit « SlideValue » au lieu de « SliderValue ». Une erreur typique avec les PropertyChanged. Grâce au mécanisme de contrôle (uniquement en mode Debug) intégré à ViewModelBase, j'évite un bogue qui, dans une application réelle et plus complexe pourrait n'être découvert que tardivement et difficile à comprendre. Merci MVVM Light !

Dans la pratique, le behavior EventToCommand a été déposé sur le Slider par drag'n drop sous Blend. Ce qui affiche ses propriétés :

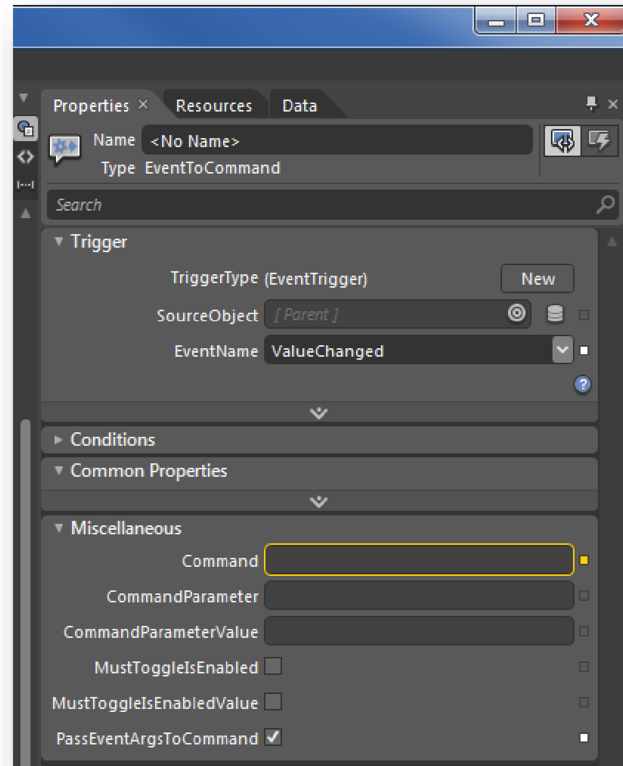


Figure 65 - Exemple 6 - Les propriétés de EventToCommand

On reconnait en haut la partie trigger et en bas la partie commande.

En haut on a lié le trigger à l'événement ValueChanged de l'objet source (le Slider), en bas on voit que la propriété Command est bindée. On sait à quoi grâce au dialogue de binding de Blend :

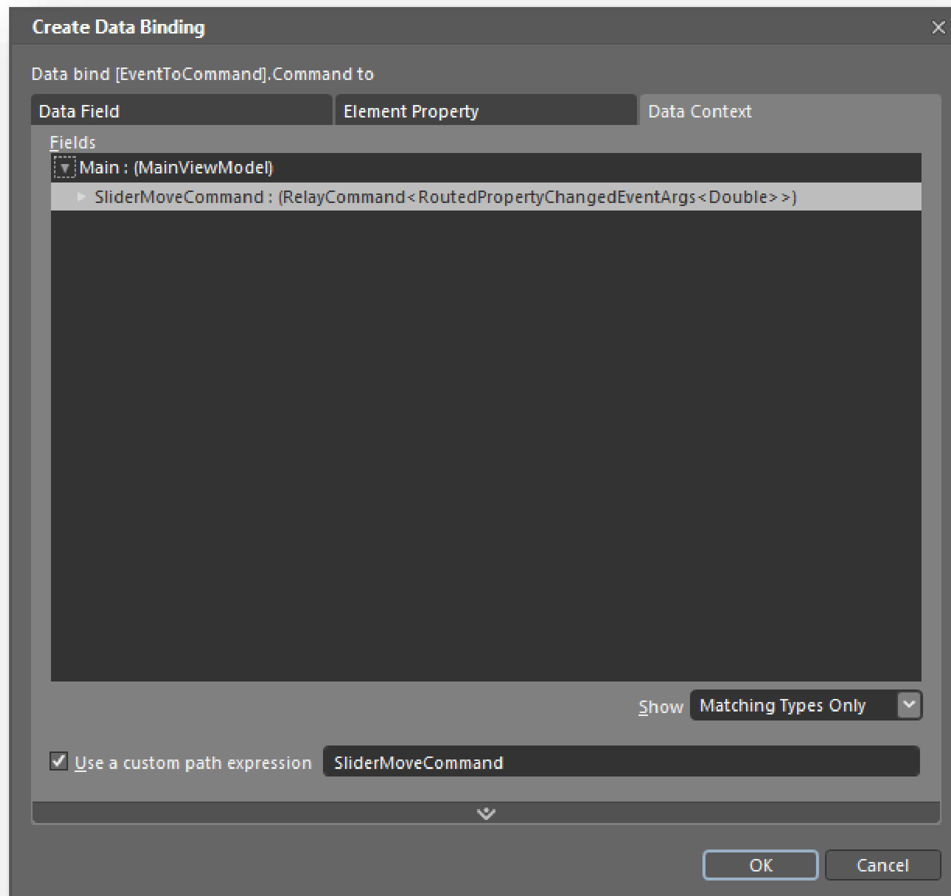


Figure 66 - Exemple 6 - Le binding de la commande sous Blend

Pour les accros du Xaml à la main, voici à quoi ressemble le code que Blend a écrit pour moi :

```
<Slider VerticalAlignment="Center" Minimum="-6" Maximum="6" Margin="5,0">
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="ValueChanged">
          <GalaSoft_MvvmLight_Command:EventToCommand
PassEventArgsToCommand="True"
                          Command="{Binding SliderMoveCommand, Mode=OneWay}"/>
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Slider>
```

L'espace de nom « i » est défini comme cela :

```
xmlns:i="http://schemas.microsoft.com/expression/2010/interactivity"
```

Mais bon, ceux qui tapent du Xaml à la main connaissent tout ça par cœur, pas besoin de trop les aider hein… ☺

> *Philosophons : Laurent le fait remarquer lui-même dans l'un de ses articles, faire descendre les arguments d'un événement d'un objet d'interface vers le Modèle de Vue n'est-ce pas casser un peu la séparation des tiers de MVVM et faire descendre un peu de connaissance de l'interface dans le Modèle de Vue ?*
> *Personnellement je ne vois pas la différence entre un objet argument d'événement et le contenu d'un TextBox… Il s'agit d'informations provenant de l'interface et il faut bien que le Modèle de Vue en prenne connaissance pour produire un travail utile ! Le Modèle de Vue peut prendre connaissance de tout ce qui est nécessaire à son fonctionnement, cela ne viole pas MVVM, même si ce sont les arguments d'un événement d'un objet d'interface. Du moins tant que le Modèle de Vue ignore tout de l'objet qui envoie ces arguments et qu'il n'est en aucun cas lié à celui-ci, ce qui est le cas avec EventToCommand, MVVM est totalement respecté. Laurent, tu peux dormir tranquille !*

Enfin nous pouvons exécuter notre application :

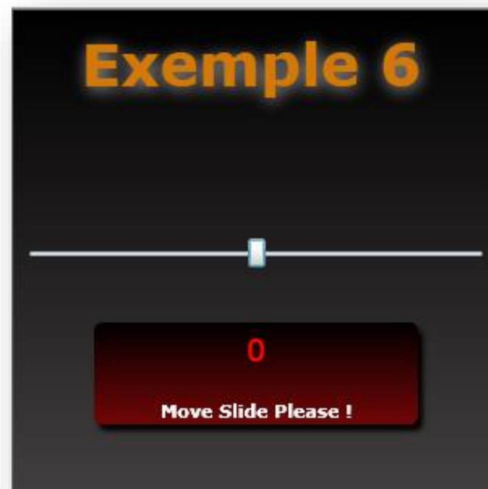


Figure 67 - Exemple 6 : Dans les starting-blocks !

On pourra regretter que les contrôles ne concernent pas le texte qu'on écrit… Le message ci-dessus devrait être « Move The Slider Please! », il manque le « r » de Slider et en anglais le point d'exclamation est collé au dernier mot de la phrase… Etre attentif à ce genre de détail dans une application est essentiel. Les fautes d'orthographe, le non-respect de la ponctuation, tout cela est très important, n'hésitez pas à faire relire vos écrans par quelqu'un d'extérieur en production… l'UX (User Experience) ça commence déjà par des textes bien écrits avant de parler des animations et des effets spéciaux !

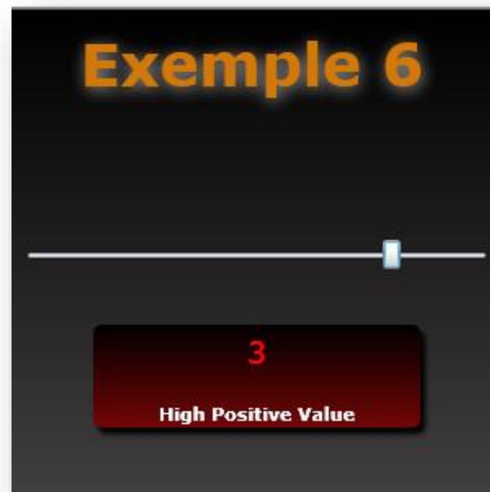


Figure 68 - Exemple 6 : une grande valeur positive

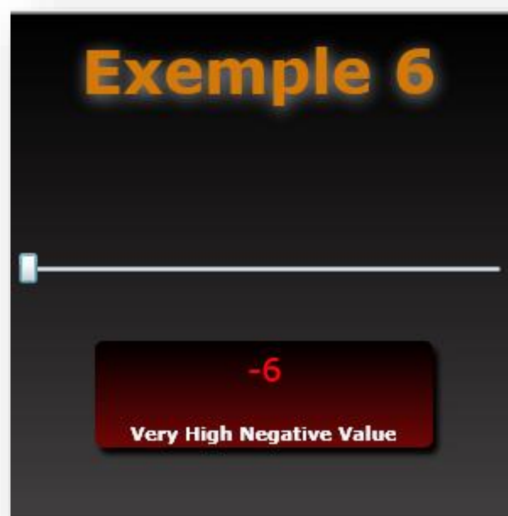


Figure 69 - Exemple 6 : une très grande valeur négative

> *Tout étant relatif, la notion de petite ou très grande valeur dans cet exemple s'entend par rapport aux bornes [-6,+6] que j'ai fixées. On peut considérer que -6 n'est pas une très grande valeur négative si l'unité est le nombre de grains de sable sur une plage landaise, mais si l'unité est le nombre de point qui seront enlevés sur votre permis tout de suite ça fait moins rire !*

Le point sur les commandes

La gestion des commandes est un point vraiment essentiel dans la pattern MVVM, c'est grâce à cette gestion que le Modèle de Vue peut concentrer le fonctionnel et que la Vue peut se débarrasser presque totalement de toute unité de code-behind.

WPF offre un système de commande plus évolué que celui de Silverlight mais, personnellement, je le trouve confus et difficile à utiliser. Peut-être l'ajout le moins bon de tout Xaml depuis sa naissance.

MVVM Light nous offre une réponse simple, portable entre les deux environnements, qui fonctionne aussi bien avec les objets supportant les commandes ICommand qu'avec n'importe quel événement.

C'est simple à mettre à œuvre, c'est pratique, et on peut facilement en maitriser l'implémentation.

Je ne vois aucune raison de se priver d'une telle aide.

CONCLUSION

J'aurais une fois de plus battu le record de longueur avec cet article de plus de 90 pages, mais il me reste la barrière symbolique des 100 pages à franchir, la prochaine fois peut-être ? ☺

MVVM Light est très simple, light. RelayCommand a été écrit par Josh Smith, EventToCommand est inspiré des exemples de Blend, le localisateur de services est à la limite simpliste, le contrôle des noms des propriétés en mode Debug provient de BindableObject du même Josh Smith… Certains pourraient se demander où se trouve l'originalité dans cette librairie.

Justement, toutes ces choses existaient de façon éparses, sans lien les unes avec les autres. Laurent a su en faire un tout cohérent, un outil simplifiant vraiment la mise en œuvre de MVVM sous WPF et Silverlight. Et c'est énorme ! Laurent n'était pas le seul à connaître tous ces bouts de code, chacun d'entre nous les a vu (ou avait la possibilité de les voir), mais seul Laurent en a tiré MVVM Light.

C'est ce qui fait la différence entre le commun des mortels et les inventeurs : tous ont les mêmes choses sous les yeux. Les premiers n'en font rien, les seconds en tirent quelque chose de neuf et d'utile.

Merci à Laurent pour cette librairie simple et utilisable qu'il a eu la bonne idée de publier gratuitement avec le code source !

Il ne vous reste plus qu'à vous en servir, et avec tout ce que je viens d'écrire cela devrait malgré tout vous rendre la tâche plus aisée…

Deux extensions gratuites pour MVVM Light

MVVM Light est l'un des toolkits les plus utilisés pour gérer le pattern MVVM au sein d'une large palette d'applications, de Windows Phone à Windows 8 en passant par Silverlight et WPF. Toutefois le côté "Light" peut laisser apparaitre des besoins non couverts "out of the box". Voici deux extensions gratuites pour combler quelques lacunes du kit…

MVVM LIGHT

Je ne vais pas faire ici l'apologie ou la critique de ce toolkit très utilisé par une foule de développeurs dans le monde. Je l'utilise aussi d'ailleurs assez souvent. Il a ses points forts et ses faiblesses, comme tous les autres kits MVVM.

MVVM Light s'installe depuis les fichiers fournis sur le site de Galasoft ou bien depuis un package Nuget ce qui est encore plus pratique.

Sur d'autres toolkits :

Jounce (MEF et MVVM sous Silverlight)
D'autres références : Comprendre et appliquer MVVM
Il y a largement de quoi lire !

DEUX BESOINS NON COMBLES "OUT OF THE BOX"

MVVM Light est populaire, évolue sans cesse, et fournit une trousse MVVM de base assez complète. Il n'en reste pas moins que MVVM Light… est Light ! C'est l'un des objectifs de ce toolkit, rester le plus léger possible face à des mastodontes comme Prism ou Caliburn, ce qui lui permet de fonctionner aussi bien sous Windows Phone que Silverlight avec le même code.

Toutefois, tout ce qui est Light finit un jour ou l'autre par poser les problèmes de ses avantages : il manque certaines choses…

En dehors de toute critique du toolkit, il faut bien concevoir qu'il y a une limite à toute chose et qu'il lui manque au moins deux cordes à son arc :

- L'asynchronisme
- Le traitement des messages de dialogue avec contexte

On pourrait trouver d'autres "lacunes" à MVVM Light si celui-ci prétendait tout gérer, mais il affiche avec franchise le "Light" dans son nom, on ne peut donc rien lui reprocher sur le fond ni sur ces oublis. Par exemple la gestion des régions, une meilleure intégration de l'aspect navigationnel, une vraie injection de dépendance, etc.

Jounce qui est un autre framework que j'adore utilise une écriture beaucoup plus moderne et offre plus de services que MVVM Light, mais au prix de ne fonctionner, hélas, que sous Silverlight. MVVM Light vise le Light et la compatibilité avec toutes les plateformes, c'est ce qui fait son succès populaire alors que Jounce ne connait qu'un succès d'estime.

Mais les uns et les autres sont des toolkits ouverts et fournis en code source. Il est donc assez facile de les compléter pour les doter de fonctions nouvelles ou d'améliorer celles proposées. L'adoption récente par la version 4 de MVVM Light d'un conteneur d'Inversion de Contrôle est un bon exemple : le code a été construit pour qu'on puisse facilement remplacer SimpleIoC, qui est vraiment .. simple.. par Unity ou MEF par exemple (même si cela n'est pas sans avoir un impact sur la blendabilité à laquelle Laurent Bugnion reste très accroché, position que je soutiens totalement).

L'asynchronisme avec AsyncRelayCommand

Quand je parle d'asynchronisme je ne souhaite pas entrer dans les subtilités de P-Linq ou des Rx (Reactive Extensions) ni même de l'écriture de code multitâche conventionnel, et encore moins du "tout asynchrone" de Windows 8. MVVM light n'a pas vocation à prendre en charge cette partie du développement d'une application, d'autres librairies s'en chargent.
Je parle plutôt d'une chose simple, d'un besoin fréquent et non exotique qui oblige à faire intervenir de l'asynchronisme dans les mécanismes MVVM du toolkit : celui de pouvoir exécuter une commande (RelayCommand) via une tâche de fond, automatiquement, le tout en ayant une gestion tout aussi automatique du CanExecute afin que la commande se désactive seule et se réactive en fin de tâche (ce qui rend l'UX plus sure et plus réactive).

Les exemples de ce besoin sont légions : interroger des données distantes (ou non), lancer une impression, la génération d'un fichier d'exportation, faire une importation de données, etc…

Or, RelayCommand ne sait gérer que des actions directes.

L'idée de ma première extension est donc de fournir un équivalent à RelayCommand, totalement compatible et fonctionnant de la même façon, mais dont l'action est automatiquement transformée en tâche de fond. Le tout, comme je le disais, en gérant automatiquement l'état de CanExecute afin que les éléments d'UI connectés à la commande (des boutons en général) soient désactivés durant l'exécution de la tâche de fond et réactivés dès qu'elle se termine.

La gestion des erreurs doit aussi être prise en compte.

De cette idée est née AsynRelayCommand. Le nom n'est pas très original et d'autres implémentations existent d'ailleurs sous ce même nom mais il me semble tellement parlant que vouloir à tout prix se démarquer l'aurait été au détriment de l'évidence du sens.

Le code de base est emprunté directement à la classe RelayCommand de MVMV Light 4. Quitte à être compatible, autant que cela ne soit pas hypocrite !

En revanche le code a été largement complété et modifié pour gérer tout l'aspect asynchrone bien évidemment absent du code original.

Le code à télécharger (en fin d'article) contient un programme de test qui montre très simplement comment utiliser cette commande spéciale.

Les commandes de type AsyncRelayCommand sont génériques par défaut, si on ne fait pas usage du paramètre il suffit de passer un object comme type. Une déclaration sera ainsi du type :

```
public AsyncRelayCommand<object> BackgroundTaskCommand { get; private set; }
...
BackgroundTaskCommand = new AsyncRelayCommand<object>(longtask, canExecute,
onError, onCompleted);
```

Le paramètre générique permet de passer (souvent par binding) des informations complémentaires à la commande. On peut passer "object" comme cela est fait ici si on n'utilise pas de paramètre. Une version non générique supplémentaire serait un petit plus je l'avoue.

Les quatre paramètres passés ensuite sont les différents actions ou prédicats permettant de gérer la commande. La déclaration de AsyncRelayCommand précise mieux les choses :

```
public AsyncRelayCommand(Action<T> execute, Func<T, bool> canExecute,
                         Action<Exception> onError, Action<object> onCompleted)
```

Le premier paramètre est l'action qui sera exécutée en tâche de fond par un BackgroundWorker, classe qui a l'avantage d'exister dans les différentes versions de .NET visées. L'action reçoit en argument le paramètre optionnel avec le type déclaré.

Le second paramètre est un prédicat qui évalue la possibilité d'exécuter la commande. Bien que laissé à la disposition du développeur pour y ajouter ce qu'il souhaite (et par souci de compatibilité avec RelayCommand) l'état CanExecute est modifié automatiquement par AsyncRelayCommand suivant l'état d'exécution de la tâche de fond.

Le troisième paramètre est une action qui reçoit en paramètre une exception. Ce callback, comme tous les autres (sauf l'action à exécuter) est optionnel. Si on définit une expression lambda ou une méthode pour le gérer le code sera appelé dans le cas où une exception se déclenche durant l'exécution de la tâche de fond.

Enfin, le quatrième paramètre permet de définition une dernière action qui sera appelée lorsque la tâche de fond se terminera (avec ou sans erreur). L'application peut ainsi décider d'activer certains menus, de changer de page, etc…

Le reste n'est qu'une question de Binding. Une fois la commande définie dans le VM, on peut binder par exemple la propriété Command d'un Button et, optionnellement, binder le paramètre à un élément de la page en cours. Ce paramètre sera reçu aussi bien par l'action à exécuter que par la méthode CanExecute et l'action de fin de tâche.

Dans l'exemple fourni, les quatre paramètres sont passés comme des méthodes écrites séparément pour clarifier les choses. On peut passer des expressions lambda si on le désire.

L'exemple utilise une liste de chaines de caractères comme stock de messages, chaque callback écrivant dans ce "log" afin de pouvoir facilement tracer dans une listbox l'enchainement des actions.

Sous WPF on peut voir que CanExecute est exécuté très souvent. Cela n'est pas le cas sous d'autres moutures C#/Xaml. Toutefois le plus important, c'est à dire la désactivation de la commande lorsque la tâche commence et sa réactivation lorsqu'elle s'arrête sont pris en charge automatiquement par le code de AsynRelayCommand.

On notera que AsyncRelayCommand propose quelques propriétés et méthodes supplémentaires, comme par exemple IsWorking qui indique si la tâche de fond est en train

de tourner ou non, ou CancelWork() qui permet d'interrompre la tâche, et ce, en plus des méthodes de l'interface ICommand (comme Execute()).

DialogMessageWithContext

Voici le second problème non réglé totalement par MVVM Light : un message de type demande de dialogue pouvant transporter un contexte utilisateur.

Certes on entre ici dans des besoins qui peuvent paraitre exotiques pour celui qui ne pratique pas souvent MVVM mais qui, je peux vous l'assurer, n'ont rien de bien extraordinaire.

Je ne réexpliquerai pas ici le mécanisme des messages de MVVM Light ni leur utilité. Encore moins je n'entrerai dans les détails du message DialogMessage qui permet à un ViewModel de demander l'affichage d'un dialogue modal et de recevoir la réponse via un callback, une vue (n'importe la quelle mais généralement le Shell) se chargeant d'afficher le dialogue et d'appeler le callback pour retourner le résultat du dialogue (oui / non, ok / cancel). MVVM est ainsi sauvé : un ViewModel n'intervient pas directement dans l'UI alors même qu'il a parfois de poser des questions directement à l'utilisateur.

C'est un peu "tordu" comme logique, j'en conviens aisément. L'application de MVVM n'est pas toujours aussi idyllique qu'on le voudrait (certains de mes billets ou articles cités plus haut sont la preuve de mon scepticisme sur certains points d'ailleurs).

Quoi qu'il en soit, si vous connaissez DialogMessage vous savez que vos VM peuvent poser des questions à l'utilisateur via un mécanisme mêlant messagerie et callback et faisant intervenir une Vue jouant le rôle de "petit rapporteur".

Quel est le problème ?

Le mieux dans de tels cas est de partir d'un scénario.

Prenons une application affichant une View avec une liste de données. L'utilisateur a la possibilité de supprimer chaque donnée. Mais le mécanisme ne peut utiliser les automatismes éventuellement existants car le ViewModel veut demander une confirmation à l'utilisateur.

Certaines personnes tentent de régler la question en écrivant du code-behind : le bouton "supprimer" est géré dans ce dernier qui pose la question et si elle positive ce code appelle alors la méthode Execute de la commande.

Que dire… C'est "cracra". C'est du code spaghetti. Ni satisfaisant techniquement, ni satisfaisant intellectuellement. Pauvre, risqué (le code s'éclate sur la vue, le code-behind, le ViewModel, le Model et que sais-je d'autre). Bref je déconseille vivement cette approche qui fait bricoleur.

Dans le respect de MVVM, et avec MVVM Light, le ViewModel expose une commande "Supprimer" qui est bindée à un bouton sur la Vue. Lorsque l'utilisateur clique sur ce bouton, le ViewModel sait quel est l'item à détruire (car il piste par exemple le SelectedItem de la liste par un binding).

- Le ViewModel utilise alors un DialogMessage pour demander la confirmation à l'utilisateur. C'est à dire que le cycle complet va être le suivant :
- Emission du message DialogMessage avec un texte du type "confirmez la suppression de l'item xxx" avec les boutons oui/non affichés, ne pas oublier de passer le callback qui sera appelé lorssque l'utilisateur aura répondu.
- Réception du message par la Vue. Elle le traite "bêtement" elle joue juste le rôle de "relai", de "répéteur" pour afficher un vrai dialogue modal, elle attend la réponse, et la transmet au demandeur initial en utilisant le callback passé dans le message. En paramètre de ce callback sera passé la valeur réponse de l'utilisateur.
- Le ViewModel voit son callback appelé après un "certain temps" (depuis le lancement du message initial) et doit maintenant traiter la demande de suppression.

Dans un monde idéal, tout cela va très vite, et l'utilisateur n'a pas le temps de cliquer ailleurs entre l'envoi du message par le ViewModel et la réception de la réponse par le callback de ce dernier. De même le ViewModel est un automate très sage mono tâche ignorant tout de la programmation asynchrone.

Seulement voilà, notre monde n'est pas idéal. Encore moins pour une application tournant dans des environnements multitâches préemptifs où l'asynchronisme règne désormais en maitre absolu.

Je ne vais pas lister tout ce qui pourrait se passer entre ces deux moments cruciaux (envoi du message par le ViewModel et réception de la réponse utilisateur par le callback), mais il peut se passer des tas de choses. Cette seule possibilité rend le processus non déterministe. Et ce qui n'est pas déterministe ne peut pas se programmer proprement avec les langages et les OS actuels. Donc ça boguera un jour ou l'autre et çà sera très difficile de savoir pourquoi, où et comment.

La solution ? Elle tient à un pauvre petit paramètre de type objet. Et c'est DialogMessageWithContext qui l'ajoute.

Je ne vais pas reprendre tout le cycle expliqué plus haut, faisons juste preuve d'un peu d'imagination en ajoutant ce qui manque :

Le ViewModel fait toujours la demande de message, mais il utilise DialogMessageWithContext auquel il passe un "context" en paramètre. Ce paramètre "context" contient tout ce qui est nécessaire pour que le callback sache exactement quel item il faudra supprimer (ou traiter de n'importe quel façon) lorsque la réponse utilisateur arrivera.

Lorsque le callback est appelé par la vue, le "contexte" est tout simplement retourné à l'envoyeur. Rien de plus. Mais maintenant le callback dispose des informations précises lui permettant de traiter la demande sans risque d'erreur peut importe ce qui s'est passé entre le moment de la demande de message et le traitement par le callback de la réponse.

La démonstration montre comment cela est exploitable :

Il existe une commande pour supprimer l'item en cours de sélection dans la liste. Cette commande est bindée à la propriété Command d'un bouton. La propriété Parameter de la commande est aussi bindée mais sur un Element binding pointant l'item sélectionné dans la liste.

De ce fait, lorsque la commande est activée par l'utilisateur, le ViewModel reçoit à la fois l'ordre d'exécuter la commande mais aussi, en paramètre, l'item sélectionné (ici son simple numéro d'ordre dans la liste, c'est une démo…).

D'une part lorsqu'il envoie la demande de message il peut construire une chaine précisant le nom, le code, ou toute information qui permet à l'utilisateur de savoir exactement sur quoi porte la confirmation de suppression (ou de traitement quelconque), mais surtout, lorsque le callback recevra la réponse il récupèrera ce précieux contexte qui lui permettra de traiter la demande sans aucune ambigüité…

Le contexte peut être un simple binding sur le SelectItem ou même le SelectedIndex d'une liste comme dan la démo, ou être un objet plus complexe. Dans certaines applications Silverlight je me suis même servi de cette technique pour transmettre le contexte RIA Service ouvert à l'instant de la demande pour que le callback puisse disposer non seulement de l'item à traiter mais aussi de la connexion ouverte permettant de le faire. Toutes les options sont possibles, cela dépend de l'application et de son fonctionnement.

CONCLUSION

On le voit clairement ici, MVVM est un pattern délicat dès qu'on sort des sempiternelles présentations à deux cents que tout le monde peut écrire. Dès qu'on applique le pattern

dans une vraie application et quel que soit le toolkit choisi, des problèmes plus ou moins sérieux se posent.

Certains finissent par s'autoriser des entorses à MVVM, d'autres buttent et ne s'en sortent pas.

Les plus persévérants arrivent néanmoins à trouver des réponses qui respectent MVVM sans créer non plus des usines à gaz qui ne peuvent pas être maintenues.

Par cette modeste contribution j'espère avoir montré à tous qu'on peut se sortir des mauvaises passes dans lesquels MVVM nous envoie parfois tout en respectant le pattern et sans complexifier à outrance le code.

Dans tous les cas vous avez gagné deux extensions gratuites à MVVM Light

Le petit disclaimer pour terminer : le code offert l'est uniquement à titre d'exemple, aucune garantie n'est donnée et vous l'utiliserez sous votre seule responsabilité notamment dans des environnements de production. Si vous ne comprenez pas le code, si vous n'êtes pas capable de le déboguer, ne l'utilisez pas en dehors de tests. Enfin, ce code est gratuit mais je conserve mes copyrights, aucune distribution ni publication ni aucune exploitation par aucun moyen ne peut en être fait sans mon autorisation expresse.

Voilà, ça c'est fait.

Amusez-vous bien avec MVVM Light et ses nouvelles extensions !

OD MVVM Light Extensions

MVVM : simplifier le circuit des messages

J'ai abordé MVVM de milles façons ici, notamment sous l'angle d'un questionnement sur la nature même de ce pattern et les complications qu'il entraine. Dans cette lignée voici une courte réflexion sur la simplification du circuit des messages.

MVVM ET LA MESSAGERIE

Quel que soit la boite à outils qu'on utilise, Jounce, MVVM Light, Caliburn, ... il existe toujours quelque part une circuiterie, un bout de plomberie qui joue le rôle de messagerie.

Je me suis déjà interrogé sur Dot.Blog sur la nature même de pattern qu'on pouvait ou non attribuer à MVVM tant sa définition est bien floue comparée aux “vrais” patterns comme ceux du Gang Of Four. Ainsi, MVVM n'a pas de définition officielle. On en trouve des traces,

des interprétations sur certains blogs, c'est tout. Après il reste les boites à outils aidant à mettre en œuvre MVVM, elles sont toutes différentes, se basent sur des principes différents, voire une terminologie différente (la messagerie par exemple qui porte à chaque fois un nom particulier).

Or MVVM, "de base", "out of the box" on pourrait dire, ne parle que de séparation UI/Code et de l'utilisation du Binding comme clé de voute de l'ensemble. Nulle part il n'est fait mention d'injection de dépendance ou de messagerie par exemple.

C'est ainsi que chaque librairie de code propose "sa" propre approche du problème. Jounce utilise MEF pour faire de l'injection de dépendance sous Silverlight, MVVM Light ne le propose pas, Caliburn en fait, autrement. MVVM Light propose une messagerie (classe Messenger) pour gérer le découplage entre les tiers que fait apparaitre le pattern, Caliburn ou Jounce propose un "EventAggregator" qui joue le même rôle.

Etc. Etc. On pourrait trouver comme cela cent différences dans la façon d'interpréter MVVM.

Mais une chose revient malgré tout, sous un nom ou un autre, construite selon tel ou tel autre principe, c'est la messagerie.

Le cas d'école est celui de l'ouverture d'un dialogue, prenons le plus simple, un message avertissant l'utilisateur que des données sont arrivées ou une erreur d'exécution. Un message sans retour.

LE CIRCUIT DU MESSAGE DE BASE

Dans ce cas le plus simple, c'est le ViewModel qui, détectant la condition (qui peut être une exception par exemple) souhaite afficher une boite de dialogue.

Hélas, cela lui est interdit par MVVM, seule un vue peut appeler une autre vue. Et une boite de dialogue est une vue.

En mode '"classique", le code se contenterait d'un simple "MessageBox.Show("coucou"). Exemple trivial sous Windows Forms notamment.

En mode MVVM, c'est tout un tralala que d'afficher ce pauvre bout de texte !

Pour ne pas "violer" MVVM, seule la Vue peut afficher le dialogue. Mais comme seul le ViewModel peut prendre des décisions, c'est de lui que viendra à la fois l'initiative de cet affichage et son contenu.

Dès lors il faut bien que le ViewModel demande à la Vue d'effectuer elle-même le fameux "MessageBox.Show(message)" (ou tout équivalent, vous l'avez compris).

Et sans messagerie (ou équivalent), cela n'est pas possible. Tout simplement.

Il va donc falloir créer un identifiant de message, créer peut-être une classe spécifique pour ce message, puis créer une instance, la renseigner, et transmettre, via la messagerie, ce message depuis le VIewModel vers la Vue. Cette dernière devra s'être enregistrée d'une façon ou d'une autre pour recevoir ce message, le filtrer (en fonction de son type, de son identifiant, du contexte...), l'interpréter (par exemple extraire le message à afficher, créer une instance de la classe affichant les messages, etc) et enfin ordonner à la vue dialogue de s'afficher...

On est là dans des circonvolutions qui peuvent paraitre délirantes à beaucoup de développeurs.

Qu'ils se rassurent je pense la même chose ! Et c'est ceux qui appliquent de telles recettes alambiquées sans se questionner qui, à mon sens, ont un problème.

Mais ici j'ai pris l'exemple le plus simple, il y a pire...

LE CIRCUIT D'UN MESSAGE AVEC REPONSE

Là, c'est le "22 à Asnières", "Ubu Roi" ou toute autre référence du même type qui vous plaira.

Car en effet, l'interdiction est à double sens : nous avons vu que le ViewModel ne peut pas afficher le dialogue, mais bien entendu la Vue ne saurait exécuter quoi que ce soit, et encore moins traiter une réponse au dialogue !

Plus "amusant", la Vue ne connait pas son ViewModel, c'est interdit.

Voici notre Vue qui reçoit, en quelque sorte par voie divine (la messagerie est à MVVM ce que le Saint Esprit est au catholicisme), un "message à afficher". Ce qu'elle fera comme indiqué plus haut. Mais la voici maintenant avec sur les bras une réponse !

Que faire de cette réponse ?

Rien. Elle ne peut rien en faire, je vous l'ai dit plus haut.

Faire un appel du genre LeViewModel.Réponse=laRéponse n'est pas même envisageable en rêve.

Reste la messagerie...

Ce coup-ci c’est le Vue qui est émettrice, qui va devoir confectionner un message (ayant son propre identifiant, sa propre classe éventuelle, différente du message de demande de dialogue d’ailleurs) et l’envoyer “dans les airs”. Et c’est au ViewModel de prendre le rôle de récepteur ce qui lui imposera de s’être lui-même enregistré comme récepteur du message, de le filtrer, l’interpréter et enfin d’exécuter le code que la fameuse réponse doit déclencher (s’il n’y a rien à exécuter, la réponse ne sert à rien, logique).

Il existe des variantes amusantes.

Par exemple MVVM Light propose des messages avec callback (de type Action<>). Du coup, le VIewModel peut envoyer le code à exécuter sur la réponse dans le message original demandant l’affichage du dialogue, la Vue n’aura qu’à exécuter le code du callback en lui passant en paramètre la valeur de retour du dialogue... Cela ne viole pas MVVM puisque le callback est envoyé au runtime et ne réclame pas une connaissance de la Vue par le ViewModel ni l’inverse.
Je ne parle pas de message un peu plus complexe à traiter, comme la simple confirmation de suppression d’une donnée par exemple. Car dans le fameux callback, situé dans le ViewModel mais appelé depuis la Vue, il faudra certainement récupérer un “contexte”, comme le contexte RIA services sous Silverlight par exemple. Et si plusieurs messages arrivent en même temps ce contexte devra avoir été préparé par le ViewModel à l’envoi du message original pour contenir tout ce qui est nécessaire au traitement de la réponse par le callback sans se mélanger les pinceaux.

On notera que cela n’existe pas de base (j’ai par exemple créé des extensions à MVVM Light dont un message avec callback et contexte, code qu’on trouvera gratuitement dans les posts traitant de cette librairie).

C’est à ce genre de délire que celui qui suit MVVM est confronté en permanence, et ce qui fait dire à certains que ce n’est pas raisonnable.

Je ne peux que les comprendre et abonder dans leur sens, ce n’est pas raisonnable en effet.

RACCOURCIR LE CIRCUIT ?

Cela m’apparait comme une nécessité, simplement pour restaurer un peu de santé mentale dans ce montage délirant.

La vraie question est de savoir ce qu’il est possible de faire, sachant qu’aucune librairie ne propose vraiment de réponse valable à ce jeu de méli-mélo de messages ...

Certes on pourrait “violer” MVVM.

Mais je n’aime pas le principe.

MVVM est malgré tout une bonne idée, un principe intéressant (je parle de principe et non de pattern d’ailleurs). Certaines de ses “lois” comme la séparation absolue UI/Code me semblent elles très raisonnables et justifier de se compliquer un peu la vie. *Mais un peu seulement*.

UNE QUESTION DE POINT DE VUE

Dans le circuit d’un message avec réponse exposé plus haut je me suis placé dans le cas le plus strict de l’interprétation de MVVM et dans le cadre des librairies créées pour en “simplifier” la mise en œuvre.

N’y-t-il pas moyen de faire plus court, plus simple tout en restant dans les rails de ces librairies et en respectant MVVM ?

Tout est une question de point de vue... Le flou artistique autour de MVVM et de son interprétation laisse la porte ouverte à des variantes, voire des ruses.

DEUX OPTIONS

Par exemple, si une Vue doit déclencher un traitement dans son ViewModel il existe au moins deux options.

La première est radicale. Elle consiste à dire que MVVM c’est génial mais que certaines de ces justifications n’ont pas de sens dans de nombreux projets. Le plus parlant des exemples est à ce titre la possibilité de pouvoir “substituer” un ViewModel par un autre sans que la Vue .. ne le voit.

Dans le principe cela est une application pure et dure de la séparation UI/Code, et comme je suis d’accord avec cette loi je devrais l’être avec cette application de celle-ci.

En fait je me réserve le droit de penser librement et d’interpréter les choses autrement. A l’heure actuelle, et après des dizaines de projets utilisant MVVM réalisés par moi-même, mes collaborateurs ou mes clients, pas une seule fois je n’ai rencontré le cas où, comme cela, on s’amusait à mettre un nouveau ViewModel à la place d’un autre existant.

Si un ViewModel pose des problèmes on les règle. S’il n’expose pas une propriété ou une action qui fait défaut, on l’ajoute, tout simplement. On n’en fabrique pas un autre qu’on substituerait au runtime, à la “sournois” dirais-je presque, sans que la Vue ne le sache.

De plus, si la séparation UI/Code est techniquement un bon principe, le divorce forcé entre Vue et ViewModel est moins logique. Après tout, et là encore après être nourri par l'expérience, jamais je n'ai rencontré de Vues créées "comme ça" sans avoir en tête le ViewModel qui va avec, et encore moins l'inverse.

On peut s'amuser à découpler techniquement autant qu'on veut la Vue de son ViewModel elle lui reste conceptuellement totalement inféodée !
MVVM Light utilise le principe du ViewModelLocator pour créer une (autre) séparation, une indirection entre la référence à un ViewModel et l'instance qui est réellement derrière. Dans un tel cas, puisque le ViewLocator effectue déjà la séparation et l'abstraction, la Vue peut très bien effectuer un appel direct au ViewModel qui lui est connecté en passant par le ViewModelLocator !
Depuis la Vue on pourra donc écrire ViewModelLocator.Main.FaitCeci(argument);
Adios la messagerie !

Bien que j'ai démontré de façon irréfutable que le séparation UI/Code était respectée (puisque effectuée par le ViewModelLocator) je sais que quelques esprits retors, voire chagrins, trouveront ici à discutailler.

C'est pour cela que je parlais de deux options. La première, vous la connaissez, je viens juste d'en parler.

Et la seconde ?

Pour satisfaire les plus orthodoxes d'entre nous, je propose d'utiliser une interface.

Au lieu que le ViewModelLocator n'expose des instances de classes, il suffit qu'il expose des interfaces. C'est un peu plus fastidieux car cela implique de créer une interface par ViewModel différent, puis de l'implémenter. Certaines librairies MVVM l'impose quasiment d'ailleurs.

Mais un ViewModel doit rester sobre et finalement même s'il contient beaucoup de code il n'expose à la Vue que peut de choses : des données et des commandes. Les regrouper dans une interface est chose aisée.

Dès lors on peut de nouveau écrire depuis la Vue
: ViewModelLocator.Main.FaitCeci(argument), rien ne change... toujours aussi facile et direct. Sauf que "Main" au lieu d'être déclaré comme "MainViewModel Main { get; set; }" est déclaré comme "IMainViewModel Main {get;set;}".
Sous MVVM Light la variable "Main" est statique et le getter s'occupe de créer l'instance si elle n'existe pas (dans l'esprit du pattern Singleton).

FACILITER LE DIALOGUE

Grâce à cette approche il devient possible d'effectuer des appels simples au ViewModel depuis la Vue sans briser MVVM.

Avec la première option, on admet qu'on ne pourra pas changer à la volée le ViewModel sauf à ce qu'il supporte les mêmes méthodes (mais cela me semble de toute façon une obligation).

Avec la seconde option (l'interface) on transfère les enquiquinements des messages complexes dans l'écriture, une fois pour toute, d'une interface pour chaque ViewModel ce qui n'est pas très difficile à faire, juste fastidieux si le ViewModel expose de nombreuses données ou commandes. Mais ce prix est payé une fois et vaut largement le fait de se passer de toute la circuiterie diabolique que j'ai décrite plus haut.

SIMPLIFIER ENCORE ?

C'est à mon sens possible. Toujours en changeant un peu de point de vue.

Prenons l'exemple d'un dialogue d'ouverture de fichier.

Chemin classique d'ouverture de fichier

Dans le chemin classique, le ViewModel expose une commande "OuvrirFichierCommand". Cette commande est bindée à un bouton dans la Vue.

La commande ne pouvant afficher le dialogue dans le ViewModel, elle créera un message qu'elle transmettra à la messagerie en espérant que la Vue l'attrape.

Cette dernière s'est enregistrée pour recevoir ce message, puis elle le filtre, l'interprète, l'exécute. Enfin, l'utilisateur peut choisir le nom du fichier à ouvrir...

L'utilisateur valide son choix. La Vue se retrouve avec une réponse dont elle ne sait que faire, le nom du fichier.

Soit le message original est un message avec Callback, et la Vue va alors appeler ce Callback en passant en paramètre le nom du fichier. Charge au ViewModel de récupérer la valeur et de déclencher le traitement (ouverture et lecture du fichier dans ce cas par exemple).

Soit le message original n'a pas de Callback (parce que la librairie choisie ne le gère pas ou autrement par exemple) et la Vue devra alors à son tour émettre un message contenant le nom du fichier en espérant que le ViewModel l'attrape. Ce dernier devra s'être enregistré pour réceptionner le message, etc, etc, jusqu'à pouvoir effectuer le traitement sur le fichier.

C’est très compliqué et, expérience à l’appui, ces jeux de messages deviennent vite des horreurs en termes de maintenance.

Chemin rusé

Puisque le dialogue d’ouverture de fichier est une Vue et que les Vues ont le droit d’ouvrir d’autres Vues, pourquoi se compliquer la vie inutilement avec tous les messages évoqués plus haut ?

Le bouton d’ouverture de fichier n’est ainsi plus bindé à une commande du ViewModel. Ce bouton a, oui c’est diabolique, un bout de code code-behind qui ouvre directement le dialogue de sélection de fichier. Une ligne de code qui n’est pas lié au code métier, juste du code d’interface permettant à la Vue d’en appeler une autre. *Totalement et rigoureusement conforme à MVVM*.
Ce même code-behind va traiter le retour du dialogue, de toute façon même dans le cas MVVM complexe vu plus haut cette tâche lui incombait. Il s’agit en général de tester si le résultat du dialogue est “ok” ou non. Rien qui touche le code métier.

Et maintenant, que faire de la réponse ? Utiliser le ViewModelLocator comme expliqué plus haut pour attaquer directement soit une action soit une propriété du ViewModel en lui fournissant le nom du fichier à traiter.
C’est fini.

Un bouton, un bout de code-behind de deux lignes, l’astuce du ViewModelLocator et c’est fini. **Pas un seul message** ! Pas de trucs étranges pour vérifier qu’on est sur le thread de l’UI et faire des invocations au travers d’un dispatcher dans le cas contraire. Rien de tout cela.
Et le tout en respectant à la lettre MVVM.

L’APPROCHE DE L’INJECTION DE DEPENDANCE

En remettant en page cet article je me suis dit qu’il fallait au moins ajouter quelques lignes sur d’autres procédés décrit d’ailleurs dans Dot.Blog comme les conteneurs d’IoC (Inversion de Contrôle) et la technique d’Injection de Dépendance. Je renvoie le lecteur aux articles traitant de ces techniques particuluères qui permettent de conserver une totale isolation entre les différents tiers de MVVM, notamment entre les vues et les VM mais aussi entre ces mêmes VM et les modèles voire les « services » proposés par l’application.

L’affichage de message, la mise à disposition dun ViewLocator, l’accès aux données, tout cela peut s’effectuer via un conteneur d’IoC. L’approche est au départ moins simple que celle présentée ici mais elle est plus satisfaisante.

CONCLUSION

MVVM n’est pas un pattern définit avec la précision requise pour prétendre à ce rang de “pattern”. C’est juste une idée, une bonne idée, mais floue et surtout dont la partie “mise en œuvre” est totalement ouverte, chacun se débrouille comme il peut. C’est ce que chaque librairie MVVM tente de faire d’ailleurs...

Toutefois, si un “truc” aussi mal bouclé que MVVM fait couler autant d’encre, et même si de nos jours elle n’est plus que virtuelle, c’est parce que MVVM pose des principes qui sont bons et raisonnables.

L’idée n’est pas mauvaise. Ce qui lui manque c’est d’être définie comme un véritable pattern.

C’est justement à force de s’y confronter qu’on pourra un jour peut-être forcer une définition claire, nette, prenant en compte les conséquences, les avantages et désavantages, les impacts divers que son utilisation implique.

En sachant changer la caméra de position on dévoile souvent une autre scène. Choisir des angles de vue plus lisibles ne coute pas grand chose.

Je vous ai proposé ici de régler l’un des problèmes les plus pénible de MVVM en se débarrassant totalement de la messagerie. Juste en modifiant le point de vue habituel sur MVVM.

On doit pouvoir faire beaucoup mieux encore avec cette idée. Il faut juste penser à placer la caméra correctement...

MVVM : Gérer les données de Design de façon propre

De Silverlight à WinRT en passant les Smartphones, sous Windows ou d’autres OS, les patterns de type MVVM sont devenues indispensables. Toutefois gérer des données de Design pour faciliter la création des UI est souvent mal géré ou oublié. Cela est aussi essentiel pourtant...

MVVM “GENERIQUE”

Il existe de nombreux frameworks pour gérer la séparation UI/Code, qu’ils soient basés sur MVVM, MVC ou d’autres patterns. C’est pourquoi je vais faire abstraction ici de toutes ces subtilités pour discuter des données de design sans faire appel à aucun de ces frameworks (même si l’article suit la logique de MVVM).

C’est à la fois attirer votre attention sur le principe de fournir des données de design qui m’intéresse ici et la mise en œuvre de moyens simples réutilisables dans une multitude de contextes (quitte à les adapter).

SILVERLIGHT

L’exemple pourrait être fait avec WinRT, WPF, WP7 ou WP8 ou même d’autres technologies, cela n’a pas grande importance. Comme il faut faire un choix et que j’aime toujours autant Silverlight c’est donc au travers d’une application de ce type que nous allons voir comment créer des données de design. Bien que sans framework spécialisé, l’exemple suivra la logique de MVVM, ce qui est adaptable à d’autres patterns du même type.

PRINCIPES DE BASE DES DONNEES DE DESIGN

Tout d’abord j’aimerais insister sur la nécessité de proposer des données de design. C’est vraiment quelque chose d’essentiel pour l’infographiste ou pour l’intégrateur qui devra marier les jolies présentations avec le code des développeurs. Sans données de design il y a fort à parier que c’est lors des premières exécutions (si on a de la chance) qu’on s’apercevra que telles zones est trop longue ou trop courte, trop haute ou pas assez... C’est là aussi qu’on verra que telle fonte qu’on croyait géniale ne passe visuellement pas lorsque toute la fiche est remplie d’information. Et plein d’autres choses qui ne peuvent se voir que s’il y a quelque chose à voir !

Les données de design doivent être :

- Disponibles au moment du design, c’est à dire sans avoir besoin d’exécuter l’application
- Utilisables sous Visual Studio et Expression Blend (pour la plateforme Windows)
- Représentatives du contenu réel
- Suffisamment “passe partout” pour que l’œil lors du design ou de l’intégration ne soit pas “attrapé” par le texte de design mais qu’il reste concentré sur le design lui-même
- Discrète au runtime : c’est à dire que le code les générant ne doit pas être intégré à l’exécutable final
- Conçues pour le design, par pour des tests de montée en charge.

Il ne s’agit que des grands principes de base. Mais ils sont très importants.

Par exemple la “blendability”, le fait de pouvoir voir les données sous Blend ou VS en mode conception est vital.

La représentabilité des données est tout aussi importante. Un nom et un prénom ce n’est pas du Lorem Ipsum. En revanche un commentaire doit être du Lorem Ipsum ! L’attention ne

doit pas être perturbée par un contenu lisible, le cerveau à trop d'attirance pour ce qu'il sait reconnaitre... Et c'est une distraction inutile.

De même, les données de design se doivent d'être réalistes mais peu nombreuses : en conception pas besoin de 5000 entités "personne" pour mettre en place l'affichage d'une seule personne...

Enfin, les données de design doivent savoir s'effacer au runtime, le code qui les génère, les données externes utilisées, etc, tout cela doit être absent du code exécutable final.

Lorem Ipsum

Ce sont les premiers mots d'un texte en pseudo latin utilisé par les imprimeurs pour tester les mises en page. Son intérêt est d'être proche du français (même alphabet, longueur des mots, fréquences de ceux-ci, longueur des phrases etc) mais de ne pas être du français (ni même du vrai latin) afin que celui qui s'en sert ne soit pas perturbé, distrait, par un texte ayant un sens.

Vous trouverez à l'adresse suivante un générateur de mots, de phrases et de paragraphes Lorem Ipsum que j'ai écrit sous Silverlight et qui est accessible gratuitement, utilisez-le lorsque vous avez besoin de générer du Lorem Ipsum !

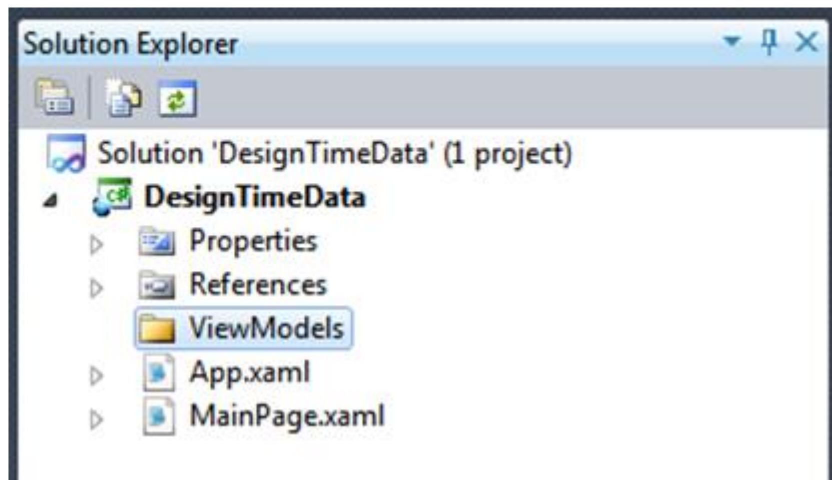
E-Naxos Lorem Ipsum Generator

VIEW, VIEWMODEL ET MODEL

Nous allons partir d'une simple application Silverlight de base, sans aucune fioriture. Je vous fais grâce des étapes de cette création d'un nouveau projet de ce type.

Le résultat est un projet vierge qui contient une page principale appelée MainPage.xaml. C'est le comportement par défaut de Visual Studio dans ce cas.

La première chose que nous allons faire est d'ajouter un sous-répertoire "ViewModels" pour conserver une structure propre et réutilisable même si, ici, nous n'aurons qu'un seul ViewModel.



Rien d'extraordinaire alors continuons...

Le Modèle

Pour parfaire l'exemple et le rendre le plus réaliste possible je vais ajouter selon le même principe un sous répertoire "Models".

Il contiendra la classe Personne qu'on supposera faire partie d'un ensemble constituant le BOL et le DAL de l'application.

Dans ce répertoire je vais ajouter la classe "Personne" :

```
using System.Collections.ObjectModel;
using System.ComponentModel;

namespace DesignTimeData.Models
{
    public class Personne : INotifyPropertyChanged
    {
        #region fields
        private int id;
        private string nom;
        private readonly ObservableCollection<string> applications = new
ObservableCollection<string>();
        #endregion

        #region properties
        public int ID
        {
            get
            {
                return id;
```

```
    }
    set
    {
        if (id==value) return;
        id = value;
        doChanged("ID");
    }
}

public string Nom
{
    get
    {
        return nom;
    }
    set
    {
        if (nom==value) return;
        nom = value;
        doChanged("Nom");
    }
}

public ObservableCollection<string> Applications
{
    get
    {
        return applications;
    }
}
#endregion

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

public event PropertyChangedEventHandler PropertyChanged;
#endregion
```

```
    }
}
```

Cette classe est constituée de trois champs : ID, Nom et Applications.

Le premier sera l'identifiant de la fiche Personne, le Nom... vous avez deviné, et Applications sera une liste des applications que la personne a installées sur son ordinateur. Tout cela est purement fictif pour l'exemple.

La création de Modèles ou de classes métier n'est pas le sujet du jour mais vous remarquerez que bien que modeste notre classe "Personne" n'en respecte pas moins les minimum vitaux : séparations propres avec des régions, support de INotifyPropertyChanged, champ liste géré par une ObservableCollection initialisée par l'instance et en readonly, la propriété Applications ne contient qu'un getter et surtout pas de setter, une méthode spécifique est créée pour gérer les notifications de changement (doChanged).

Le ViewModel

Dans le répertoire ViewModels je vais rajouter une classe qui s'appellera MainPageViewModel. Le suffixe "ViewModel" est une habitude que je conseille, cela permet rapidement d'identifier ces classes spéciales, quant au nom lui-même, de préférence on utilise le nom de la Vue (ici MainPage).

Comme j'ai déporté tout ce qui est "données" dans la classe Personne, le ViewModel qui expose une personne sera très simple :

```
using System.ComponentModel;
using DesignTimeData.Models;

namespace DesignTimeData.ViewModels
{
    public partial class MainPageViewModel : INotifyPropertyChanged
    {
        #region fields

        private Personne personne;
        #endregion

        #region properties
```

```
        public Personne Personne
        {
            get
            {
                return personne;
            }
            set
            {
                if (personne==value) return;
                personne = value;
                doChanged("Personne");
            }
        }
        #endregion


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

        public event PropertyChangedEventHandler PropertyChanged;
        #endregion

    }
}
```

C’est un “"vrai” ViewModel, supportant notamment INotifyPropertyChanged.

On notera juste une petite chose mais qui va faire la différence : la classe est marquée “**partial**”.
Pourquoi ?

Parce que justement, nous arrivons au cœur du sujet, nous allons exploiter cette possibilité pour “externaliser” le code de création des données de design dans un autre fichier. Il ne faut surtout pas que le code de conception vienne se mélanger à la classe réelle qui sera utilisée en exploitation...

CREATION D'UN SWITCH DE COMPILATION OU PAS ?

Arrivé à cette étape je suis obligé de vendre un peu la mèche... le code de génération de données pour le design est un code qui ne devra pas être compilé dans la version finale du logiciel. Pour se faire je vais utiliser une compilation conditionnelle (nous verrons qu'ici il y a une feinte à connaître...).

La première idée consiste à ajouter dans les paramètres de Build du projet, et pour le mode Debug, un nouveau switch qu'on pourrait appeler "Design" par exemple. Absent du build "release" par défaut il remplira parfaitement son rôle.

Toutefois cela revient à créer un synonyme de "Debug" déjà définit par VS ... Aurons-nous besoin de données de design autrement qu'en mode Debug ?

La question reste ouverte... Et chacun fera en fonction de son contexte. Pour ma part je ne vais pas ajouter de switch et j'utiliserai "Debug" comme marqueur de phase de conception.

Du coup, rien à ajouter ni à modifier nulle part. Mais comme l'apprend tout bon prof de philo à ses élèves éberlués par cette évidence à laquelle ils auront une vie pour réfléchir : Décider de ne rien changer est un changement en soi...

L'AJOUT DU CODE DE GENERATION DES DONNEES DE DESIGN

C'est ici que les choses intéressantes commencent vraiment. En tout cas celles concernant le sujet de ce billet... Où créer les données de design ?

J'ai vu certains développeurs bricoler cela au niveau des Modèles. Je ne suis vraiment pas pour cette approche. Les modèles doivent être "purs", ils sont le socle de l'application (sans données... pas d'application) et moins on tripote un code déjà testé mieux on se porte. D'autant plus que si les modèles peuvent être du POCO comme dans mon exemple, le plus souvent il va s'agir d'une couche de services type Web ou RIA Services. Et là, il est plus délicat (voire impossible) d'aller faire du bricolage dans le code de ces derniers où de leurs proxy auto-générés.

Le ViewModel est là pour faire l'adaptation des données pour sa vue (et mémoriser son état). C'est son job.

C'est donc au niveau du ViewModel que les données de design doivent être créées.

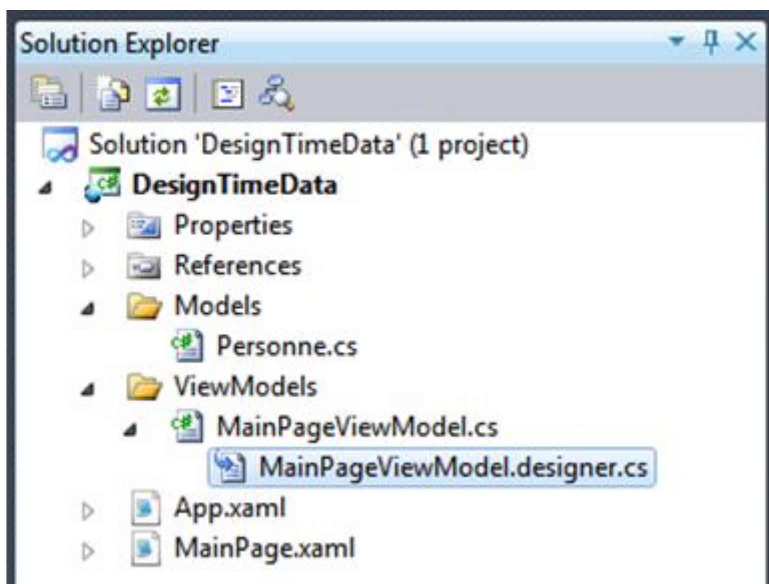
D'où la raison qui m'a fait ajouter "partial" au code du ViewModel et non à celui de la classe "Personne".

Ce code “partiel” sera contenu dans un fichier séparé et certainement pas mélangé à celui, définitif, du ViewModel. Tout l’intérêt de “partial” est là d’ailleurs et VS s’en sert abondamment dans ce sens pour simplifier le développement ou rendre extensible du code auto-généré par exemple.

Une astuce : en choisissant le nom de ce code partiel avec doigté VS le considèrera comme un comme spécial de design et le fichier apparaitra alors sous le nom du ViewModel comme tous les fichiers de design (que cela soit sous Windows Forms ou ASP.NET notamment, donc cette “norme” est déjà bien ancrée !).

Puisque le ViewModel s’appelle MainPageViewModel.cs, le fichier que nous allons créer s’appellera MainPageViewModel.designer.cs.

Il apparaitra sous le nom du ViewModel en y étant comme lié :



Malin, isn’t it ?

La génération des données

Que va contenir ce fichier de code ?

Tout simplement une partie “cachée” de MainPageViewModel puisque cette classe est partielle justement pour cela...

Nous allons ainsi compléter la classe avec une méthode de génération de données qui va créer ici une instance de Personne avec des informations fictives.

```
using System.ComponentModel; // designer properties
using System.Diagnostics; // conditional
```

```
using DesignTimeData.Models;

namespace DesignTimeData.ViewModels
{
    public partial class MainPageViewModel
    {
        [Conditional("DEBUG")]
        private void createDesignData()
        {
            if (!DesignerProperties.IsInDesignTool) return;

            personne = new Personne
                        {
                            ID = 1526,
                            Nom = "Olivier Dahan",
                            Applications =
                                {
                                    "Visual Studio",
                                    "Expression Suite",
                                    "MS Office",
                                    "PaintShop Pro",
                                    "Ableton Live"
                                }
                        };

        }
    }
}
```

Comme vous le constatez, une seule méthode a été ajoutée à la classe MainPageViewModel, il s’agit de “createDesignData”.

Deux choses à noter:

- La méthode teste le mode conception et s’en retourne aussi vite si aucun designer n’est accroché à l’application. C’est une simple sécurité de bon sens (et cela permet d’exécuter le code en mode Debug sans pour autant voir les données de conception…).
- La méthode est décorée par l’attribut “**Conditional**” et la condition est liée à la présence du switch de compilation DEBUG.

L'attribut Conditional est particulier et fort sympathique en ce sens que le code ainsi décoré ne sera compilé que si le switch passé en paramètre est présent. Pour fabriquer un mode "démo" d'une application cela est très pratique et très sûr : il suffit de placer le code réel des fonctions non activées dans la démo sous un Conditional testant le mode release et aucun pirate, même le plus malin ne pourra transformer votre démo en application utilisable... et pour cause, le code non autorisé ne sera tout simplement pas dans l'exécutable !

Vous allez me dire, c'est bien gentil de faire apparaitre ou disparaitre du code comme ça façon magicien... mais les appels à ce code ? Ils sont où et ils deviennent quoi quand le code n'est pas compilé ?

C'est là que Conditional est vraiment pratique ! Les appels aux méthodes ainsi marquées peuvent rester à leur place, si le code de ces méthodes n'est pas compilé, les appels à ces méthodes seront elles aussi ignorées et de façon automatique !

Nous allons d'ailleurs le voir tout de suite.

Notre code de design créée une Personne pour aider la conception visuelle de l'application. Mais il reste à appeler ce code quelque part...

Pour ce faire nous allons ajouter l'appel dans le constructeur du ViewModel. Ici nous n'avons pas de constructeur (ce qui est rare pour un ViewModel qui au minimum en général créée les commandes ICommand à cet endroit) nous allons en ajouter un.

C'est bien entendu dans le code MainPageViewModel.cs et non dans le code MainPageViewModel.designer.cs que le constructeur sera ajouté (tout simplement parce le ViewModel peut réellement avoir besoin d'un constructeur et qu'il serait stupide de le cacher dans le code design).

```
#region constructor
public MainPageViewModel()
{
   createDesignData();
}
#endregion
```

Ce n'est pas bien compliqué...

La méthode createDesignData() est appelée par le constructeur du ViewModel. Si nous sommes en mode conception (ici détecté par le switch DEBUG mais nous avons vu que nous aurions pu créer un switch à part) la méthode sera appelée. Si un concepteur est accroché au projet la méthode créera une instance de Personne disponible durant la conception. Cela est pratique lors de l'exécution du code même en debug, les données de design ne s'affichent pas.

Si nous ne sommes pas en mode de conception (absence du switch DEBUG dans notre cas, par exemple en mode Release), à la fois le code conditionnel de createDesignData() ne sera pas compilé dans l'exécutable et l'appel à cette méthode sera ignoré... Il n'y a donc rien à modifier dans le ViewModel. Il pourrait y avoir cent appels à la méthode cachés un peu partout dans notre code, ils seraient ignorés de la même façon...

Créer des données de design est une façon d'utiliser Conditional. Faire des démos sans le code fonctionnel et donc in-piratable est une autre idée que je développais plus haut, mais on peut trouver bien d'autres utilisations à cette technique (méthodes de test, version "light" / version "pro", etc...).

LE DESIGN

Après tout, nous avons fait tout cela pour simplifier le design...

La MainPage par défaut que VS a créé pour nous sera parfaite, nous n'avons pas besoin d'enjoliver dans cet exemple. En revanche il faut bien relier cette page Xaml à son ViewModel.

La plupart des frameworks offrent des moyens bien à eux pour ce faire. Comme ici nous n'en utilisons aucun il faudra faire cette liaison à "la main" :

```
<UserControl x:Class="DesignTimeData.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"
   mc:Ignorable="d"
   xmlns:ViewModels="clr-namespace:DesignTimeData.ViewModels"
        d:DesignHeight="300" d:DesignWidth="400">

   <Grid x:Name="LayoutRoot" Background="White">
      <Grid.DataContext>
         <ViewModels:MainPageViewModel/>
      </Grid.DataContext>
```

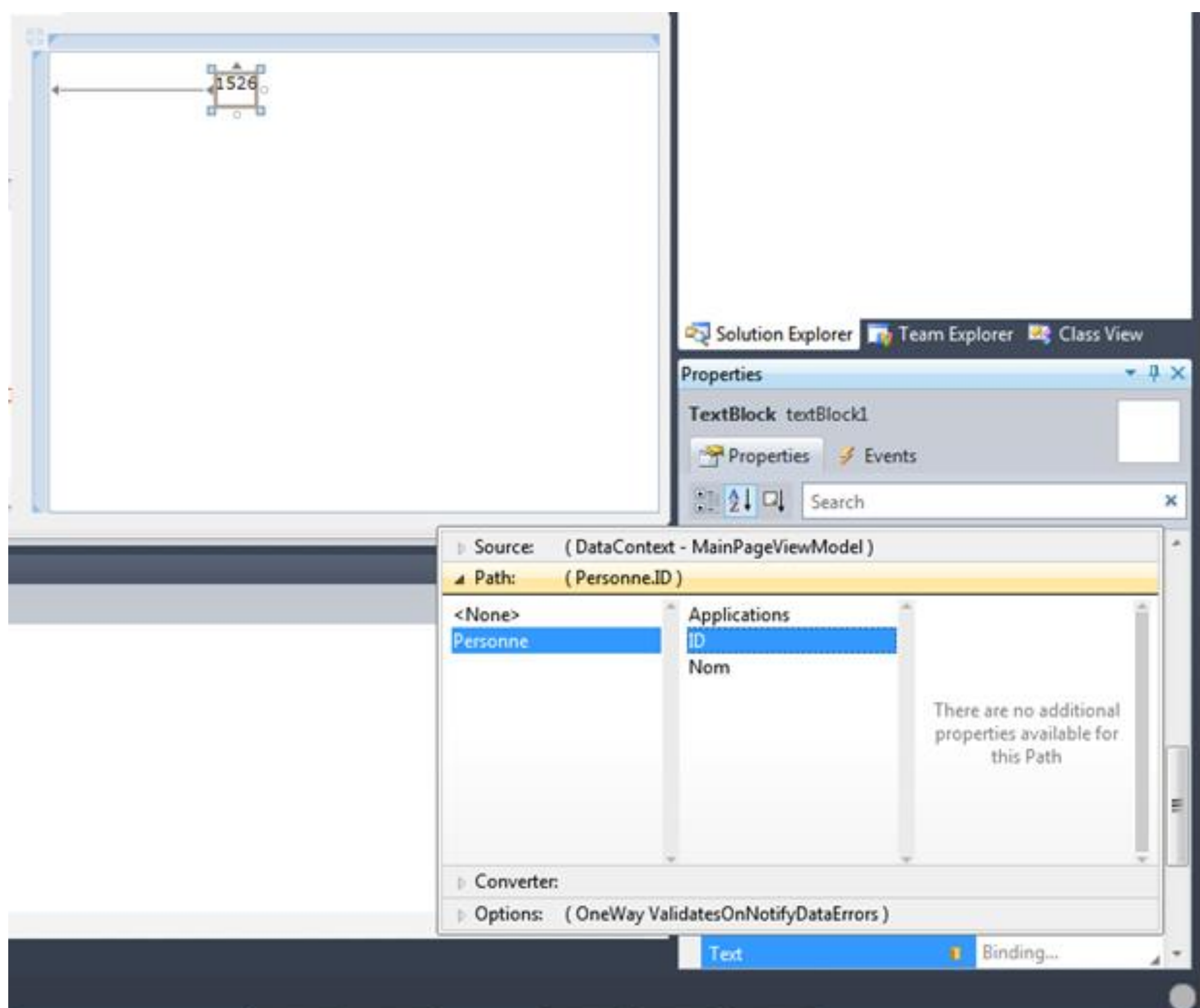
```
    </Grid>
</UserControl>
```

La première chose à faire est d'ajouter une référence à notre espace de noms des ViewModels, c'est le rôle de la ligne soulignée par mes soins dans l'entête de l'objet UserControl.

Ensuite il faut affecter le DataContext de la grille principale (LayoutRoot) pour qu'il pointe vers le ViewModel. C'est le rôle de la balise <Grid.DataContext> et du code qu'elle enchâsse.

Voilà... Le code de design est créé, il fabrique automatiquement des données pour mettre en page facilement l'application, ce code sera automatiquement supprimé de la version Release de notre application.

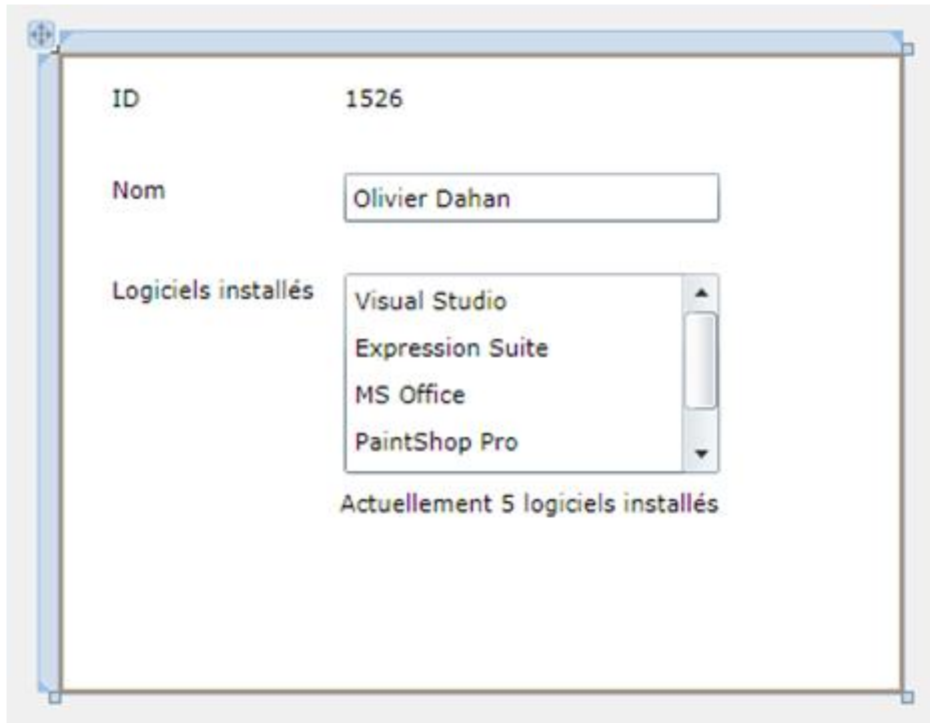
Ajoutons un TextBlock pour afficher l'ID de la Personne et ouvrons le gestionnaire de Binding sous Visual Studio, cela donne :



On voir la valeur "1526" s'affichée immédiatement après avoir sélectionné la propriété "Personne" du DataContext et la propriété ID de Personne...

Cela fonctionne de la même façon sous Expression Blend, à la différence de présentation des dialogues de Binding.

La mise en page peut s'effectuer, le designer ou l'intégrateur verra immédiatement si les zones sont bien placées, si elles sont assez grandes, etc...



(mode design sous Visual Studio)

Si nous exécutons l'application, les données ne seront pas affichées et cela bien que nous soyons toujours en mode DEBUG. C'est la raison d'être du petit test qui détecte le mode conception dans la méthode de génération de données de design...

Mais si nous passons le code sous Reflector par exemple, nous verrons bien que le code de design est toujours là.

En revanche en basculant en mode "Release" ce code aura totalement disparu ainsi que l'appel qui est fait dans le ViewModel sans avoir à toucher à quoi que ce soit...

Vous me croyez sur parole, pas besoin de rallonger ce billet par d'autres captures écran

CONCLUSION

Les données de design sont essentielles pour garantir une bonne mise en page de l'UI d'une application. Elles simplifient le travail du designer / intégrateur.

Comme nous venons de le voir ici, il suffit de très peu de choses pour créer de telles données de façon propre et totalement transparente au regard de l'exécutable définitif.

Il n'y a donc aucune raison de s'en passer... quel que soit le framework MVVM que vous utilisez et quelle que soit la plateforme choisie : Silverlight, WPF ou même WinRT.

MVVM, Chronomètre et Illustrator

MVVM, j'en ai parlé souvent et à travers de longs articles à télécharger, mais qu'est le rapport entre MVVM, un chronomètre et Illustrator ? Aucun. Si ce n'est qu'une fois associés, les trois permettent de voir comment construire une application MVVM tout en démontrant la fonction d'importation Photoshop/Illustrator de Expression Blend et quelques autres avantages de ce logiciel incontournable. Let's Go !

VOIR LE PROJET FINALISE

Il est toujours important de voir le projet fini avant de le décortiquer, on comprend mieux où on va...

Cliquez sur le lien suivant et jouez avec le chronomètre et ses deux boutons (en haut : start / stop, à gauche : split /reset, comme sur un "vrai") :

http://www.e-naxos.com/SLSamples/Chronos/TestPage.html

MVVM LIGHT

J'ai présenté cette bibliothèque de code au sein d'un gros article (Cf. "appliquer la pattern MVVM avec MVVM Light") dont je vous recommande la lecture si ce n'est déjà fait. Je ne donnerai pas ici de précisions sur ce toolkit pour éviter les redites. Mais le code démontré utilise MVVM Light. Il est donc temps de faire un crochet par l'article indiqué avant d'aller plus loin 😊

EXPRESSION BLEND

Expression Blend est un outil fantastique pour concevoir des applications Silverlight ou WPF. C'est l'un des rares EDI qui m'a autant enthousiasmé ces dernières années, en dehors de Visual Studio. Expression Blend n'est pas un gadget dont on peut se passer. Bien que VS ait intégré un designer pour Silverlight dans les dernières versions, il ne permet tout simplement pas de tout faire en XAML. De plus VS n'a pas été conçu globalement comme un outil pour le design artistique, c'est un IDE de codeur. VS reste donc a sa place et Blend ne

peut le concurrencer sur ce point (même si on peut éditer du code avec Blend). Pour tout ce qui est mise en place du visuel d'une application, seul Blend est l'outil vraiment adapté.

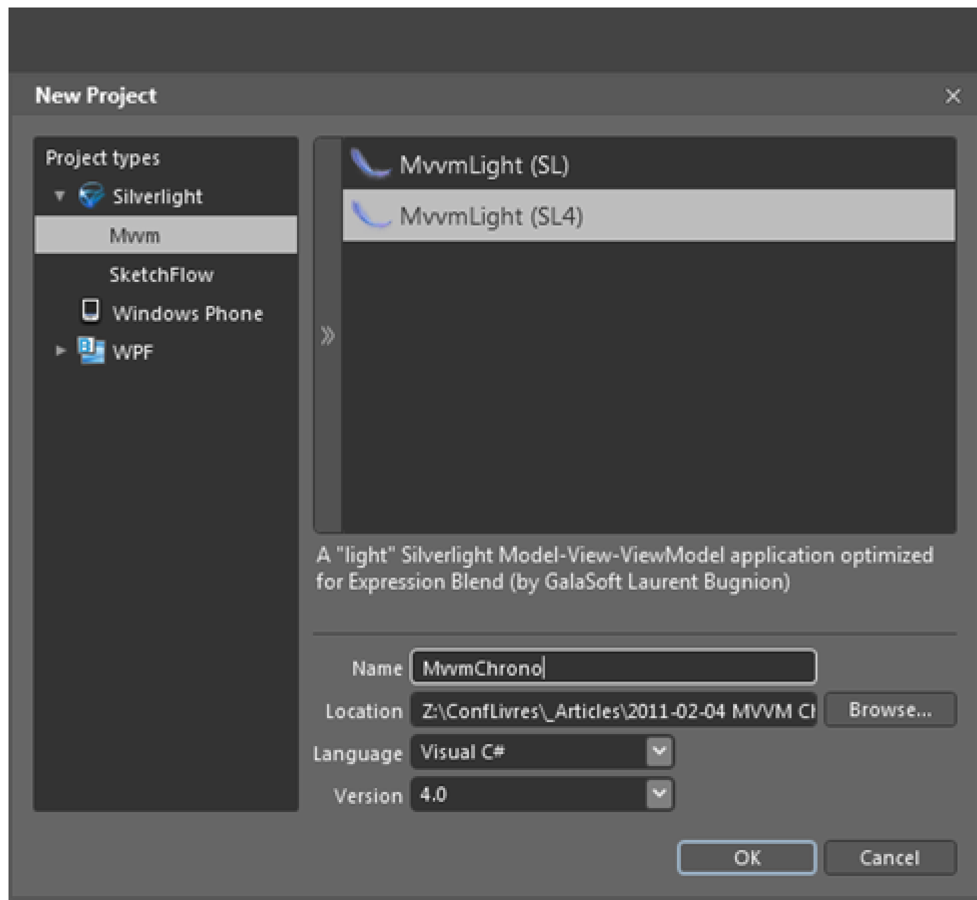
Tout naturellement c'est dans Expression Blend que se fera l'essentiel de ce billet. Notamment l'importation Adobe Illustrator qui n'existe pas dans VS.

L'OBJET GRAPHIQUE DU PROJET

On pourrait utiliser un graphique bitmap issu de Photoshop, mais un vectoriel de Illustrator sera généralement plus conforme à l'esprit XAML, vectoriel par nature. Avec un objet Illustrator on peut retravailler plus facilement le dessin, l'adapter, le modifier car il reste en vectoriel, là où un bitmap ne peut être corrigé sous Blend. Mais en fait cela ne change pas grand-chose du point de vue de la manipulation que je vais vous faire voir. En vous baladant sur le Web vous trouverez de nombreux sites offrant des dessins gratuits dans l'un ou l'autre de ces formats. Ici je vais utiliser un chronomètre que j'ai récupéré sur le site oneter (un fichier EPS que j'ai mis au format Illustrator “ai” pour l'importer sans problème). Pour les bitmaps il y a aussi le site “psdGraphics” qui propose des JPG ou des fichiers PSD de Photoshop. Il y a des tonnes de sites de ce genre, je vous laisse surfer...

CREATION DU PROJET

Dans Expression Blend, donc, Fichier / Nouveau Projet. Là il faut avoir installé le toolkit MVVM Light car il faut choisir un projet de type “MvvmLight (SL4)” comme le montre l'écran ci-dessous:



Une fois le projet créé nous obtenons une base simple qui a le mérite d'avoir une page d'accueil et son ViewModel déjà créé et recensé dans le ViewModelLocator. Pour un projet d'une seule page il n'y a rien à ajouter, et cela tombe bien pour cet exemple...

Je vais ouvrir directement la page "MainPage.xaml" en édition, sachant que le template de projet a déjà lié son DataContext au ViewModel correspondant (ViewModel/MainViewModel.cs).

J'ai fais le nettoyage de la page (MVVM Light ajoute un TextBlock lié à une propriété du ViewModel pour démontrer le mécanisme, on peut tout enlever et ne conserver que le LayoutRoot puis ajouter les éléments dont on a réellement besoin).

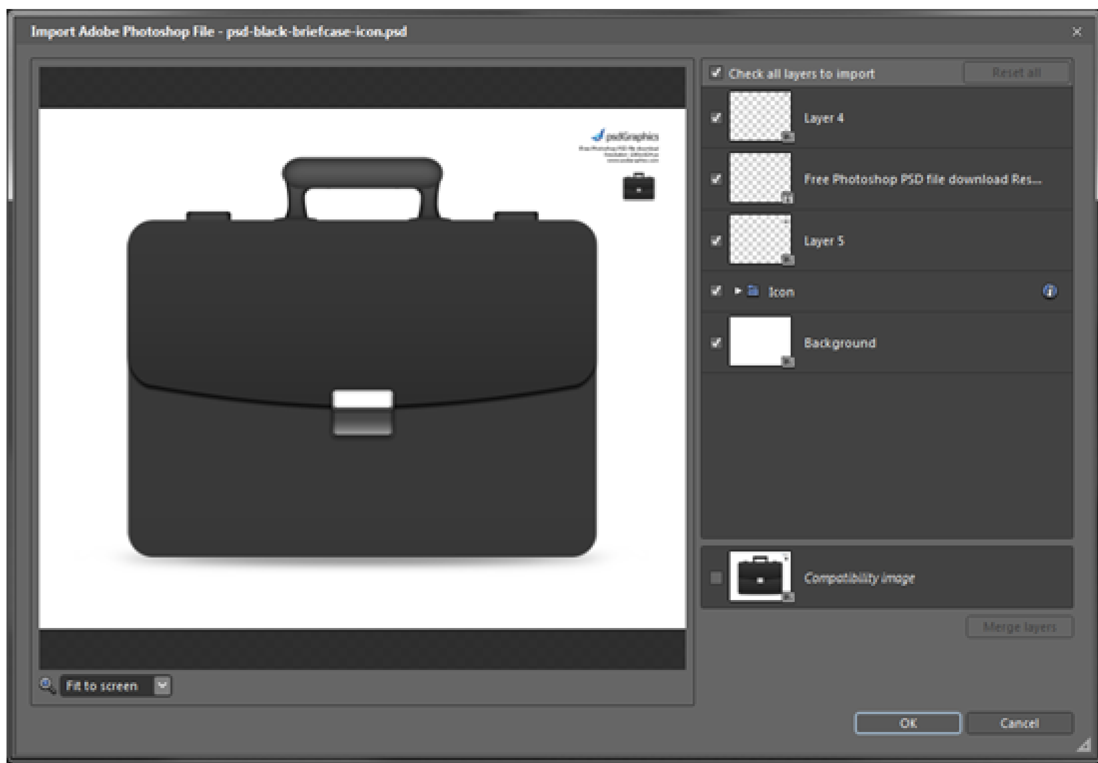
L'IMPORTATION DU GRAPHIQUE

Comme je l'ai dit, il s'agit d'un graphique vectoriel récupéré sur le Web en format EPS puis transformé en "ai" Adobe Illustrator. Cela m'a permis de simplifier le dessin original et de lui enlever tout ce qui n'était pas vraiment nécessaire (comme une seconde copie du chronomètre avec d'autres couleurs). Bref, ce qui est important, c'est de disposer au départ d'un dessin, bitmap ou vecteur, provenant de Photoshop ou Illustrator.

Maintenant dans le menu Fichier, cliquons sur “Importer fichier Adobe Illustrator”. La boite de dialogue d’ouverture de fichiers est affichée, ne reste plus qu’à sélectionner celui qu’on désire importer et de cliquer sur Ok.

C’est tout...

Si le fichier ne contient qu’un seul layer, il n’y a pas de question à se poser. Dans le cas où le fichier comporte plusieurs layers, Blend affiche un dialogue intermédiaire permettant de choisir ceux qu’on souhaite importer. Par exemple, avec un fichier PSD (Photoshop) le dialogue ressemble à cela :



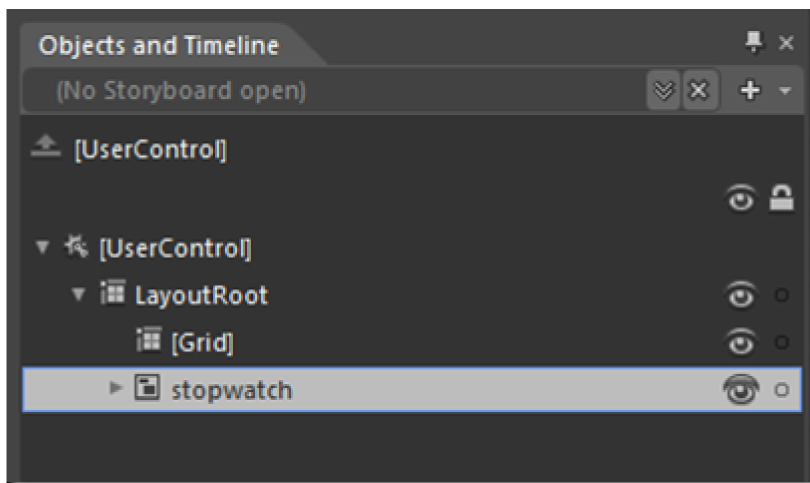
Sur le côté droit on voit la liste des layers, tous sélectionnés par défaut. Imaginons qu’on souhaite ici importer la petite sacoche, il est évident qu’on préfèrera supprimer les layers affichant le texte et la miniature en haut à droite. Cela est purement conjoncturel et dépend uniquement de la façon dont le graphiste a travaillé... D’où l’importance pour des projets clients de s’être bien mis d’accord avec lui sur la façon d’organiser ses dessins et sur l’éventuelle signification des différentes couches (layers).

Un seul fichier Photoshop ou Illustrator peut ainsi contenir plusieurs illustrations différentes ou des variantes d’une même illustration, l’essentiel est que les choses soient séparées sur des layers différents pour rendre l’importation rapide et facile...

Mais revenons à notre chronomètre.

La fichier source n'ayant qu'un seul layer, nous n'avons pas eu de dialogue intermédiaire et Blend a placé directement l'image vectorielle dans un groupe à l'intérieur de LayoutRoot :

L'arbre visuel est le suivant :



J'avais commencé à placer une grille, le groupe "stopwatch" (nom issu du fichier original) a été ajouté par Blend en dessous. Il va falloir replacer le tout dans la mise en page que j'avais prévue et éventuellement modifier quelques éléments importés.

Tout cela n'étant pas très important, je vous fais grâce de ces étapes.

FAIRE MARCHER LE CHRONOMETRE

Etat des lieux

Jusque là, nous avons importé une image vectorielle directement depuis un fichier Illustrator et nous l'avons placé correctement dans notre page. C'est peu de chose, cela a été très rapide, mais tous ceux qui doivent travailler avec un infographiste pour créer leurs

applications Silverlight et WPF auront compris à quel point cela est important... Blend a transformé le fichier “ai” en un groupe correctement nommé contenant des Path contenus dans un Canvas. C’est magique, le chronomètre est désormais un objet XAML constitué de chemins modifiables (forme, couleur, etc.).

Ajustements

Pour faire marcher le chronomètre nous allons avoir besoin d’un peu de code, mais avant tout il faut s’assurer que les “morceaux” du dessin que nous voulons manipuler sont bien accessibles, bien positionnés dans le Z-order, etc. Notamment il va falloir mettre l’aiguille “à zéro”. Comme elle constituée de plusieurs Paths qui ne sont ni nommés ni groupés (le graphiste n’aurait pas fait son travail correctement s’il s’agissait d’un travail effectué pour un projet Silverlight/WPF) nous allons devoir modifier tout cela. D’où, une fois encore l’intérêt de Expression Blend qui nous offre des outils vectoriels comme Expression Design ou Adobe Illustrator. Bien entendu Visual Studio ne propose pas de tels outils trop spécifiques au monde de l’infographie.

Un petit détail concernant l’aiguille : pour la mettre “à zéro” il suffit de lui appliquer une rotation. Certes. Mais il ne faut surtout pas oublier de modifier le centre de rotation de l’objet avant. Ici il faut faire coïncider “à l’œil” ce centre de rotation et le rond de l’aiguille qui simule son axe.

Comme il y a fort peu de chance pour que le dessin d’origine soit uniquement conçu avec des chiffres ronds en pixels (les outils de dessin vectoriels ne travaillent pas en pixels, et les graphistes ne font pas la petite fixette que nous faisons sur la beauté des puissances de deux !) il sera nécessaire de placer ce point “à l’œil”. Il faut donc l’avoir bon, mais surtout savoir se servir du zoom et ne pas hésiter à zoomer à 3000% si cela est nécessaire.

Idem pour la rotation qui va amener l’aiguille sur midi. Le faire à la main n’est pas très précis mais vous pouvez approcher la perfection en travaillant en deux temps : d’abord utiliser la palette de transformation de Blend et changer la rotation en bougeant la souris afin d’approcher au plus près le centrage désiré, puis, à la main, modifier l’angle indiqué en y ajoutant des décimales jusqu’à ce que tout soit ok (en ayant bien zoomé !). Par exemple dans ce projet, à -120° l’aiguille était trop à droite, et à -121° elle était trop à gauche... Le zoom étant à 1600% pour bien visualiser la zone. A la main et selon une approche de type recherche dichotomique (n’en parlez pas à l’infographiste 😊) je suis arrivé rapidement sur la bonne position : –120,85%.

Le principe de mise en page

J’ai ici opté pour une mise en page des plus simples : le LayoutRoot de la page principal (et unique) est une grille en mode 100% automatique sur les deux axes. Elle remplira donc tout l’espace dédié au plugin Silverlight dans la page Web. Le chronomètre a été importé

directement dans un Canvas, je l'ai laissé ainsi. En revanche j'ai correctement retaillé ce dernier et je l'ai paramétré pour qu'il soit centré sur les deux axes.

C'est très simple, et cela permettra que le chronomètre apparaisse centré quelle que soit la taille de la fenêtre du browser.

Le temps précis

Il manque l'affichage du temps précis écoulé. Notamment quand le chronomètre aura fait plus d'un tour il sera difficile de se rappeler exactement combien de fois l'aiguille aura tourné et encore moins de faire le calcul de tête (l'utilisateur n'est pas un informaticien !).

Pour ce faire j'ai ajouté un TextBlock sous l'aiguille. Il est de taille fixe, et c'est le texte qui est en mode centré. L'objet est placé convenablement dans l'arbre visuel pour qu'il soit sous l'aiguille, bien évidemment.

J'ai donné un nom à l'objet texte, c'est surtout pour le repérer facilement, nous allons voir plus loin qu'en MVVM aucune référence à ce texte n'est nécessaire et qu'il peut rester sans nom.

Des boutons qui marchent

Le chronomètre possède deux boutons, je vais essayer de faire simple pour cette démonstration... Le bouton du haut est le Start/Stop. Facile. Le bouton de gauche est le "Split/Reset". Quand le chronomètre tourne (Start) cela permet de figer l'affichage sans arrêter le mécanisme (temps intermédiaire). Un second appui relâche l'aiguille qui se remet à compter normalement. Quand le chronomètre est à l'arrêt, le bouton Split/Reset ramène l'aiguille à zéro et efface le temps écoulé. Il s'agit d'un automate classique et très simple pour le lequel je ne vous ferai pas un diagramme UML...

En revanche, du point de vue graphique, le dessinateur n'avait certainement pas prévu qu'un jour quelqu'un se serve de son fichier pour faire une application Silverlight. De fait les boutons ne sont pas formés de "pièces" qui peuvent être facilement déplacées. De plus les deux boutons sont assez différents.

La solution va consister ici à simplifier le dessin et à transformer l'un des boutons en contrôle templaté puis à le dupliquer, chaque copie recevant une orientation convenable.

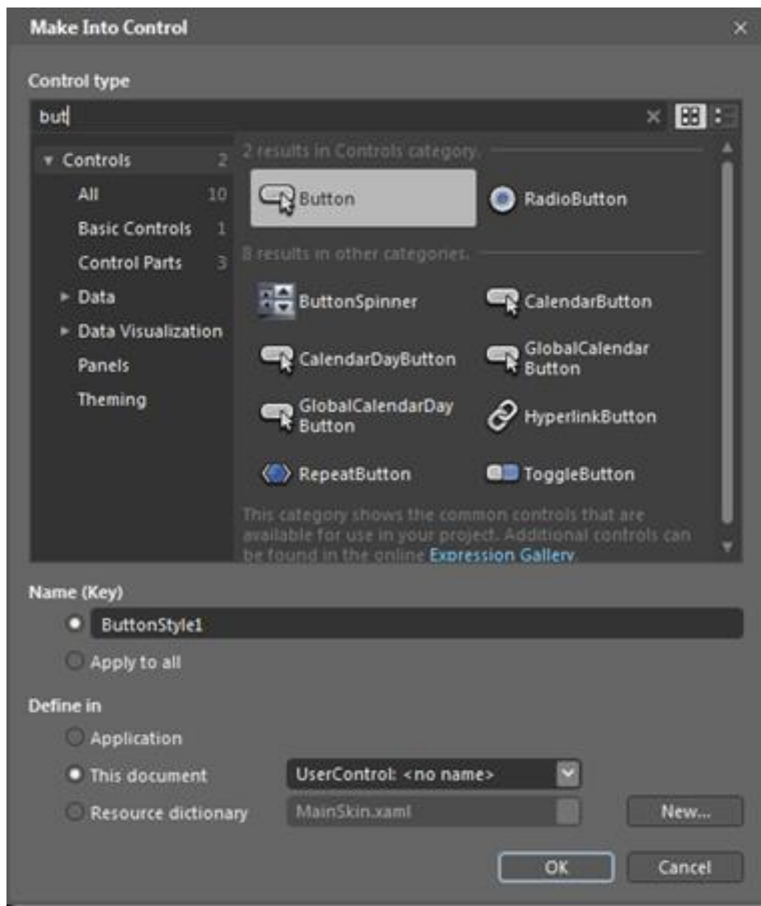
L'image ci-dessus montre la différence entre les deux boutons originaux. Il va falloir choisir lequel garder. Chacun règlera ce dilemme selon ses gouts...

Les boutons sont ici les éléments qui comptent le plus de Path ou presque. Toutes les petites rayures sont autant de chemins dessinés. Les sélectionner un à un pour les grouper ou les supprimer (ce que nous devons faire pour l'un et l'autre respectivement) peut être très compliqué malgré le zoom et les facilités de sélection de Blend. Je vous conseille plutôt l'astuce suivante : dans l'arbre visuel chaque objet est doté d'un œil, ce qui permet de le cacher ou de le montrer (uniquement en conception, aucun effet sur le rendu final), cachez tous les objets, un par un, jusqu'à ce que seuls restent les objets qui vous intéressent. On contrôle mieux visuellement ce qu'on fait, c'est plus clair, et il est plus facile ensuite de sélectionner les éléments visibles pour les grouper ou les supprimer.

J'ai choisi de supprimer le bouton de gauche et de conserver celui du haut. Aucune raison esthétique dans ce choix, juste de la logique : celui du haut est déjà dessiné "verticalement". En faire une copie orientée autrement pour recréer le bouton de gauche sera facile. Partir du bouton de gauche obligerait d'abord à le "verticaliser" ce qui sera par force approximatif (même en zoomant et en le faisant bien). L'expérience parle, croyez moi, c'est plus facile en partant de celui qui est déjà bien vertical...

Il me reste donc uniquement le bouton supérieur. Le bouton de gauche a été totalement supprimé. Le bouton du haut Il a été groupé dans un Canvas. Le Canvas remplace la notion de "layer" sous Blend. N'hésitez jamais a regrouper des éléments de dessin dans des Canvas et à les nommer si le graphiste ne l'a pas fait, on s'y retrouve mieux.

Les pièces composant le bouton étant groupées, il me suffit maintenant de sélectionner le groupe (dans l'arbre visuel. C'est souvent plus simple de manipuler les objets dans celui-ci que directement dans l'espace de travail dès que la scène est complexe) puis de faire un clic droit et de choisir "**make into control**" (je ne sais pas comment cela a été traduit dans la version FR que je n'utilise pas mais vous devriez trouver !). Ce qui ouvre le dialogue suivant :



Je fais un “make into control” et non un “make into UserControl” car ce qui m’intéresse n’est pas d’inventer un contrôle mais bien de transformer mon bouton visuel en un bouton fonctionnel. Bien entendu la classe Button est la plus proche de mes besoins et c’est elle que je choisi.

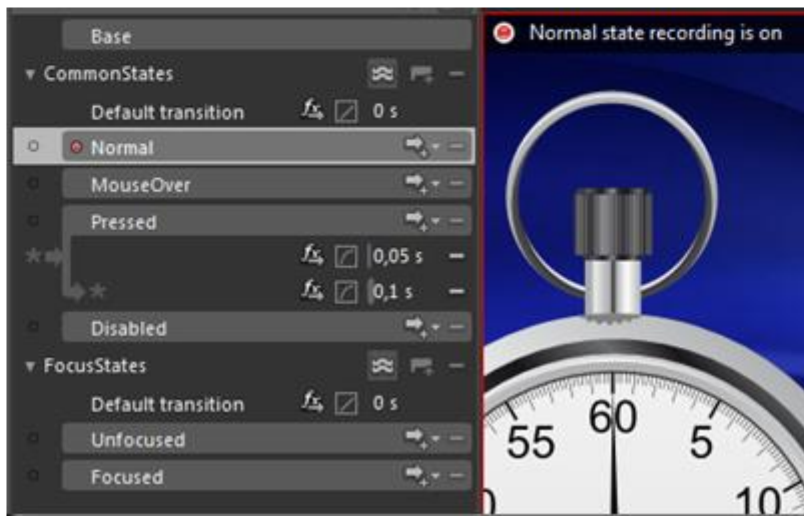
Automatiquement Blend transforme mon Canvas en un Button qu’il place exactement au même endroit. Rien à faire donc… Visual Studio est bien incapable de telles prouesses malgré ses qualités énormes pour tout ce qui touche le code.

En fait cela va si vite qu’on peut se demander si quelque chose s’est passé. Mais à mieux y regarder on comprend que : d’une part le Canvas (ou le ou les objets sélectionnés) a été supprimé de l’arbre visuel, qu’un bouton a bien été créé à la place (on voit le texte du ContentPresenter en surimpression) et que Blend a immédiatement basculé en mode templating (l’arbre visuel l’indique et de nombreuses petites choses ont changé pour qui connait bien Blend).

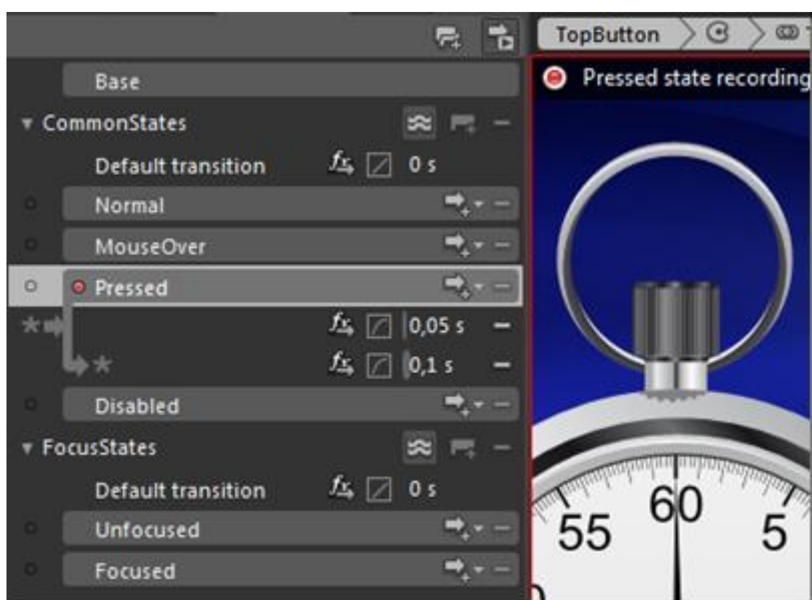
En réalité aucun nouveau “composant” n’est créé, nous sommes juste en train de donner un nouveau look à la classe Button. Ce nouveau look, ce nouvel aspect c’est le dessin formé par tous les éléments que nous avons sélectionnés avant de faire “make into control”. Et tout cela n’est rien d’autre qu’un simple template pour la classe Button. Une ressource XAML qui est appliquée à une instance de Button, celle qui a remplacé le dessin original…

Travailler avec Blend est magique et agréable. Ceux qui prétendent pouvoir s'en passer ont certainement loupé quelque chose dans la notion de User Experience et dans l'obligation de créer des interfaces utilisateurs un peu différentes de celles qu'on faisait en Windows Forms. Peut-être arriverai-je, avec le temps, à en convaincre quelques uns !

A partir de maintenant nous sommes donc en mode templating sur un bouton. L'idée générale est de séparer proprement la partie supérieure de l'axe du bouton visuel afin de pouvoir faire bouger le capuchon lorsqu'on "appuiera" dessus (lorsqu'on cliquera dessus). Une fois le travail de séparation effectué c'est en jouant avec les différents états du Visual State Manager que nous donnerons l'illusion de ce "clic" un peu spécial.



Le VSM sur l'état "Normal" et le bouton en position haute par défaut.



L'état "Pressed" est sélectionné, j'ai déplacé le capuchon vers le bas. On notera les petits réglages du VSM. Tous les états vers "Pressed" provoquent un changement assez rapide, il ne faut pas donner l'impression que le logiciel est "mou". J'utilise souvent un temps à zéro

pour les boutons. Ici j'ai mis quelques centièmes de secondes pour simuler la course d'un bouton mécanique. Dans le sens "Pressed" vers tous les autres états la durée est légèrement supérieure, c'est visuellement plus agréable (mais toujours faire attention à la sensation de molesse !). Le tout est agrémenté d'un easout cubique.

Ajouter du son

Pour rendre le tout plus sympathique et moins plat il faudrait ajouter du son. Je dispose d'une énorme banque de données de sons divers (que j'utilise principalement pour la musique je compose) dans laquelle j'ai trouvé deux sons intéressants, l'un pour le clic du bouton et l'autre qui sera joué pour imiter le bruit du chronomètre qui tourne.

Le son est-ce important ? Pour un musicien, certainement 😊 Mais dans le cadre d'un tel développement qu'est-ce que cela peut apporter ?

Tout est une question de dosage et d'à propos. Un léger bruit de clic quand on appuie sur les boutons ajoutera du réalisme et plongera l'utilisateur plus facilement dans notre univers virtuel. Ce n'est pas intempestif, ça reste léger, ponctuel et on choisira un son assez doux (pas un bip électronique qui casse les oreilles...).

Le second son que je vais utiliser est la simulation du "tic tac". Est-ce utile ? Je pourrais reprendre le même argument : cela plonge l'utilisateur dans l'univers virtuel du logiciel autrement que par la vue, en associant l'ouïe qui n'est sollicitée que dans les jeux en fait (univers conçus pour être prenant justement). Alors attention, notre application n'est pas un méga jeu, c'est une application utile. Mais utiliser les mêmes astuces que les jeux qui savent captiver l'utilisateur n'est pas un pêché... Et puis ici notre chronomètre simule le fonctionnement d'un véritable objet. Par exemple lorsque l'utilisateur figera l'affichage (temps intermédiaire) avec le bouton de gauche, comment faire sentir que le chronomètre n'est pas "en panne", pas "arrêté" ? On pourrait ajouter un message (beurk !) ou une animation... Avez-vous déjà vu une animation sur un chronomètre mécanique ? (en fait oui, des petits mouvements de balancier en général). Mais notre modèle n'a pas été dessiné ainsi. Alors, que peut-il y avoir de plus naturel que le tic tac régulier pour signifier à l'utilisateur "mon affichage est figé mais, écoutez le son, oui, je suis toujours en marche !"...

Les sons, en activant un sens peu utilisé en informatique classique peut aider à transmettre des informations utiles sur l'état du logiciel qui ne viennent pas brouiller la vue trop souvent sollicitée. Il faut s'en servir correctement, passer parfois des heures à écouter des banques de sons avant de trouver le bon, celui qui ira dans le contexte, parfois même il faudra utiliser des logiciels particuliers pour le traiter même sommairement (ce qui a été fait ici d'ailleurs, à l'aide Sound Forge de Sony, un excellent logiciel de traitement du son).

Maintenant que nous savons pourquoi nous utiliserons du son (deux pour être précis), comment les faire jouer ?

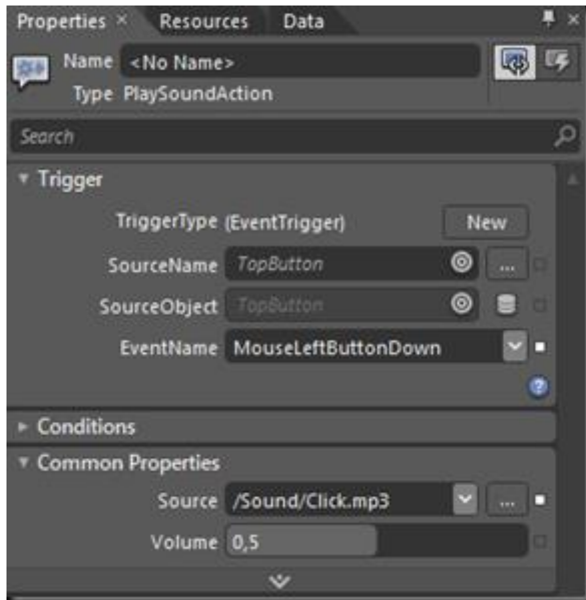
On conçoit facilement que pour le tic tac il suffira de jouer le son toutes les secondes ou demi secondes et qu'un simple timer, celui qui fera avancer l'aiguille par exemple, sera suffisant. En revanche pour le clic du bouton c'est moins évident.

En effet nous sommes au sein d'un template, pas un bout de code auquel se raccrocher. Que du XAML rangé dans un dictionnaire de ressources. Damned !

La feinte consiste repérer (ou à créer s'il le faut) un élément du dessin qui occupe l'espace cliquable. Le bouton étant déjà dans un Canvas rectangulaire qui couvre toute sa surface c'est lui que je choisirai. Sinon il aurait fallu ajouter un Canvas ou une Rectangle ou autre, peu importe, pour couvrir la région cliquable. Mais attention, le Canvas qui entoure l'objet n'a pas de Background. Vous pouvez toujours essayer de détecter le clic, ça ne marchera pas. Il lui faut une couleur transparente pour que le clic soit attrapé...

Seconde astuce, comment créer une simple couleur totalement transparente et est-ce nécessaire ? D'une part il n'est pas nécessaire de créer une couleur pour cela et d'autre part il existe une astuce : un clic sur le carré des options de la propriété Background affiche un menu (celui par lequel on effectue un data binding), il suffit de choisir l'entrée "Custom expression" (expression personnalisée). Une petite boite de saisie s'affiche, et là on tape "Transparent" puis Return (avec la majuscule). Et voici un Canvas avec une transparence totale, sans sélectionner de couleur mais qui, désormais, peut attraper le clic... C'est un pas en avant.

Mais comment gérer le clic dans un template qui ne possède aucun code-behind ? On ne le fait pas, tout simplement (ou alors on utilise des triggers, ce qui n'est pas très pratique). Encore une autre astuce : on place sur le Canvas qui nous sert "d'attrape clic" un Behavior : PlaySoundAction. L'évènement déclencheur sera le LeftMouseButtonDown, et la source du son sera un fichier mp3 préalablement ajouté au projet et qui apparaitra dans la combobox "source" du Behavior :



C'est tout bête mais il faut y penser…

On peut déjà exécuter l'application et, miracle, le bouton fonctionne bien, il s'abaisse et se relève comme prévu, le tout accompagné d'un joli son !

Il ne reste plus qu'à sortir du mode de templating, ce qui nous fait revenir à la page principale, puis de faire un copier / coller du bouton du haut pour recréer le second bouton qui était sur la gauche (un peu de placement, une petite rotation et l'affaire est jouée).

Deux choses à prendre en compte pour se faciliter le travail : d'une part après avoir fait le copier / coller le nouveau bouton se trouve en haut du Z-Order, il faut le ramener au même niveau que le premier bouton pour respecter les “couches” du dessin et que le bas de l'axe du bouton soit caché par le cadran. D'autre part, pour effectuer le placement il faut agir avec doigté ! Le faire à la sauvage en déplaçant l'objet et en espérant qu'on sera aligné sur le cadran qui est un objet rond, c'est possible, mais bien délicat.

Pour cela il faut procéder autrement. Il suffit de déplacer le point qui marque le centre des transformations pour le faire coïncider avec le centre du cadran, là où nous avons positionné le centre de l'aiguille il y a un moment. Une fois ce point placé correctement au centre du chronomètre, il ne reste plus qu'à appliquer une rotation à notre bouton : il tournera autour du cadran, on s'arrête quand l'angle nous plait, c'est aussi simple et imparable que ça !

Placé correctement le point est donc la clé de la manipulation. Pour cela je n'hésite pas à zoomer au maximum, à 6400% sur la zone centrale. Il ne faut pas mesquiner sur le zoom !

Autre détail : si vous regardez bien, vous verrez que le bouton du haut possède une ombre qui laisse supposer que la lumière arrive directement depuis le haut. Ce n'est d'ailleurs pas très homogène avec le reste de l'objet mais visuellement ça passe. Les graphistes comme les

peintres se laissent souvent aller à de telles libertés contraires aux lois de la physique mais qui “passent” visuellement. Pour un scientifique c’est un crime, pour un graphiste, c’est normal.

En tout cas, la fameuse ombre, comme le graphiste n’est plus là, il va bien falloir la simuler sous le second bouton placé sur le côté... Ici aussi, tout est question d’astuce.

Puisque l’ombre du bouton haut laisse supposer que la lumière vient directement du dessus, il faut que l’ombre du bouton de gauche soit légèrement déformée. Un graphiste ferait peut-être autrement, moi et mon esprit cartésien on pense qu’il serait bon de l’incliner un poil. La première étape (après avoir copier / coller l’originale) consiste à appliquer un Skew sur l’ombre pour l’étirer à gauche. Ensuite on place son point de référence des transformations sur le centre du chronomètre et on applique une rotation. Simple aussi donc. Reste à jouer sur sa transparence, l’ombre va se trouver sur une zone plus brillante, elle doit être atténuée.

On fait un Run, et la magie s’opère à nouveau : les deux boutons fonctionnent à merveille, avec leur petit son. Le vectoriel s’est beau, XAML c’est très beau.

Fin de la partie graphique

Graphiquement nous avons fait le tour de la question. Nous avons récupéré un objet Illustrator par importation directe dans Blend, nous l’avons modifié pour nos besoins en ajoutant un texte (qui contiendra le temps écoulé écrit en clair), nous avons transformé une partie du dessin en template de bouton que nous avons dupliqué pour reproduire l’objet original. Comble du raffinement les boutons s’enfoncent et se relâchent avec douceur (VSM plus easout) et jouent un son quand ils sont cliqués (behavior et un mp3). L’aiguille est bien à zéro (sur midi) et l’application compile. Que demander de plus ? !

Et bien... il faudrait que ça marche vraiment. Ca serait quand même mieux.

Il reste donc à faire en sorte que les boutons agissent vraiment sur l’objet, que l’aiguille tourne et que le TextBlock affichant le temps écoulé soit mis à jour.

J'ai choisi de développer ce petit projet en mode MVVM parce que tout doit être développé en MVVM. Nous allons voir maintenant comment donner vie à notre chronomètre en respectant la plomberie !

LE CODE

Nous suivons le pattern MVVM et utilisons le toolkit MVVM Light. Tout va se jouer ainsi dans le VIewModel de la page principale et unique de notre applicaton. Le fichier MainViewModel.cs.

Par défaut le template de projet MVVM Light contient déjà le ViewModel Locator, et la page principale (le DataContext de la Vue) est déjà connectée à son ViewModel via le Locator. Nous n'avons qu'à ajouter du code au ViewModel puis à faire quelques bindings dans la Vue et l'affaire sera réglée.

Deux boutons, deux commandes

Notre interface comporte deux boutons, nous retrouverons ainsi deux commandes dans le ViewModel. MainButtonCommand, le bouton du haut (start / stop) et LefButtonCommand, le bouton de gauche (split / reset).

Le code source complet du projet est fourni est fin d'article, je ne montrerai ici que des extraits pour illustrer le propos. Par exemple la commande MainButtonCommand sert uniquement à basculer le chronomètre en mode marche / arrêt. Cela est représenté par une propriété State (état) qui est de type enum et qui accepte les valeurs MainState.Stopped et MainState.Running.

La commande MainButtonCommand sera ainsi initialisée dans le constructeur du ViewModel :

```
MainButtonCommand = new RelayCommand(()=>
 {
   State = State == MainState.Stopped ? MainState.Running : MainState.Stopped;
 });
```

La commande étant déclarée comment suit :

```
/// <summary>
/// Gets or sets the main button command.
/// </summary>
/// <value>The main button command.</value>
public ICommand MainButtonCommand { get ; private set; }
```

C’est bien entendu la propriété State qui, en changeant de valeur va déclencher ou arrêter l’horloge.

Qui dit horloge dit Timer. Dans une application qui doit être très précise il faudrait utiliser des timers qui le sont aussi. Ici j’utilise un DispatcherTimer qui évite de se poser des questions sur d’éventuels problèmes de threads (car seul le thread de l’UI peut mettre à jour l’UI). Lorsqu’on utilise que des data binding une telle précaution n’est pas obligatoire, le Framework le gère. Mais je vais aussi utiliser des Messages pour la démonstration et je veux éviter d’avoir à vérifier les invocations. En revanche avec un DispatcherTimer nous aurons des temps approximativement corrects. Le chronomètre avancera par pas de 500 ms, et comme le prouvera le TextBlock affichant le temps écoulé, les millisecondes tomberont rarement juste ! (ce qui ne serait pas très beau dans une application réelle).

Une fois le chronomètre démarré, le Timer se met en route. Regardons le code de son évènement Tick :

```
void timer_Tick(object sender, EventArgs e)
{
   Messenger.Default.Send(new GenericMessage<string>("TICK"));
   if (substate == SubState.Splitting) return;
   var old = SecondAngle;
   var delta = DateTime.Now - startDate;
   var ms = (delta.Seconds*1000d) + delta.Milliseconds;
   SecondAngle = (ms / 60000d) * 360d;
   RaisePropertyChanged("SecondAngle", old, SecondAngle, true);
   RaisePropertyChanged("ElapsedTime");
}
```

La première chose qui est faire est l’envoi d’un message MVVM Light, un message générique transportant une chaine de caractère indiquant “TICK”. A chaque Tick il faudra bien jouer le son prévu.

Mais comment un ViewModel peut-il jouer du son alors qu’il n’a aucun lien avec l’UI ?

Le plus simple est d’envoyer un message. L’UI qui possèdera un MediaElement n’aura qu’à répondre au message “TICK” pour jouer le son. Sans savoir d’où vient le message, sans couplage fort entre les classes et sans astuces alambiquée. Un mécanisme simple et sain donc, qui respecte le pattern.

Notre chronomètre gère un “sous état”, qui est déclenché par le bouton de gauche. Lorsque le chronomètre est en marche, un appui sur ce dernier suspend l’affichage ou le reprend. C’est la raison du test qui suit le message: si nous sommes en mode suspension, le tick ne fera rien d'autre que d’envoyer le message pour jouer le son.

Sinon, l’angle de l’aiguille est calculé. Rien de bien complexe, je voulais seulement que l’aiguille avance par pas d’une demi-seconde, d’où les calculs en ms et non en secondes.

Enfin, le tick déclenche deux PropertyChanged (légèrement modifiés dans MVVM Light). Concernant l’aiguille, c’est la partie transmission d’un message en plus du PropertyChanged qui nous intéresse. En effet, le zéro de l’aiguille a été obtenu en début de projet en appliquant une rotation sur le dessin original. Pour rendre les choses plus propres encore, cette rotation “originelle” a été transformée en ressource stockée dans la Vue.

De deux choses l’une, soit nous dupliquons cette valeur dans le ViewModel afin qu’il retourne un angle directement utilisable, soit le ViewModel ne retourne qu’un angle “pur”, à l’interface de faire l’adaptation.

Bien entendu c’est la seconde solution qui est retenue. Stocker des valeurs propres à l’interface dans le ViewModel est une hérésie... Le ViewModel calcule un angle idéal, propre. Le même ViewModel pourrait servir à plusieurs Vues, chacune possédant ses propres “bricolages” d’affichage.

Du coup, on ne peut pas binder directement la valeur de la rotation de l’aiguille à la propriété angle du ViewModel. Il faudra adapter cette valeur avant de s’en servir. Un convertisseur alourdirait la programmation sans vraiment être une solution satisfaisante.

Nous utilisons ainsi la propriété de RaisePropertuChanged de MVVM Light de pouvoir (ce n’est pas obligé) transmettre un message en même temps que d’appeler PropertyChanged. La Vue va s’abonner à ce message et sera ainsi notifiée comme par un binding mais en gardant la main pour adapter la valeur reçue.

Regardons le code intégré à la Vue. Il n’y a que celui-là, placé dans le constructeur. La Vue peut posséder du code du moment que celui sert à traiter l’interface, c’est le but ici :

```
public MainPage()
    {
        InitializeComponent();
        Messenger.Default.Register<GenericMessage<string>>(this,
            m=>
             {
```

```
            if (m.Content != "TICK") return;
            this.meTick.Stop();
            this.meTick.Play();
          });
    Messenger.Default.Register<PropertyChangedMessage<double>>(this,
        m=>
          {
            if (m.PropertyName != "SecondAngle")return;
            ((CompositeTransform)
              Needle.RenderTransform).Rotation =
                (double)Resources["InitialNeedleAngle"] +
                m.NewValue;
          });
}
```

Il s’agit du constructeur de la Vue. Les deux messages auxquels elles s’abonnent sont :

GenericMessage<string> pour le message de “TICK” qui déclenche le MediaElement

et PropertyChangedMessage<double> pour la valeur de l’angle.

On voit que les messages sont filtrés, le message générique utilise une chaine “TICK” pour s’assurer qu’il va bien traiter le message qui lui est destiné. Le message de changement de propriété est filtré par le type qu’il reçoit (un double) et ensuite par contrôle du nom de la propriété qui vient de changer (“SecondAngle”).

Le calcul de la position de l’aiguille est simple : on récupère la valeur transmise par le message, on l’ajoute à l’offset stocké en ressource (InitialNeedleAngle) et on le stocke dans la Rotation du CompositeTransform du RenderTransform de l’objet Needle (aiguille).

Pour ce qui est du texte affiché sous l’aiguille, le temps écoulé, j’ai choisi de vous montrer la méthode la plus directe : le ViewModel expose une propriété ElapsedTime, de type string qui est calculée à la volée en fonction de la date de départ du chronomètre (initialisée par le passage à l’état Running) et la date courante puis mise en forme par un String.Format.

Comme cette propriété n’est pas mise à jour mais qu’elle calcule sa valeur quand on la lit, elle ne peut pas émettre elle-même de signal PropertyChanged... C’est pourquoi vous voyez une telle notification en fin du code de la méthode timer_Tick un peu plus haut. C’est à chaque battement d’horloge qu’on simulera un property changed de Elapsed time pour indiquer au data binding de la Vue qu’il faut venir lire une nouvelle valeur...

Et bien entendu, dans ce cas, la propriété Text du TextBlock est bien liée par data binding à la propriété ElapsedTime du ViewModel selon la méthode classique :

```
<TextBlock x:Name="txtElapsed"
  Text="{Binding ElapsedTime, Mode=OneWay}"
  ..... />
```

CONCLUSION

Il faut bien conclure un jour... Mai sil reste des choses à découvrir dans cette petite application !

C’est pourquoi je vous fourni son code complet (mais à vous d’installer MVVM Light), mais aussi le fichier EPS d’origine, dans le cas où, par volonté d’auto formation, vous souhaiteriez parcourir à nouveau le chemin mais seul, en partant de zéro pour vérifier que vous avez bien intégré tous mes conseils !

SOMMAIRE

Code Source 309

Préambule 310

MEF 310

MVVM 311

Jounce vs MVVM Light 311

- Différences et similitudes 312
- Pourquoi utiliser MEF 315
- Quel choix au final ? 315

Que propose Jounce ? 316

La Librairie 318

- Présentation 318
- Le code 319
  - Le Tree Map et les espaces de noms 319
  - L'organisation du code 322
  - Graphe des dépendances 323
  - L'organisation des espaces de noms 326

Les bases 327

- L'architecture globale d'un projet 328
  - Les références à MEF 329
  - La référence à Jounce 329
  - App.xaml, Application Service, et messagerie 330
  - La MainPage. Connexion au ViewModel et Blendabilité. 333
    - MainPage.xaml 333
    - Le support des données de conception 334
    - Le code de MainPage 339
      - MainPage.Xaml 339
      - MainPage.Xaml.cs 339
      - L'interface du ViewModel 340
      - Le ViewModel « réel » 341
      - Le ViewModel de design 342
  - Point intermédiaire 342

L'essentiel sur … 343
L'Application Service 343
Marquage d'une Vue 345
Marquage d'un ViewModel 346
BaseViewModel, BaseNotify et IViewModel 346
ViewModelRouter 351
Créer des routes 356
Point Intermédiaire 357
BaseEntityViewModel 357
L'exemple BaseEntityVM 359
Le résultat final 359
Le code, généralités 360
L'interface du ViewModel 360
Le ViewModel de conception 361
Le ViewModel de runtime 362
La région IMainViewModel 363
La région des validations 365
La région du constructeur 368
La région du code privé 369
La Vue – Code-behind 371
La Vue – Xaml 371
Point Intermédiaire 373
Les Commandes 373
L'exemple du clic sur un Rectangle 377
L'exemple du Slider 380
ActionCommand 383
Le code de l'exemple 386
Point intermédiaire 392
La messagerie EventAggregator 392
L'exemple 394
Le visuel 394
Le code 396

L'émetteur 397
Le récepteur 397
Navigation simplifiée : Event Aggregator et ViewRouter 399
Naviguer ? 400
ViewRouter 400
Le mécanisme 401
L'exemple 403
Les régions 404
L'exemple 405
Chargement de XAP 408
Logger personnalisé 411
Créer un Logger personnalisé 411
Workflows 417
Qu'est qu'un Workflow Jounce ? 420
L'exemple 421
Le visuel 421
Le code 421
VSM Aggregator et GotoVisualState 424
GotoVisualState 426
Exemple 427
Visual State Aggregator (VSA) 430
Conclusion 435

TABLE DES FIGURES ET DES TABLEAUX

Figure 1 - Tree Map des namespaces (en IL) ... 320
Figure 2 - le tree map (en nombre de classes) ... 321
Figure 3 - L'oganisation des sources de Jounce ... 322
Figure 4 - Graphe des dépendances ... 325
Figure 5 - Le template Jounce ... 327
Figure 6 - L'arbre du projet template ... 328
Figure 7 - MainPage par défaut ... 334
Figure 8 - Le mécanisme des données de conception ... 336
Figure 9 - L'application BaseEntityVM en cours d'utilisation ... 359
Figure 10 - InvokeCommandAction et le Slider ... 383
Figure 11 - ActionCommand. Affichage de l'application ... 384
Figure 12 - ActionCommand. Sélection d'une personne ... 385
Figure 13 - ActionCommand. Le bouton a été cliqué ... 386
Figure 14 - Event Aggregator - Envoi de message ... 395
Figure 15 - Event Aggregator - Exception non gérée ... 396
Figure 16 - Logger personnalisé ... 417
Figure 17 - Workflow ... 421
Figure 18 - Mr Smiley est content ! ... 428
Figure 19 - Mr Smiley est triste ! ... 429
Figure 20 - VSA – Panel A (B désactivé) ... 430
Figure 21 - VSA - Panel B activé ... 431

Tableau 1 - Similitudes et Différences entre Jounce et MVVM Light ... 313

CODE SOURCE

Ce livre blanc est accompagné du code source complet des exemples. Si vous le recevez sans ces derniers il s'agit d'une copie de seconde main. Téléchargez l'original sur www.e-naxos.com/blog.

Décompressez le fichier Zip dans un répertoire de votre choix. Attention les projets nécessitent Silverlight 4 et Visual Studio 2010 au minimum, Expression Blend 4 est conseillé mais non obligatoire. Les exemples utilisent tous le framework Jounce téléchargeable sur CodePlex (voir l'adresse dans le Préambule) mais possèdent leur propre copie de la dll et peuvent être exécutés sans installation de Jounce préalable.

Code des exemples fournis sous la forme d'une solution contenant les divers projets.

Solution ODJounceSamples.sln

Exemple1 BasicJounceApp

Exemple2 BaseEntityVM

Exemple3 SLInvokeCommand (dans une solution à part, n'utilise pas Jounce)

Exemple4 CommandDemo

Exemple5 EventAggregator

Exemple6 SimpleNavigation

Exemple7 SimpleNavitationWithRegion

Exemple8 CustomLogger

Exemple9 Workflow

Exemple10 GotoVisualState

Exemple11 VSMAggregator

PREAMBULE

Jounce est un nouveau framework (ou toolkit) créé par *Jeremy Likness* (un pair MVP Silverlight travaillant aux USA pour Wintellect) et publié sur CodePlex (jounce.codeplex.com) où le toolkit est défini de la façon suivante :

> ***Jounce est un framework pour Silverlight ayant pour but de fournir des blocs de base pour la construction d'applications de gestion et d'entreprise modulaires qui suivent le modèle MVVM et qui utilisent MEF (Managed Extensibility Framework). Jounce s'inspire de frameworks existants que vous êtes invités à découvrir, dont Prism.***

J'invite les lecteurs ne connaissant pas MEF à lire mon précédent livre blanc « MEF et Silverlight 4+ » publié sur Dot.Blog (article PDF de +70 pages accompagné d'exemples de code).

> *On notera que « Jounce » est un verbe anglais qui signifie « secouer » et qui a pour synonymes « Jolt, Bounce, Jar, Jerk ». Si cet article n'est ni un cours de Jerk ni une invitation à sautiller sur votre siège, on voit poindre une légère provocation chez Jeremy en nommant son toolkit « secouer », comme une façon de secouer le petit monde tranquille des toolkits tels que Caliburn, Prism ou MVVM Light. Si je n'ai pas eu l'occasion de lui poser la question directement, je suis certain que le choix de ce verbe pour désigner son toolkit n'est pas innocent !*

Livre Blanc ?

J'ai écrit beaucoup d'articles et aussi des livres. Les « articles » que j'écris depuis quelques années sont généralement des « monstres » de près de cent pages. Leur but est faire la présentation d'un sujet, à la fois pour transmettre un savoir technique et pour permettre au lecteur de faire des choix éclairés sur certaines technologies. Diffuser une information au public pour l'aider à prendre des décisions est la définition même de « livre blanc ». Et puis un article fait quelques pages tout au plus, pas une centaine. A de telles tailles on est déjà dans le monde du livre (les pages A4 représentent plus d'une page de livre).

C'est pour cela que le présent document se présente comme un livre blanc et non plus un simple « article », terme finalement trop trompeur au regard de la taille du document final.

MEF

MEF est un framework d'inversion de contrôle de type injection de dépendances qui est intégré au Framework .NET depuis la version 4.0 (et dans Silverlight 4.0).

Comme précisé dans le préambule ci-dessus, j'ai écrit dernièrement un très long article sur ce sujet et je ne reviendrai pas ici sur les notions, explications et exemples développés dans cet article téléchargeable sur Dot.Blog.

MEF fait partie de .NET et permet d'écrire des applications où chaque partie est indépendante des autres, MEF se chargeant de les coupler au dernier moment. MEF ressemble à Unity tout en ayant la particularité d'être plus orienté vers la modularité externe (de type plugin) et de faire partie de .NET là où Unity n'est encore qu'un projet du groupe Design & Patterns de Microsoft.

MVVM

MVVM est un modèle de programmation visant à découpler l'interface utilisateur du code de commande de celle-ci. MVVM s'inscrit dans la série des patterns de type MVC et autres MVP.

Il se trouve que j'ai aussi écrit un très long article sur MVVM et je renvoie le lecteur à ce dernier s'il souhaite comprendre dans le détail l'application de ce pattern.

« MVVM avec Silverlight »

http://www.e-naxos.com/Blog/post.aspx?id=c2053091-cd46-4523-aa37-08ba70e37c23

Les objectifs de MEF et de MVVM se rejoignent tout en étant de nature finalement assez différentes. MVVM est un pattern qui vise uniquement au découplage Vue / ViewModel, MEF vise à la modularisation globale de toute l'architecture d'une application. Qui peut le plus, peut le moins, dit le proverbe. De fait, MEF est un excellent support de base pour appliquer MVVM. MEF est un toolkit (une implémentation concrète) imposant un style, c'est finalement un design pattern sans en dire le nom, MVVM se positionne comme un design pattern, donc sans implémentation. Le mariage de MEF et MVVM devient naturel quand on comprend cette complémentarité (l'un le code, l'autre le pattern, l'un global et générique, l'autre ciblant uniquement une partie spécifique de l'architecture).

JOUNCE VS MVVM LIGHT

Inéluctablement la question se posera, surtout chez le lecteur qui suit mes articles et donc connait MVVM Light : Lequel de ces toolkits est le « mieux » ?

MVVM Light est un toolkit conçu pour aider à l'application de MVVM. Très simple, focalisé sur cette tâche, c'est un excellent toolkit utilisable avec Silverlight, WPF et Windows Phone 7.

J'ai aussi traité MVVM Light dans un très long article que j'invite le lecteur à télécharger s'il désire se faire une idée précise de ce toolkit (« Appliquer la pattern MVVM avec MVVM

Light », http://www.e-naxos.com/Blog/post.aspx?id=e8b8964e-10e8-426d-b774-cc750cf76fe9).

J'aime beaucoup MVVM Light car ce toolkit est simple, facile à prendre en main, et qu'il répond aux besoins de base. Il est conçu pour autoriser la blendabilité, c'est-à-dire la capacité à fournir des données de design sous Expression Blend. On peut télécharger le toolkit sur CodePlex (http://mvvmlight.codeplex.com/).

Le présent Livre Blanc est-il une façon de dire qu'il est aujourd'hui préférable d'utiliser Jounce ?

Pas forcément.

Chaque toolkit possède ses avantages et le choix de l'un ou l'autre dépendra du contexte. Jounce est malgré tout beaucoup plus puissant et riche que MVVM Light.

Le plus simple est de brosser un rapide tableau des similitudes et des différences principales…

Différences et similitudes

Le tableau ci-dessous liste les principales différences et similitudes entre les deux toolkits.

Topic	Jounce	MVVM Light	Commentaire
Support de MVVM	Oui	Oui	Les deux toolkits visent à simplifier l'écriture d'applications suivant le pattern MVVM
Simplicité de prise en main	Oui	Oui	Les deux toolkits sont simples à prendre en main
Simplicité d'implémentation	Oui	Oui	Les deux toolkits ont un code léger et n'occupent que peu de place dans l'application compilée
Code source	Oui	Oui	Les deux toolkits sont des projets CodePlex fournis avec code source
Gratuité	Oui	Oui	Ce sont des projets ouverts et gratuits, autant au niveau utilisation qu'au niveau déploiement
Blendabilité	Oui	Oui	Les deux toolkits offrent un support pour les données de design sous Blend
Indépendance	Oui	Oui	Jounce se base sur MEF qui fait partie du Framework .NET, il n'y a donc aucune dépendance à des projets externes.
Support de MEF	Oui	Non	Le mariage MVVM Light et MEF ne fonctionne pas très bien. Jounce est

			totalement adapté à la situation en revanche
Gestion des Regions	Oui	Non	MVVM Light ne couvre pas cet aspect. Jounce utilise un principe proche de celui de Prism
Gestion de Workflow	Oui	Non	MVVM Light ne couvre pas cet aspect. Jounce propose une solution originale qui sera développée ici.
Support WPF et WP7	Non	Oui	Jounce ne supporte que Silverlight actuellement
Support de la navigation	Oui	Non	MVVM Light ne supporte pas cet aspect
Chargement dynamique de modules	Oui	Non	Comme expliqué plus haut, MVVM Light fonctionne mal avec MEF alors que Jounce se base sur MEF
Service de Log	Oui	Non	MVVM Light ne prend pas en charge cet aspect
Messagerie	Oui	Oui	Les mises en œuvre diffèrent, les possibilités restent proches
Recherche de ViewModel	Oui	Non	La communication entre les ViewModels sous MVVM Light repose uniquement sur la messagerie. Jounce offre un procédé plus direct (le « Router »).

Tableau 1 - Similitudes et Différences entre Jounce et MVVM Light

Bien que les deux toolkits visent les mêmes buts, ils le font de façon différente. On pourrait dire que Jounce a une écriture plus « moderne » que MVVM Light : il traite les problèmes en utilisant les possibilités les plus récentes du Framework Silverlight 4.0 alors que MVVM Light possède une approche plus classique visant la portabilité WPF et WP7. C'est un avantage de MVVM Light là où Jounce ne supporte que Silverlight.

Jounce règle aussi certains problèmes que MVVM ne traite pas, comme la synchronisation des tâches asynchrones qui est un peu le calvaire du développeur sous MVVM (avec les messages qui se « baladent » de façon asynchrone) surtout dès qu'on ajoute une gestion de données de type WCF Ria Services par exemple (asynchrone par nature) ou même de simples Web Services.

Mais il ne serait pas bien difficile de prendre uniquement le code du Workflow de Jounce et de l'utiliser avec MVVM Light. L'avantage principal de Jounce ne se situe donc pas là, mais bien dans sa parfaite intégration avec MEF et sa richesse globale.

Jounce utilise MEF pour l'injection de dépendance. De fait c'est par MEF que s'effectue le découplage entre Vues et ViewModels, alors que MVVM Light utilise un service Locator limité à cette tâche de séparation Vue / ViewModel. En se basant sur MEF, beaucoup plus puissant et générique, Jounce applique MVVM en tirant profit d'une nouveauté du Framework .NET.

Pour assurer la blendabilité, MVVM Light est obligé d'implémenter des instances statiques des ViewModels dans le Locator. Ce qui est terriblement gênant si on souhaite utiliser MEF. Dans la pratique (et mon précédent article le montre) si on utilise MVVM Light avec MEF on doit le faire en sacrifiant la blendabilité, argument pourtant essentiel de MVVM Light. Jounce règle le problème différemment en utilisant les dernières possibilités de Blend, notamment celle permettant de définir des données de runtime ignorées à l'exécution. C'est à ces détails que Jounce apparait plus « moderne » dans son écriture que MVVM Light dont le code ignore toutes les nouveautés introduites depuis Silverlight 2 ou 3.

Les deux approches sont intéressantes, mais celle de MVVM Light est un peu bloquante dès lors qu'on désire utiliser MEF (sans que cela ne soit impossible). Jounce lève ce problème et cela est indispensable pour assurer un design rapide et efficace de l'interface utilisateur.

Enfin, c'est une évidence mais cela va mieux en le disant, Jounce « force » l'utilisation de MEF qui impose sa propre logique dans le découpage (la modularisation) de l'application alors que MVVM Light reste totalement en dehors de cette problématique et ne propose de résoudre qu'un seul aspect de la modularisation : la séparation Vues / ViewModel. C'est assez pour satisfaire MVVM, mais c'est un peu juste pour assurer une vraie modularisation d'une application de taille moyenne ou plus.

MEF est un élément crucial du Framework en cela qu'il permet une construction réellement modulaire de toute l'application, pas seulement la séparation entre Vues et ViewModels. Ces derniers ne sont pas les seuls « morceaux » d'une application. Un logiciel complet comprend aussi de nombreux services (au sens large) comme le log des erreurs, des modules de calculs, d'impression, des fournisseurs de données, etc… MVVM Light est focalisé uniquement sur le découplage Vue / ViewModel. Jounce, en se basant astucieusement sur MEF offre une gestion bien plus subtile et plus vaste du découplage de toutes les parties de l'application. MEF simplifie aussi la découverte de modules externes (type plugin) là où, évidemment, MVVM Light ne fait rien en ce sens.

> *Choisir entre Jounce et MVVM Light ne se fait pas seulement en fonction des possibilités ou de la stylistique de chacun de ces toolkits, le choix s'opère*

principalement sur l'adoption ou non de MEF et de son modèle particulier de découplage fort entre toutes les parties (ou modules) de l'application.

Si l'on ne désire pas utiliser MEF, l'intérêt de Jounce diminue grandement puisqu'il se base sur MEF…

La portabilité du code sous WP7 et WPF pourra aussi jouer un rôle important dans le choix puisque, pour le moment en tout cas, Jounce n'est fourni que pour Silverlight.

Toutefois cela reste à relativiser grandement. Aucune application un peu sérieuse ne peut être portée directement entre les trois environnements (Silverlight, WP7, WPF) sans de profondes modifications soit de son code, soit de son interface, voire des deux. La portabilité se situe bien plus au niveau des compétences pour développer sous ces trois environnements que dans le code lui-même. Mais certains projets peuvent tirer avantage de la grande proximité de ces environnements et dans ce cas il sera préférable d'utiliser un toolkit portable comme MVVM Light. Mais si l'application commence à grossir, ce dernier sera insuffisant et il faudra opter pour Prism ou Caliburn.Micro.

De fait, l'absence, pour le moment, d'un Jounce pour WPF et WP7 n'est qu'un désavantage tout à fait relatif. Mais il est important de le prendre en compte si on doit concevoir un projet s'inscrivant, même partiellement, dans le cadre d'une telle portabilité.

Pourquoi utiliser MEF

Sans refaire mon article précédent sur MEF, disons que MEF est un choix stratégique pour une application car MEF :

- Est un moyen standardisé d'exposer et de consommer des « composants » ;
- De connecter automatiquement ces composants ensemble dans l'ordre correct de leurs dépendances ;
- Offre un moyen très souple de découvrir et d'utiliser des composants ;
- Supporte un système de métadonnées performant autorisant des requêtes et un filtrage sur les composants ;
- Offre une assistance à la gestion du cycle de vie des composants ;
- Fait partie du Framework.

Dès lors qu'on est sensible à l'un de ces arguments, choisir MEF devient une évidence, si ce n'est une nécessité (gestion de plugins par exemple).

Quel choix au final ?

Au travers de mes différents articles et Livres Blancs j'essaye d'offrir au lecteur les éléments objectifs qui lui permettront de choisir en connaissance de cause. Il n'est pas dans mon

intention de faire des choix stratégiques à sa place, surtout de façon générale sans prendre en compte le contexte particulier de ses développements.

Pour cela j'offre mes services d'audit et de consulting, services qui me permettent de conseiller personnellement une entreprise en fonction de son contexte et de ses problématiques à résoudre. Faire du conseil à distance, globalement, pour tous les lecteurs sans différencier leur contexte et leur problématique friserait l'escroquerie !

Je me garderai donc bien de dire si de Jounce ou MVVM Light l'un est « meilleur » que l'autre. Ce sont tous les deux d'excellents toolkits avec leur originalité propre.

Le choix premier concerne l'adoption de MEF. Si on opte pour ce dernier, Jounce semble mieux adapté par nature, si on n'utilise pas MEF, on peut choisir Jounce ou MVVM Light, tout dépendra du contexte.

A titre personnel j'ai un léger penchant pour Jounce aujourd'hui. Parce que MEF me semble proposer une architecture modulaire tout à fait intéressante pour concevoir de bons logiciels et que choisir MEF pousse à utiliser Jounce plus que MVVM Light. Jounce est aussi bâti selon une approche plus « moderne » (même si ce terme reste flou) que MVVM Light et j'aime sa façon de résoudre les problèmes. En toute objectivité technique, Jounce adresse aussi beaucoup de situations et de problèmes que MVVM Light. Mais en dehors de ces considérations personnelles, c'est au lecteur de faire le choix entre les deux toolkits !

QUE PROPOSE JOUNCE ?

J'ai présenté Jounce rapidement dans le préambule, mais cela ne nous dit pas ce que Jounce apporte réellement.

Voici quelques points importants pour lesquels Jounce apporte une solution simple et élégante :

- Une connexion simplifiée entre les Vues et les ViewModels tout en conservant un découplage fort entre ces deux parties ;
- Une communication simplifiée entre les ViewModels évitant d'avoir à utiliser des messages asynchrones ;
- La création et la consommation de message à la volée ;
- L'utilisation simplifiée et quasi automatiquement de XAP chargés dynamiquement ;
- Une approche simple et efficace de la navigation ;
- La gestion des régions (pour l'affichage des modules)
- La gestion des commandes
- La gestion des logs de l'application
- La gestion de traces permettant de comprendre ce que Jounce fait automatiquement (très utile pour le débogage) ;

- La prise en charge des ViewModels effectuant des opérations de type CRUD[22] avec validation des données ;
- La prise en charge de la synchronisation des tâches asynchrones via la notion de Workflow.

La connexion entre les Vues et les ViewModels est un problème classique avec MVVM. Ce patron impose que le ViewModel ne connaisse pas sa (ou ses) Vue(s), mais pas l'inverse. Là où MVVM Light utilise un *service Locator* assurant un découplage de plus non obligatoire sous MVVM (dans le sens des Vues vers les ViewModels), Jounce préfère un système purement déclaratif et plus direct (quel ViewModel se connecte à quelle Vue) tout en conservant une séparation nette. Les deux approches ont leurs avantages. Jounce est ici plus proche de l'esprit de Prism ou de Caliburn, les deux frameworks dont il s'inspire ouvertement d'ailleurs.

La communication entre les ViewModels est aussi une problématique classique de MVVM. Sachant que les ViewModels ne doivent pas se connaître, le seul moyen « légal » sous MVVM de les faire communiquer est d'utiliser des messages, asynchrones généralement, ce qui complique singulièrement l'écriture du code. Jounce offre une messagerie, mais il permet aussi grâce au « routeur » d'obtenir une référence sur un ViewModel par le nom du contrat qu'il expose. De fait il devient possible pour un ViewModel d'interroger un autre ViewModel directement sans pour autant posséder de référence codée en dur vers ce dernier et surtout en échappant à un dialogue complexe et asynchrone. Cet aspect ainsi que tous ceux présentés dans cette section seront bien entendu détaillés plus loin.

Le chargement dynamique de fichiers XAP est un point essentiel de Jounce. MEF ne propose pas directement de solution sous Silverlight pour découvrir automatiquement les XAP externes (Cf. mon article sur MEF pour plus de précisions), ni même pour les charger. Même si les briques pour le faire existent, il n'existe donc pas de système de découverte automatique des XAP comme MEF le propose pour les modules externes sous WPF. L'interdiction (pour des raisons de sécurité) pour Silverlight d'aller lire le contenu d'un répertoire sur un serveur est à l'origine de cette différence. Jounce ajoute le code nécessaire pour charger facilement un XAP externe depuis un serveur et en découvrir les composants exposés. Il s'agit là d'une fonction essentielle pour assurer la modularité et la réactivité des applications Silverlight. Jounce ne propose pas de système de découverte des XAP externes, pour les raisons de sécurité évoquées plus haut, toutefois, dans mon article sur MEF j'expose

[22] Create, Read, Update, Delete, les quatre opérations de base pour la persistance des données.

une solution qui passe par la gestion d'un catalogue XML des extensions XAP et qui peut facilement être utilisée en conjonction avec Jounce.

Nous verrons plus loin des illustrations des principales fonctions de Jounce, je n'irai donc pas plus loin pour l'instant dans l'exposé des possibilités du toolkit.

LA LIBRAIRIE

Comme je l'avais fait pour MVVM Light, il est intéressant, même rapidement, d'avoir une vision d'ensemble du code de la librairie. Le lecteur pourra aussi comparer les librairies entre elles en se reportant à mon papier sur MVVM Light. J'utiliserai ici le même outil d'analyse de code, à savoir l'excellent NDepend de Patrick Smacchia (www.ndepend.com).

Le but reste de créer un premier « contact » entre le lecteur et le code de la librairie, savoir « à qui on a affaire », « voir sa tête ». Il est bien entendu hors de mon propos d'analyser ici tout le code de Jounce. Il y aurait beaucoup de choses à dire, à apprendre, mais aussi à remanier ou refactorer, comme dans tout code en évolution. Le code source étant librement accessible je laisse aux plus passionnés d'entre vous le loisir d'effectuer un tel décorticage ! Une vue d'ensemble nous suffira ici.

Présentation

Physiquement, Jounce se présente sous la forme d'une unique DLL de 91 Ko: Jounce.dll. Il est donc aisé de l'ajouter à une solution dans un répertoire dédié. Ce que je conseille car si vous utilisez un logiciel de gestion de version vous serez certain d'archiver la bonne DLL de la bonne version avec le code qui l'utilise (très précieux lorsqu'on ressort un logiciel plusieurs mois après l'avoir créé et que toutes les librairies externes ont évolué entre temps).

Le code source est disponible avec des exemples sur le site CodePlex indiqué en début de document.

Il n'y a pas, comme pour MVVM Light, une procédure d'installation bien précise à suivre : il suffit juste d'utilise la dll… Le site propose malgré tout un template de projet Jounce que je vous conseille d'installer. Cela simplifie la création de la coquille de base.

Attention : le template créée automatiquement un sous-répertoire Jounce dans le projet et y place une copie de Jounce.dll. Je n'aime pas trop cet automatisme car, par force, et à moins que le template ne soit mis à jour, la Dll en question ne suivra pas l'évolution des versions de Jounce ce qui peut poser problème dans le futur.

Pour ma part, je préfère disposer d'une copie du code de Jounce que je peux compiler en *Debug* et en *Release*. Une fois le projet créé par le template je m'empresse de remplacer la

Dll par celle que j'ai compilée. Le tout étant versionné sous Subversion. Tout changement ultérieur de la DLL sera conservé, versionné et archivé.

Le code

Commenter le code de chaque classe de Jounce prendrait un livre entier, je ne l'ai pas fait pour MVVM Light, cela sera encore moins possible ici. Mais je vais essayer de vous donner une vision d'ensemble du code et de son organisation, information qui ne se trouve nulle part pour l'instant (d'ailleurs pour MVVM Light, mon article reste le seul à aborder la librairie sous cet angle particulier même un an après sa publication).

Le Tree Map et les espaces de noms

Cette vision du code obtenue selon plusieurs critères (ici le nombre d'instructions IL) permet de visualiser rapidement les classes les plus grosses, généralement les plus importantes fonctionnellement, ainsi que les espaces de noms et leur taille respective.

La figure suivante montre le tree map du code de Jounce du point de vue des espaces de noms. La taille de chaque carreau de la mosaïque est directement liée au nombre d'instructions IL du code compilé correspondant.

Cela permet de voir ce qui a demandé le plus de travail dans la librairie, les endroits où la majorité du code se concentre mais aussi l'ensemble des espaces de noms qui constituent la librairie.

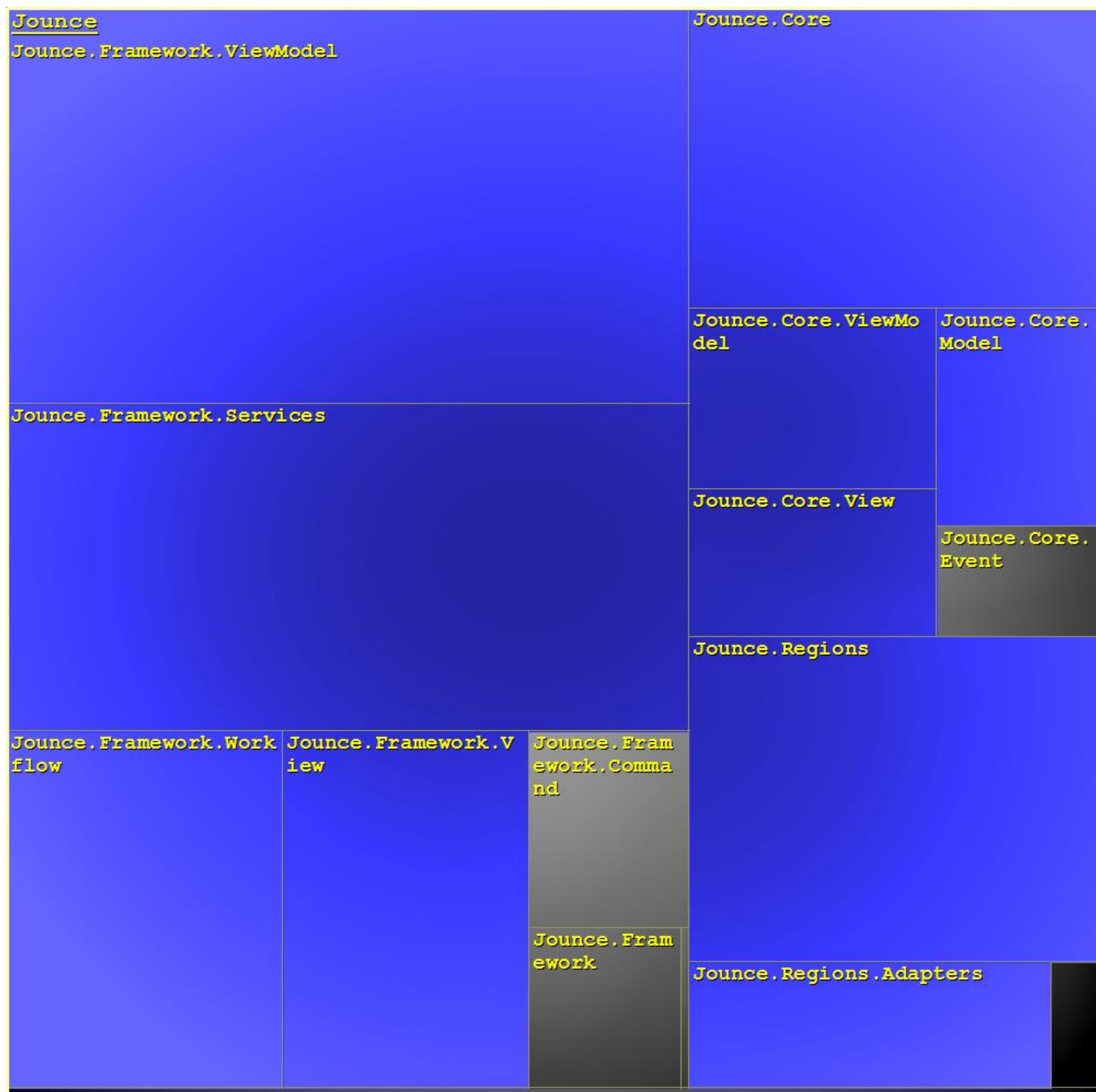


Figure 70 - Tree Map des namespaces (en IL)

On voit nettement que les partie Services et ViewModels représentent une grosse partie de Jounce, et qu'avec la gestion des Regions ainsi que le noyau de base Core on a presque la totalité du code de l'ensemble.

La gestion des View, celle des Workflow, les Adapters, les Commands, bien qu'essentiels fonctionnellement ne prennent pas beaucoup de place. 91 Ko compilé en mode Debug, il faut dire que Jounce n'est pas énorme.

Une autre vision intéressante est donnée par la figure suivante qui reprend le même principe mais en comptant les classes déclarées. On voit que les carreaux de la mosaïque ne sont plus tout à fait les mêmes :

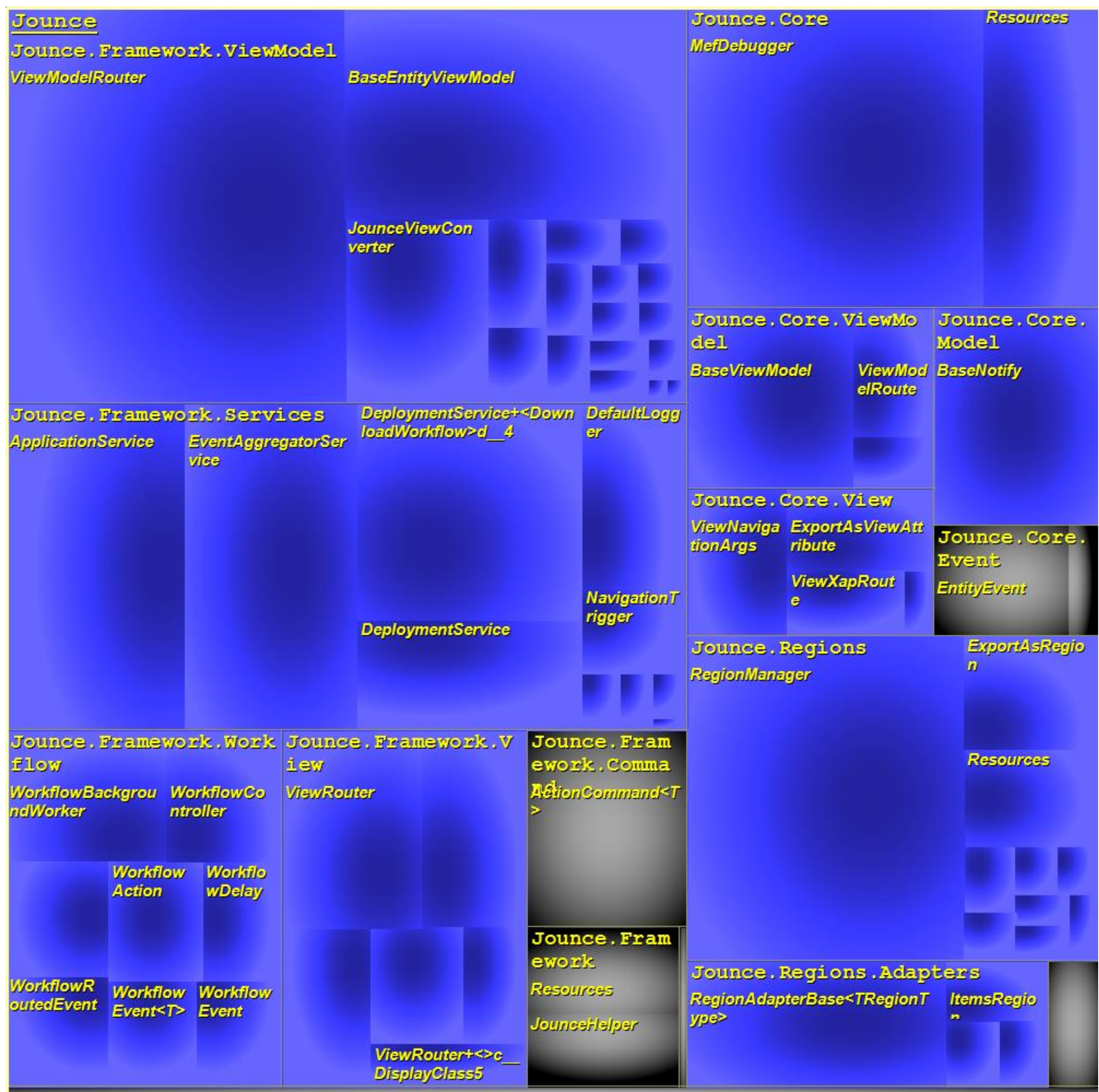


Figure 71 - le tree map (en nombre de classes)

La gestion des ViewModel et des Regions sont les plus grandes, ce qui est conforme à la taille du code de ces espaces de noms. Ils sont donc plus gros, mais déclarent plus de classes, ce qui laisse supposer que la complexité de ces dernières est relativement constante. Quelques exceptions notables : la classe ViewModelRouter de Jounce.Framwork.ViewModel qui a elle seule prend tout le coin supérieur gauche indiquant que cette classe pourrait certainement être *redesignée* (une classe ne doit jamais contenir trop de méthodes, une vingtaine étant une limite approximative). La classe MefDebugger dans le noyau et RegionManager dans la gestion des régions sont un peu dans le même cas. Jounce

s'architecture ainsi autour de quelques grosses classes, de plus petites étant satellisées autour.

L'organisation du code

Avant de regarder les espaces de noms jetons un œil sur l'organisation des sources du projet :

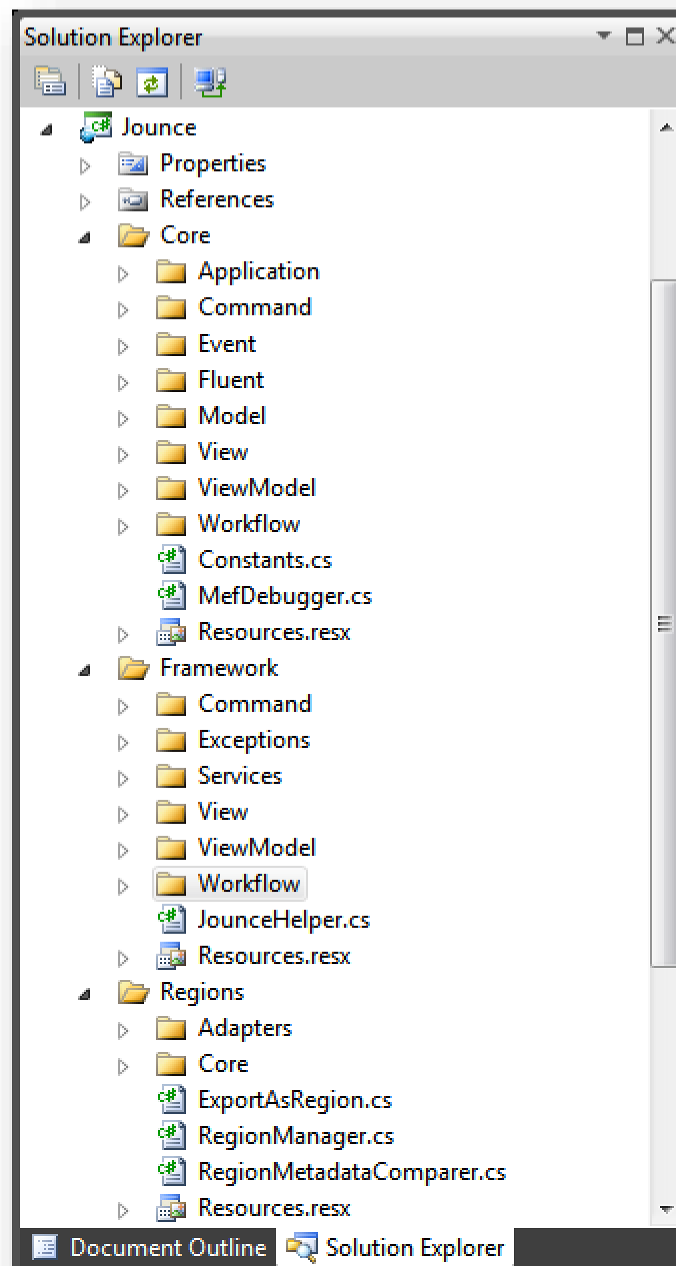


Figure 72 - L'oganisation des sources de Jounce

Le code étant bien organisé on voit clairement apparaître les futurs espaces de noms dans le découpage.

Trois zones se dégagent :

- Core. Le Noyau. Toutes les bases de Jounce sont là.
- Framework. C'est plutôt la couche « pratique ». C'est par exemple ici que sont implémentées des classes utiles se basant sur des types définis de façon abstraite (classes de base, interfaces…) dans le noyau.
- Regions, la gestion des régions, est une partie bien spécifique qui s'ajoute à l'ensemble et qui est du même niveau que Framework (et qui aurait pu y être intégrée, la séparation étant certainement plus historique qu'autre chose, les régions ayant été ajoutées dernièrement).

Il s'agit d'un découpage large, une vue de très haut mais qui permet de savoir comment s'orienter dans le code source selon ce qu'on cherche. Les grandes autoroutes. Reste à connaître aussi les nationales et les départementales !

Graphe des dépendances

Il y a plusieurs angles de vues possibles sur un code, et même en choisissant l'un d'entre eux il y a mille manières de le regarder. Le graphe des dépendances permet d'avoir une vision du couplage au sein de la librairie. J'ai choisi ici une vue montrant le couplage entre les espaces de noms, la taille des blocs étant reliée au nombre de flèches entrantes, les espaces de noms les plus utilisés étant ainsi les plus gros. La taille des flèches est liée au nombre de types en jeu. Plus les liaisons sont épaisses, plus il y a de classes.

Je ne vous commenterai pas chaque bloc ni chaque lien, ces différents graphiques ont pour seul but de vous fournir différentes vues du code source pour vous aider à « prendre contact » avec ce dernier.

Je pense qu'il est toujours intéressant d'avoir une idée, au moins large, de ce « qu'il y a dans la boîte ».

Ce qui est intéressant de noter :

- On voit se dessiner assez facilement la rupture entre ce que Jounce appelle son noyau (Core) et sa partie Framework (plutôt à gauche sur le graphique).
- L'essentiel des flèches va dans le même sens : la structure est propre et se dirige du plus haut niveau vers le noyau sans allers et retours ni liens de traverse, c'est un signe de bonne conception.

- Certains éléments ont des liens très serrés et nourris comme Jounce.Regions avec Jouce.Region.Adapters. La logique de nommage de ces espaces de noms est cohérente avec les relations qu'ils entretiennent.
- La gestion des ViewModel est liée à celle des commandes et à des éléments du noyau mais n'a pas de relation avec le reste. L'écriture de Jounce est bien structurée, ce n'est pas du code spaghetti …
- De même la gestion des Vues (Jounce.Framework.View) n'a que peu de relations en dehors de celles qu'elle tisse avec les éléments du noyau. Un lien la relie aux ViewModels (mais à la partie Core de cette gestion). MVVM autorise une Vue à connaître son ViewModel, pas l'inverse, les relations du graphique montrent que le respect du pattern s'inscrit dans le code source lui-même de la librairie. C'est un gage de qualité et de cohérence.

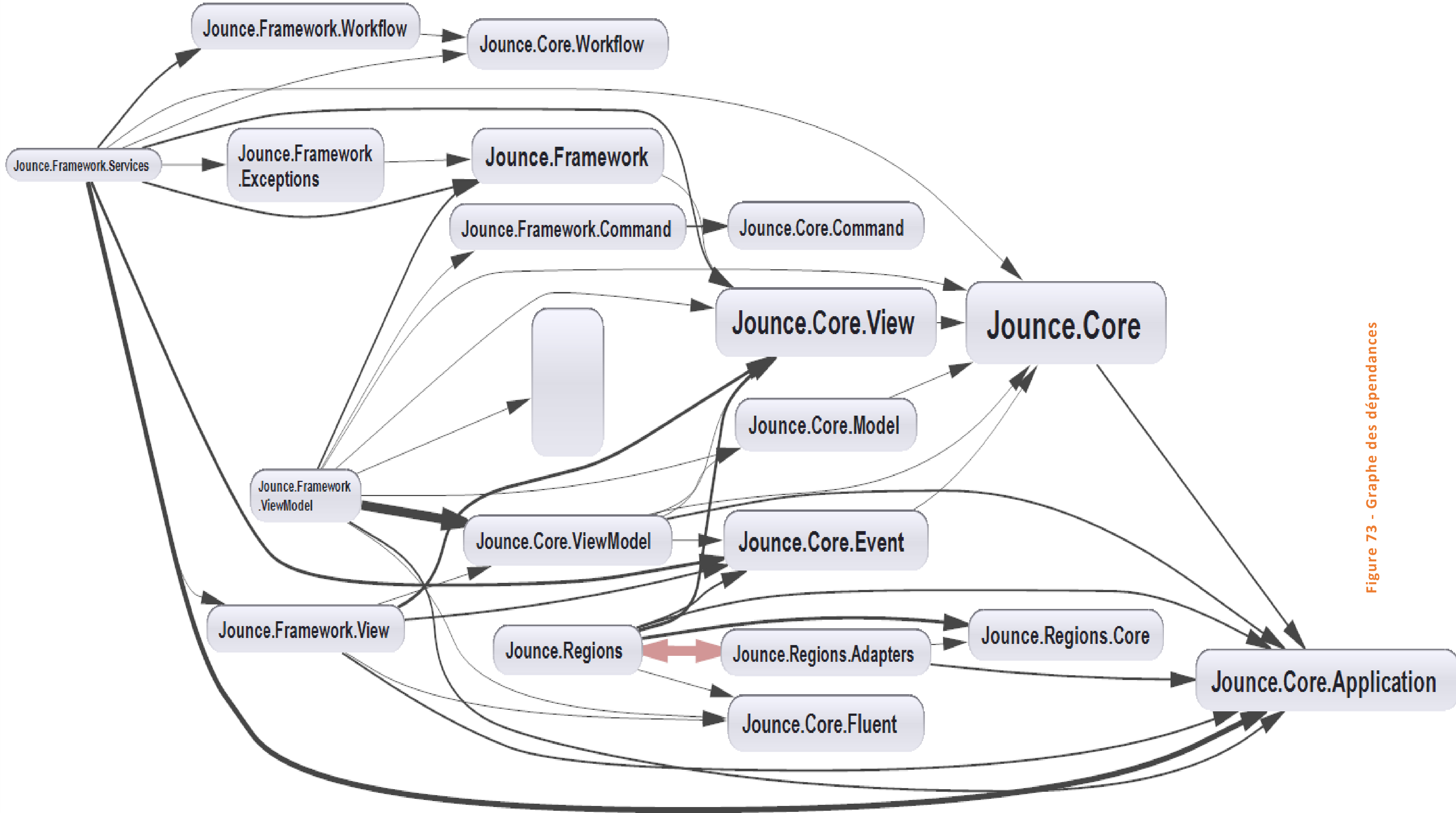


Figure 73 - Graphe des dépendances

L’organisation des espaces de noms

Jounce s’organise en plusieurs sous-espaces de noms, le principal étant Jounce lui-même.

Les principaux espaces de noms sont découpés en sous-espaces, Core et Framework étant un peu en reflet, ce qui est logique vu l’angle d’attaque choisi par Jeremy : un noyau définissant les bases, un « framework » implémentant concrètement les principes posés par le noyau.

Jounce.Core est le noyau de Jounce.

Le sous-espace Application définit par exemple les interfaces des services exposés par l’Application Service. Dans l’espace Jounce.Framework.Services on trouvera les classes qui implémentent ces services.

Dans le sous espace Jounce.Core.Command on trouve aussi une interface, celle qui enrichit ICommand de Silverlight. Naturellement, dans Jounce.Framework.Command on retrouvera une classe (ActionCommand) qui implémente cette interface.

Jounce.Core.Event définit des classes et des interfaces, qui là encore serviront à construire des classes comme EventAggregatorService ou le ViewRouter.

Nous verrons toutes ces classes à l’œuvre dans la suite de ce document, je ne vais donc pas vous abreuver de noms qui ne vous disent rien pour l’instant. L’essentiel étant d’avoir compris comment les espaces de noms s’organisent dans le code de Jounce pour savoir où regarder. Les documentations ne sont pas toujours très claires, ni très abondantes pour Jounce. Lire le code reste la meilleure façon de se former réellement à la librairie et d’en comprendre les mécanismes. Il est donc essentiel d’avoir une première idée, un plan général qui permet ensuite de se promener et d’aller à la découverte des petites rues.

La seule exception à la règle de correspondance entre Core et Framework est l’espace Jounce.Regions qui ne s’occupe que des régions d’affichage. Cet espace est conçu comme une réplique isolée de Jounce et on y retrouve un sous-espace Core posant les bases et une série de classes directement dans Jounce.Regions qui les utilise.

Une fusion entre le Core de Regions et le Core de Jounce serait évidente dans un refactoring de la librairie, de même qu’intégrer le reste de Regions à Jounce.Framework.

L’isolation de Jounce.Regions tient certainement au fait que ce bloc a été ajouté tardivement et que Jeremy ne voulait pas lors de sa mise au point déstabiliser ce qui existait déjà et qui commençait à être utilisé par des développeurs.

Après cette visite rapide du code de Jounce il est temps de passer au plus important : qu'est -ce que Jounce ? Comment marche-t-il ? Que pouvez-vous en attendre ? Les quatre-vingt-dix pages (environ) qui suivent sont là pour répondre à ces questions !

LES BASES

Plutôt que de faire une redite de la documentation de Jounce qui est très complète (et que vous trouverez ici : http://jounce.codeplex.com/documentation), je vous propose de voir immédiatement une première démonstration qui nous permettra de poser les bases.

Créer une application Jounce est extrêmement simple, et peut se faire en quelques manipulations évidentes, ce que la documentation explique très bien. Je vais donc partir un cran plus loin en utilisant directement le Template de projet Jounce qui peut être téléchargé à part sur le site CodePlex du toolkit.

L'installateur du template s'appelle JounceApplication.vsix, il suffit de le télécharger puis de l'exécuter pour qu'un nouveau template « Jounce Application » apparaisse dans Visual Studio.

Pour notre première démonstration je créé directement un nouveau projet à partir du template. Attention ce dernier apparait sous « **Visual C#** » et non sous « Silverlight » comme le montre la capture ci-dessous (figure 5).

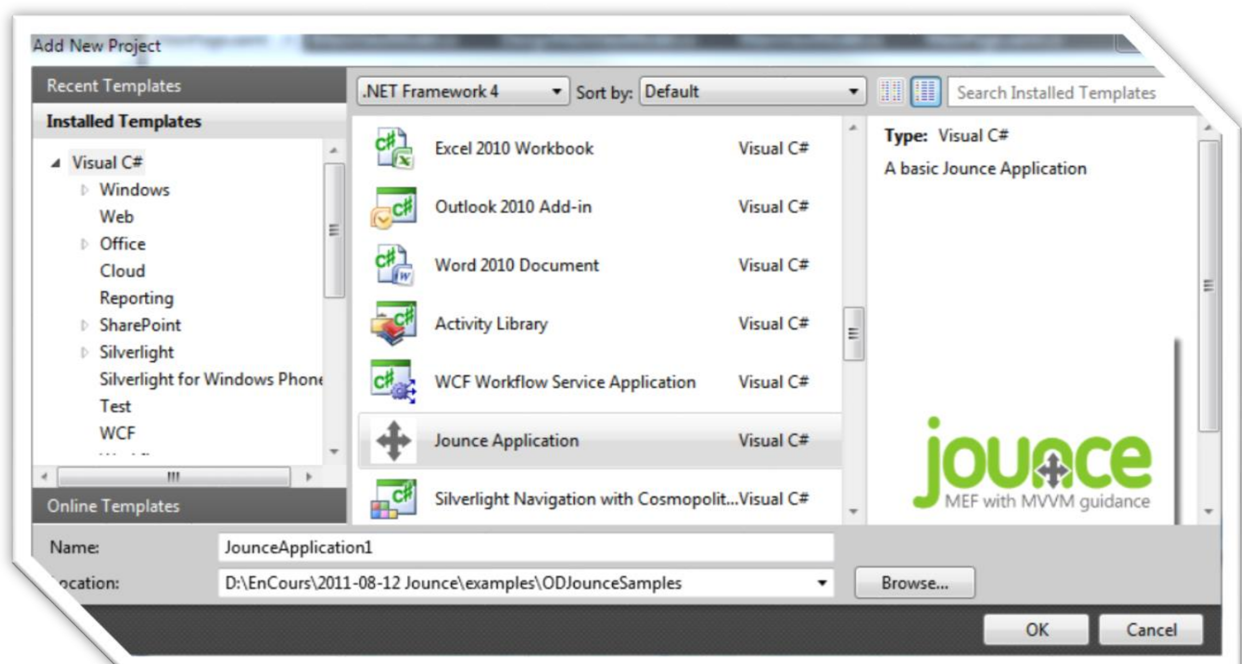


Figure 74 - Le template Jounce

L'avantage principal de partir de cette coquille est bien entendu un gain de temps lorsqu'on doit créer un nouveau projet utilisant Jounce. Les références à Jounce et à MEF sont posées,

les répertoires de l'application sont créés proprement, et un shell accompagné de son ViewModel est déjà en place.

L'autre avantage de template Jounce est qu'il permet, sans écrire une ligne de code, de voir en action les principes fondamentaux de Jounce, ce qui est parfait pour le premier exemple de ce Livre Blanc !

Ainsi, je vous invite dans un premier temps à une visite guidée dans ce simple template qui, vous allez vous en rendre compte, montre déjà beaucoup de la richesse de Jounce.

L'architecture globale d'un projet

La figure ci-dessous montre l'arbre du projet exemple juste après sa création depuis le template Jounce (figure 6):

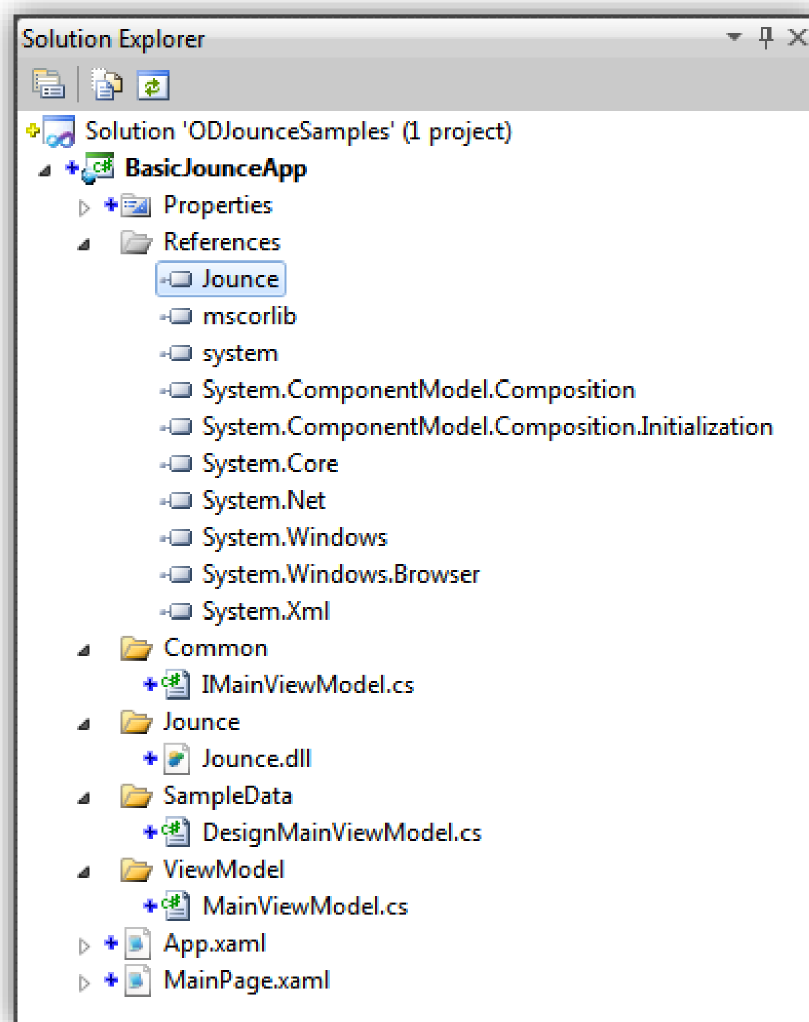


Figure 75 - L'arbre du projet template

Globalement il s'agit naturellement d'un projet Silverlight assez classique avec une MainPage, un App.Xaml et quelques références.

Mais en y regardant de plus près nous pouvons découvrir tout ce qui fait la logique de Jounce :

Les références à MEF

Le projet référence les deux DLL de MEF :

- System.ComponentModel.Composition, espace de noms principal de MEF
- System.ComponentModel.Composition.Initialization, module spécifique de MEF pour Silverlight lié au déploiement des XAP.

Utiliser Jounce, c'est utiliser MEF. Comme déjà expliqué, je ne reviendrai pas ici sur les avantages ou le fonctionnement de MEF, mais le lecteur qui ne connaîtrait vraiment rien à MEF est invité à lire mon précédent article sur ce sujet.

La référence à Jounce

Le template de Jounce utilise une stratégie bien à lui pour s'assurer de la présence de la DLL de Jounce (84 Ko, donc très légère) : un répertoire Jounce est créé dans l'application avec pour contenu une copie de la DLL de Jounce. La référence pointe cette version de la DLL.

C'est pratique, cela assure que la DLL est toujours présente dans le projet, toutefois dans le cadre d'une organisation plus stricte il s'agit là d'un exotisme que certains trouveront discutable.

C'est pratique et cela peut être utile pour versionner la DLL en même temps que l'application (ce qui la met à l'abri de *breaking changes* dans une version ultérieure de Jounce) mais à vous de décider si cette organisation vous convient.

On note que Jounce est une petite DLL qui ne prend vraiment pas beaucoup de place. Toutefois il faut ajouter les 279 Ko de MEF sans lequel Jounce ne marche pas.

Ainsi, le poids réel de Jounce est d'environ 360 Ko, mais avec le bénéfice de toute la logistique MEF.

Pour comparaison le poids de MVVM Light n'est que de 20 Ko et monte à 30 Ko si on inclut les « extras » (DispatcherHelper et EventToCommand). Mais on ne dispose pas de la puissance de MEF, juste un toolkit pour MVVM.

La taille de Jounce est donc à peine supérieure à celle de MVVM Light (une cinquantaine de Ko en plus), ce qui reste négligeable et ne peut pas constituer un élément de choix objectif.

App.xaml, Application Service, et messagerie

Silverlight possède un *Application Service* qui permet de s'insérer dans le traitement de l'objet Application tout en pouvant continuer à utiliser directement la classe Application (et donc sans en créer des classes dérivées). Il s'agit d'un ajout de Silverlight 3 qui est passé « sous le radar » et dont presque personne n'a parlé.

Pourtant il s'agit d'un ajout intéressant.

Jeremy a eu la bonne idée d'exploiter ce « hook » pour vider totalement de son code App.xaml en en confiant la gestion à un service propre à Jounce. C'est à ces petits détails que je considère l'implémentation de Jounce plus « moderne » que celle de MVVM Light : les particularités de Silverlight, ses petites subtilités sont bien utilisées.

Ainsi, dans sa version la plus simple, une application « *Jouncifiée* » commence par ajouter une référence à Jounce.dll puis à modifier App.Xaml de la manière suivante :

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
         xmlns:Services="clr-namespace:Jounce.Framework.Services;assembly=Jounce"
         x:Class="Jounce.App"
         >
   <Application.ApplicationLifetimeObjects>
      <Services:ApplicationService/>
   </Application.ApplicationLifetimeObjects>
   <Application.Resources>
   </Application.Resources>
</Application>
```

Vous pouvez supprimer tout ce qui se trouve dans App.xaml.cs à l'exception de la partie appelant InitializeComponent.

En s'insérant malicieusement, mais subtilement, dans la chaîne ApplicationLifeTimeObjects de l'objet Application, Jounce s'occupe de tout, même de la gestion des exceptions non gérées.

Bien entendu le template Jounce gère tous ces détails. Mais le App.xaml.cs fourni n'est pas totalement vide, il se présente ainsi :

```
namespace BasicJounceApp
{
   public partial class App : IEventSink<UnhandledExceptionEvent>
   {
      [Import]
      public IEventAggregator EventAggregator { get; set; }
```

```
    public App()
    {
        this.Startup += this.Application_Startup;
        InitializeComponent();
    }

    private void Application_Startup(object sender, StartupEventArgs e)
    {
        CompositionInitializer.SatisfyImports(this);
        EventAggregator.SubscribeOnDispatcher(this);
    }

    public void HandleEvent(UnhandledExceptionEvent publishedEvent)
    {
        MessageBox.Show(publishedEvent.UncaughtException.ToString());
        publishedEvent.Handled = true;
    }
}
```

D'abord on voit que la classe App supporte une interface IEventSink générique spécialisée pour la classe UnhandledExceptionEvent.

Ce mécanisme est à cheval entre MEF, Jounce et Silverlight. Une utilisation encore une fois astucieuse des briques de la boîte de Lego…

IEventSink appartient au mécanisme de messagerie de Jounce dont l'implémentation diffère beaucoup de celle de MVVM Light, plus simple et plus isolée.

Une classe souhaitant écouter des messages doit tout simplement supporter IEventSink. Elle doit préciser le type des messages qu'elle désire recevoir, ici UnhandledExpcetionEvent. Si une classe veut écouter plusieurs types de messages, il lui suffit d'ajouter autant de fois que nécessaire le support de l'interface IEventSink dans sa déclaration (avec un type différent à chaque fois bien entendu). Le type des messages est totalement libre, vous pouvez ainsi créer des classes très spécialisées véhiculant tout ce qui est nécessaire pour traiter le message.

Cette interface oblige à l'implémentation de HandleEvent, seul élément du contrat. La signature de cette méthode doit reprendre le type déclaré au niveau de la déclaration de la classe et de son support de IEventSink. Si la classe a déclaré plusieurs IEventSink pour des types différents, elle devra exposer autant de méthodes HandleEvent qui seront différenciées uniquement par leur signature (le type de l'argument reçu).

Ensuite on note la présence d'une importation typiquement MEF :

```
[Import]
public IEventAggregator EventAggregator { get; set; }
```

Ici, la classe App réclame à MEF l'importation d'une instance supportant IEventAggregator. Le principe est repris de Caliburn de Rob Eisenburg.

Jounce offre un service de messagerie permettant de transmettre n'importe quelle instance dans le message, mais de façon finalement plus simple que le Messenger de MVVM Light. En revanche le mécanisme se découpe en deux phases :

1. Le support de IEventSink pour écouter les messages d'un type particulier (ou de plusieurs) avec l'implémentation du contrat.
2. La demande d'écoute qui démarre la communication en quelque sorte

Le premier point a été vu juste à l'instant (implémentation de IEventSink), c'est le second point qui oblige à obtenir *l'Event Aggregator*. En effet, pour indiquer qu'elle souhaite écouter les messages déclarés, la classe App doit le signaler à la messagerie.

C'est ce qui est fait dans ce morceau de code :

```
private void Application_Startup(object sender, StartupEventArgs e)
{
   CompositionInitializer.SatisfyImports(this);
   EventAggregator.SubscribeOnDispatcher(this);
}
```

Comme la classe utilise des importations MEF (et comme la classe elle-même n'est pas exportée) elle peut jouer le rôle central d'initialiseur des mécanismes MEF. Cela ne peut pas être fait dans le constructeur (car les importations ne sont pas encore satisfaites), c'est donc en toute logique dans Application_Startup qu'à lieu l'appel à l'initialisation de MEF (première ligne de la méthode ci-dessus).

La demande d'écoute des messages déclarés (ici uniquement les exceptions non gérées) s'effectue en appelant la méthode SubscribeOnDispatcher(this) de l'EventAggreator (la messagerie). Bien entendu, pour appeler cette méthode encore faut-il disposer d'une instance de la messagerie, d'où la présence de l'importation un peu plus haut et l'appel à SatisfyImports à la ligne précédente dans la méthode.

Arrivé à ce point Jounce a remplacé les mécanismes usuels de App.xaml par les siens et l'objet App se met à l'écoute des messages de type exception non gérée.

On retrouve le comportement presque « classique » de App.xaml.

La partie qui gère l'erreur, la méthode HandleEvent (venant du contrat IEventSink) est en revanche implémentée *a minima*. On y trouve un simple appel à MessageBox pour afficher le message d'erreur.

Il s'agit vraiment ici du strict minimum puisque nous sommes bien dans le code original non modifié du template Jounce.

Il faut préciser que dans la réalité la gestion des exceptions non gérées ne sera peut-être pas dans l'objet App mais plutôt dans le ViewModel de la Vue principale et qu'en place et lieu d'un MessageBox d'autres mécanismes plus subtils seront certainement utilisés.

> *Le template n'est pas à confondre avec un exemple d'implémentation. Il ne fait que mettre en place un décor minimum là où un exemple à but pédagogique montrerait une bonne façon de faire telle ou telle chose. Le template simplifie la création d'un projet, il n'est pas en lui-même une guideline absolue. Il faut toujours avoir à l'esprit cette nuance subtile mais essentielle entre template de projet et bonnes pratiques…*

La MainPage. Connexion au ViewModel et Blendabilité.

Par défaut, un template Jounce propose une MainPage qui sera le *shell* de l'application. Une MainPage cela signifie plusieurs fichiers qui chacun porte la trace de MEF et de Jounce :

- MainPage.xaml, l'interface visuelle
- MainPage.xaml.cs, le code-behind de l'interface

Mais aussi

- MainViewModel.cs, le ViewModel de la page principale

Commençons par l'interface.

MainPage.xaml

S'agissant d'un simple template cela ressemble beaucoup à ce qu'une application Silverlight propose par défaut : un UserControl jouant le rôle de page principale exposant une Grid appelée LayoutRoot.

Jusque-là, rien de bien extraordinaire.

Si on regarde de plus près, sous Visual Studio nous voyons ceci (figure 7) :

Figure 76 - MainPage par défaut

Ce qu'il faut noter est que la grille contient déjà un texte « This is some design-time text. » (« *ceci est un texte de conception* »). Il y a donc un TextBlock par défaut dont le seul but est de nous rappeler que Jounce prend en charge les données de conception, ce qui est l'un des éléments de la blendabilité. Pour voir ce texte il faut construire le projet au moins une fois.

Comment s'opère cette blendabilité ?

Le support des données de conception

Les données de conception sont essentielles, aussi bien sous Expression Blend que sous Visual Studio : elles permettent une mise en page plus précise et plus rapide tenant compte de la nature réelle des données à afficher (textes, images…).

Lorsque l'application est sous éditeur elle ne tourne pas. C'est une lapalissade. De fait afficher des données de conception n'est pas forcément un problème simple.

Il existe plusieurs façons de résoudre ce problème.

Expression Blend propose la création de données de test (avec un mini générateur de données aléatoires, en partant d'une classe, etc). Mais peu de gens, par la faute d'une

stratégie commerciale indécodable de Microsoft, ne connaissent ni n'utilisent ce logiciel pourtant merveilleux. Laissons ainsi de côté les solutions purement Blend.

MVVM Light se sort d'affaire en créant un service Locator qui expose les ViewModels sous la forme de propriétés statiques. Les Vues faisant pointer leur DataContext vers l'une des propriétés du ViewModel Locator. Chaque ViewModel peut, dans son constructeur généralement, détecter s'il est en mode Design ou non et fournir de fausses données ou bien accéder « pour de vrai » aux sources de données de l'application.

Ce principe marche plutôt bien sauf qu'il n'a pas été conçu pour être utilisé avec MEF qui s'occupe lui-même de la connexion entre les modules sans avoir à passer par un service Locator, a fortiori en utilisant des propriétés statiques interdisant l'utilisation du mécanisme d'importation / exportation propre à MEF.

Jounce apporte une fois encore ici une réponse pleine d'astuce et sachant exploiter les possibilités « modernes » de Silverlight. Notamment le support de données de conception…

Si vous vous reportez à la figure 6 (montrant l'arbre du projet) vous verrez un répertoire SampleData. Il ne contient pas vraiment des données comme on pourrait s'y attendre (par exemple un fichier XML) mais un fichier nommé DesignMainViewModel.cs.

Un second ViewModel pour la MainPage ? Oui.

Mais expliquons le mécanisme qui est plus subtile encore.

Regardez encore la figure 6. Elle montre un autre répertoire du projet portant le nom de Common. Ce répertoire contient le fichier IMainViewModel.cs. Une interface.

Cela fait beaucoup de petites choses, un dessin permettra d'y voir plus clair (en tout cas je l'espère !). Le schéma suivant (figure 8) montre l'ensemble des participants. Si la rigueur UML n'est pas forcément respectée, j'ai tenté de rendre les choses lisibles par tous.

En regardant le schéma, ne vous affolez-pas ! Ce n'est pas si compliqué que cela, et je vais détailler le mécanisme.

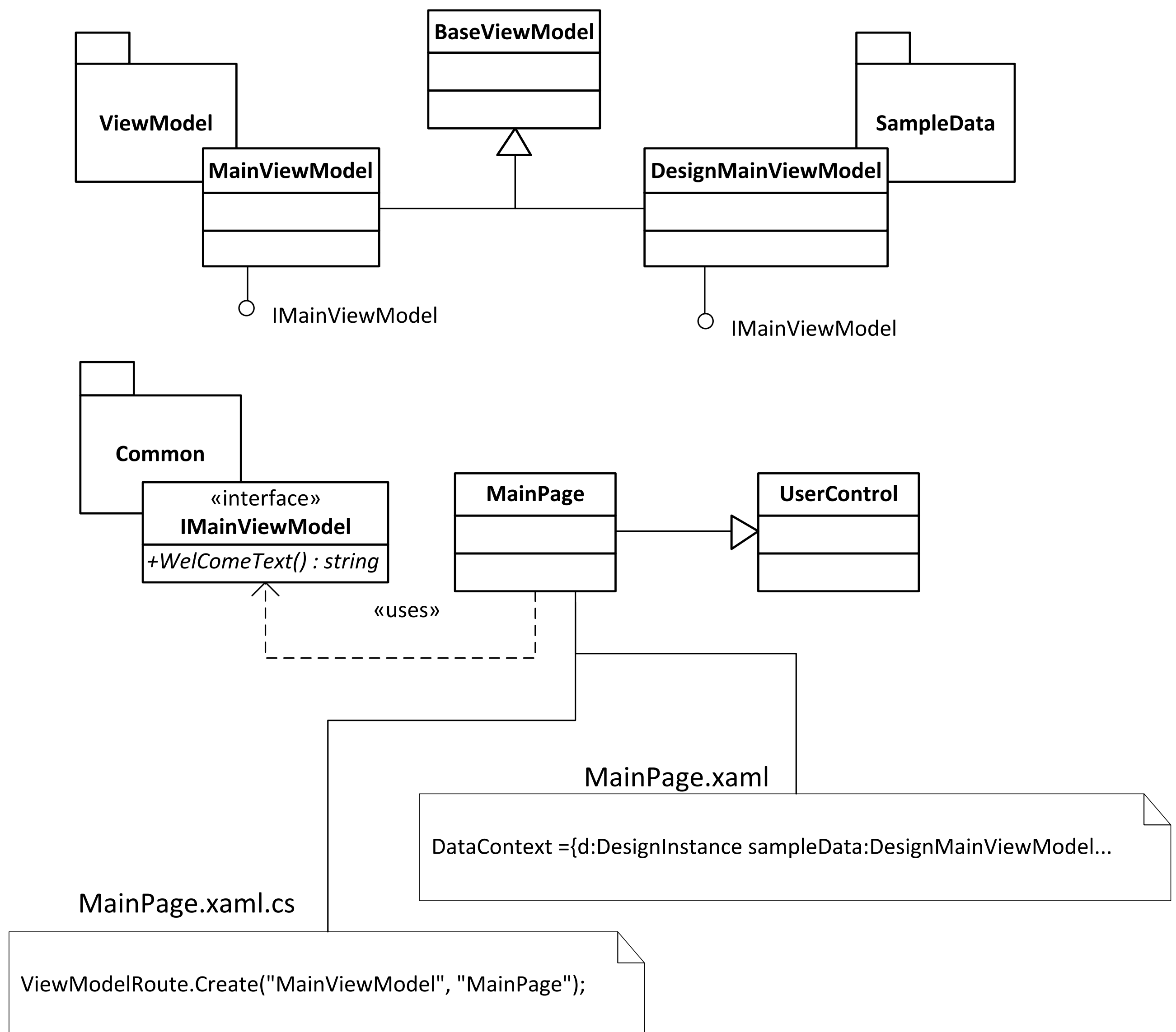


Figure 77 - Le mécanisme des données de conception

Tout d'abord les ViewModels exposent une interface faite généralement de simples propriétés (les données pour la Vue ainsi que les commandes). Un ViewModel peut décider aussi d'exposer des méthodes, c'est une question d'architecture et de choix purement contextuels.

L'avantage de définir les ViewModels sous la forme d'interfaces est que, bien entendu, cela créée un découplage assez fort entre le contrat et l'implémentation qui le supporte. De fait, tout module de l'application qui veut accéder à un ViewModel le fera via son interface ce qui permet à tout moment de décider de remplacer une implémentation par une autre sans chambouler tout le logiciel.

Ainsi, l'interface IMainViewModel est définie dans le répertoire Common (qui peut être en plus un espace de noms, ce qui est le cas dans le template Jounce). Cette interface est celle du ViewModel de la MainPage. Dans le template Jounce l'interface n'expose qu'une seule propriété « WelcomeText », une chaîne de caractères.

Regardons en haut du schéma maintenant. Nous voyons une classe nommée BaseViewModel qui donne naissance à deux classes filles : MainViewModel, dans le répertoire ViewModel et DesignMainViewModel dans le répertoire SampleData.

Ces deux classes implémentent l'interface IMainViewModel.

La première (celle du répertoire ViewModel) est le « vrai » ViewModel de la MainPage. Elle contient le code réel nécessaire au fonctionnement de la Vue ainsi que les propriétés et commandes exposées. Tout ce qui est exposé (*public*) est défini dans l'interface.

La seconde classe (celle du répertoire SampleData) est le « faux » ViewModel, celui ne servant qu'au design pour fournir des données simplifiant la mise en page de la Vue. Cette classe a tout d'un ViewModel « normal » : elle hérite de ViewModelBase et supporte l'interface qui lui est propre (IMainViewModel). Seulement elle ne possède aucun code (en tout cas cela n'est généralement pas nécessaire). Si elle en possède il s'agira uniquement de code générant des données aléatoires par exemple. Dans le cas le plus simple elle reprend les propriétés de l'interface et retourne directement des valeurs. Fabriquer ce « faux » ViewModel est très simple et rapide une fois l'interface définie. Visual Studio aidé par exemple de Resharper permet facilement d'implémenter tous les membres d'une interface en un clic.

ViewModelBase est une classe de Jounce qui offre à un ViewModel certains services sur lesquels nous reviendrons (comme la validation des données ou la gestion de IPropertyChanged).

Donc faisons le point : nous avons défini une interface pour notre ViewModel. Nous avons bien entendu créé un vrai ViewModel implémentant cette interface. Nous avons ensuite créé un second ViewModel d'aspect identique mais dont le code se borne à retourner des données de design pour les propriétés publiques de l'interface.

Voyons maintenant la Vue, MainPage.

MainPage est un simple UserControl ici (cela pourrait être une Page dans une application utilisant la navigation). Une Vue Silverlight se compose en réalité de deux fichiers ayant chacun leur importance : un fichier XAML qui définit le visuel et un fichier C#, le code-behind.

L’astuce commence à prendre forme…

En mode conception, que cela soit Blend ou Visual Studio, seul le code Xaml est interprété, le code-behind est ignoré. A l’exécution ce dernier sera en revanche exécuté.

Donc, dans MainPage.Xaml, nous allons exploiter la capacité de Xaml à définir des données de conception ignorées au runtime. Pour cela nous faisons pointer le DataContext (du LayoutRoot) vers une DesignInstance de SampleData.DesignMainViewModel. En conception, et à condition que le projet ait été construit, VS ou Blend seront capables d’instancier DesignMainViewModel et de l’affecter au DataContext de la grille LayoutRoot. Nous verrons le message défini par WelcomeText.

Mais dans le code-behind, nous ajoutons le code suivant :

```
[Export]
public ViewModelRoute Binding
{
   get { return ViewModelRoute.Create("MainViewModel", "MainPage"); }
}
```

Je reviendrais plus loin sur les détails de ce code, mais disons simplement qu’il définit dynamiquement au runtime une « route » entre une Vue et son ViewModel. La route créée ici relie la Vue exportée sous le nom de contrat « MainPage » au ViewModel exporté sous le nom de contrat « MainViewModel ».

A l’exécution, les données de conception seront totalement ignorées, le DataContext de la LayoutRoot ne sera pas initialisé. En revanche le code-behind sera exécuté, et il exporte une route qui connecte la Vue au ViewModel « réel ».

Nous verrons alors à l’exécution un autre texte, celui qui est retourné par la propriété WelcomeText se trouvant dans MainViewModel et non pas celui de DesignMainViewModel …

Et le tour est joué !

Cela vous paraît un peu compliqué ?

Finalement c’est un montage logique et simple. A la seconde lecture on comprend souvent mieux…

C’est aussi ici qu’on comprend mieux pourquoi je dis que Jounce est plus « moderne » que MVVM Light. Ce dernier est plus direct, plus simple à comprendre, là où Jounce joue sur une connaissance plus vaste et plus subtile de nombreux éléments (xaml, MEF, …).

Intellectuellement, Jounce est plus élaboré c'est une évidence. Jeremy a fait un condensé de Prism et Caliburn, deux frameworks considérés comme très bien pensés mais assez complexes. MVVM Light est parti d'une idée simple, implémenter une solution « light » pour assurer les exigences de base de MVVM. Ces deux voies n'aboutissent bien entendu pas à la même chose.

Le code de MainPage

Il y a bien sûr un peu de code dans le template Jounce. Très peu, juste le minimum, mais il utilise tout ce qu'il faut savoir de base pour commencer avec le toolkit. Autant regarder de plus près.

MainPage.Xaml

C'est le code minimum pour afficher une grille avec un TextBlock bindé à la propriété WelcomText. Le DataContext de la grille est bindé aux données de conception.

```
<UserControl x:Class="BasicJounceApp.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:sampleData="clr-namespace:BasicJounceApp.SampleData"
   mc:Ignorable="d"
   d:DesignHeight="300" d:DesignWidth="400">

   <Grid x:Name="LayoutRoot" Background="White"
        d:DataContext="{d:DesignInstance sampleData:DesignMainViewModel,
                                IsDesignTimeCreatable=True}">
      <TextBlock Margin="10" FontSize="20" Text="{Binding WelcomeText}" />
   </Grid>
</UserControl>
```

MainPage.Xaml.cs

Le code minimum pour une Vue : L'utilisation de l'attribut Jounce/MEF ExportAsView auquel sont passés différents paramètres : Le nom du « contrat » MEF sous lequel la Vue est exportée, et l'indication IsShell = true qui signifie que cette Vue est le shell de l'application. Une seule Vue peut posséder cette indication, ce qui est évident.

La Vue elle-même dérive de UserControl et ne possède pas de code particulier en dehors de l'exportation de la « route », c'est-à-dire du couple ViewModel / Vue qui permet à Jounce d'instancier le bon « partenaire » (qu'on parte de la Vue ou du ViewModel).

L'exportation se base sur le type (ViewModelRoute), le nom de la propriété n'est pas important, le *getter* retourne un objet créé par la méthode statique Create du type ViewModelRoute. Nous verrons d'autres utilisations des méthodes de routage plus loin.

A noter qu'il existe deux écoles : celle qui préfère ajouter dans chaque Vue l'exportation de « sa » route, et celle qui préfère créer une classe à part contenant les exportations de toutes les routes de l'application. A vous de choisir selon le contexte comme toujours.

Enfin, et comme il est d'usage avec MEF de façon générale, on évitera les « *magic strings* » (les chaînes de caractères magiques) dans une application réelle pour définir les contrats d'exportation. Ainsi l'attribut ExportAsView, tout comme le code de création de la route, et de façon générale tout ce qui fait référence à un contrat d'importation ou d'exportation doit, de préférence, utiliser des constantes regroupées quelque part dans l'application et jamais des bouts de texte libres sujets à coquille, erreur de copie, faute d'orthographe, etc.

```
using Jounce.Core.View;
using Jounce.Core.ViewModel;

namespace BasicJounceApp
{
   [ExportAsView("MainPage", IsShell = true)]
   public partial class MainPage : UserControl
   {
      public MainPage()
      {
         InitializeComponent();
      }

      [Export]
      public ViewModelRoute Binding
      {
         get { return ViewModelRoute.Create("MainViewModel", "MainPage"); }
      }
   }
}
```

L'interface du ViewModel

S'agissant du template, cette interface se limite à une propriété retournant un texte de type « *hello world !* ».

```
namespace BasicJounceApp.Common
{
   public interface IMainViewModel
   {
```

```
        string WelcomeText { get; }
    }
}
```

C'est d'abord ici qu'une application réelle définira toutes les propriétés, commandes et méthodes que le ViewModel exposera.

Le ViewModel « réel »

Comme je l'ai expliqué, la stratégie pour obtenir des données de conception passe par la création de deux ViewModels supportant chacun la même interface. Dans le template non modifié de Jounce la nuance est très subtile entre ces deux ViewModels puisqu'elle se situe uniquement dans le texte retourné par l'unique propriété exposée.

Dans la réalité le ViewModel « réel » se chargera au cours de l'implémentation de tout le code nécessaire au fonctionnement de la Vue.

```
using Jounce.Core.ViewModel;
using BasicJounceApp.Common;

namespace BasicJounceApp.ViewModel
{
    [ExportAsViewModel("MainViewModel")]
    public class MainViewModel : BaseViewModel, IMainViewModel
    {
        public string WelcomeText
        {
            get { return "Welcome to Jounce."; }
        }
    }
}
```

Bien qu'il soit très simple, ce code permet d'observer les points importants de la conception d'un ViewModel sous Jounce :

Tout d'abord, le ViewModel hérite de BaseViewModel, une classe fournissant des services essentiels au ViewModel que nous détaillerons plus loin. Ensuite le ViewModel implémente l'interface IMainViewModel, celle-là même qui permet de découpler l'implémentation du ViewModel du contrat qu'il représente. Cette interface est spécifique au ViewModel en cours, chaque ViewModel de l'application exposant une interface différente (en général en tout cas).

Bien entendu, on note l'exportation de la classe qui utilise l'attribut ExportAsViewModel auquel est passé le nom de contrat MEF sous lequel la classe sera connue. Comme pour la

Vue il est fortement conseillé d'utiliser une constante définie proprement dans l'application qu'une chaîne de caractères.

Le ViewModel de design

Voici le « faux » ViewModel, celui qui est chargé de fournir les données de conception. Il ressemble en tout point à un « vrai » ViewModel, sauf qu'en général il ne fait que retourner des données (fixes ou aléatoires). Si l'interface du ViewModel contient des commandes, celles-ci peuvent être retournées à null (puisque de toute façon les commandes ne seront pas invoquées en conception). Dans le même esprit, ce ViewModel de design ne contient jamais de code fonctionnel. Le seul code qu'il peut contenir est du code de génération de données aléatoires (simuler une liste de clients, d'articles, …).

```
using BasicJounceApp.Common;

namespace BasicJounceApp.SampleData
{
   public class DesignMainViewModel : IMainViewModel
   {
      public string WelcomeText
      {
         get { return "This is some design-time text."; }
      }
   }
}
```

On notera toutefois quelques particularités intéressantes :

- La classe n'hérite pas de BasViewModel (c'est inutile) ;
- Elle implémente directement l'interface du ViewModel considéré ;
- Elle n'est par exportée ;
- Elle n'importe aucune donnée non plus.

Si le premier et le second point sont faciles à comprendre, les deux derniers sont des impératifs absolus. La classe ne doit pas être exportée, ce n'est pas une option, cela n'aurait aucun sens et brouillerait même le fonctionnement de l'application. De même, elle n'importe aucune donnée puisque cela ne fonctionnerait qu'au runtime et n'aurait pas plus de sens.

Point intermédiaire

Nous venons de décortiquer le squelette d'une application supportant Jounce. Il s'agit d'un petit squelette, d'une application minuscule, le template Jounce…

Toutefois cela nous a permis de voir de nombreuses spécificités de Jounce et surtout de mettre en évidence les mécanismes essentiels de ce toolkit.

Beaucoup de choses restent toutefois à découvrir et à préciser. Mais avant d'aller plus loin précisions certains points abordés jusqu'ici.

L'essentiel sur …

Cette section revient sur des classes, des procédés ou particularités de Jounce qui ont été abordées ou évoquées lors de l'étude du template de projet Jounce mais sur lesquels il y a encore beaucoup à savoir. Evidemment il ne s'agit pas ici non plus de reprendre ce que vous trouverez dans la documentation en ligne de Jounce, mais plutôt d'expliquer ce qui parfois n'est pas facile à comprendre du premier coup (surtout que tout ce qui tourne autour de Jounce est en anglais).

L'Application Service

Dans le premier exemple présenté plus haut, j'ai parlé de *l'Application Service* (Voir page 330). Cette façon d'étendre les capacités de l'objet Application sans avoir besoin de créer une classe fille a été introduite dans Silverlight 3 et n'a pas fait la une des blogs spécialisés. Peu de gens certainement ont compris la nécessité ou l'intérêt de cette extension.

Pourtant elle permet d'ajouter à l'application des comportements ou des services qui restent accrochés au cycle de vie de cette dernière. Cela peut avoir un grand intérêt dans de nombreux cas.

Jounce s'en sert pour insérer son propre fournisseur de IApplicationService. C'est une utilisation astucieuse et une belle démonstration de ce que peut être l'extensibilité de l'objet Application. Une fois encore, Jounce utilise habilement des capacités récentes mais éprouvées de Silverlight.

Le service créé par Jounce effectue de nombreuses choses que l'exemple présenté ne permet pas de voir. D'ailleurs, pour beaucoup, les fonctions assurées par ce service sont justement placées là pour être invisibles tout en apportant certains petits « plus » propres à Jounce. La liste des fonctions assurées par le service Jounce donne un aperçu de la richesse sous-jacente :

- Création et connexion d'un fournisseur de service IApplicationService pour gérer le cycle de vie de l'application ;
- Création d'un AggregateCatalog MEF par défaut (voir mon article sur MEF) ;
- Création d'un CompositionContainer MEF ;
- Ajout d'une instance de MefDebugger pour l'aide au débogue ;
- Création d'un objet Logger et affectation d'un niveau de *logging* par défaut ;
- Création d'un ViewModelRouter ;
- Création d'un Service de déploiement ;

- Maintien d'une collection de toutes les Vues
- Création d'un EventAggregator (messagerie) pour l'abonnement aux exceptions non gérées ;
- Mise à disposition de nombreux hooks dépendants du cycle de vie de l'application comme Exited, Exiting, Started et Starting.

Jounce étant basé sur MEF et voulant en simplifier l'utilisation, il se charge de nombreuses tâches propres à ce framework. La création d'un AggregateCatalog permet par exemple à Jounce de gérer le chargement dynamique des XAP, tout comme le CompositionContainer. Le développeur n'ayant pas à se préoccuper de comment tout cela fonctionne.

Le service de débogue de Jounce permet d'obtenir les traces des messages et de toute la « plomberie » interne de Jounce. C'est un point fort de ce toolkit même si cela n'est pas lié directement aux fonctions qu'il propose. Mais dans un tel système où beaucoup d'évènements s'enchaînent un peu « magiquement », disposer d'un système de trace efficace qui plonge dans le toolkit est une aide véritable en cas de bogue un peu récalcitrant. Jounce n'oublie pas qu'une application ne marche pas toujours comme prévu, et c'est un point fort de la librairie.

Jounce créée aussi un service de Log, ce qui est essentiel dans une application sérieuse. Bien entendu Silverlight n'est pas WPF et de base il n'est pas possible d'écrire des fichiers de Log sur disque (en tout cas facilement et en dehors de l'Isolated Storage dont la capacité est très limitée sans intervention de l'utilisateur). Jounce ne peut pas passer outre les limites ultra sécuritaires de Silverlight. En revanche, le Logger par défaut fonctionne en mode Debug sous Visual Studio ce qui est déjà énorme. Et le mécanisme est en place. Jounce vous permet de créer une classe de Log personnalisée qu'il branchera automatiquement à la place de la sienne. Il est ainsi possible de créer un Logger qui transmet les messages à un serveur par exemple, ou qui garde une trace minimum dans l'Isolated Storage, ou n'importe quoi d'autre. Jouant sur la flexibilité de MEF, le service de Log de Jounce fait partie de toutes les bonnes idées contenues dans ce toolkit. Tout comme les traces de débogue, la gestion d'un système de Log n'est pas fantastique en soi, mais est indispensable pour concevoir des applications un peu sérieuses. Jounce ne se borne pas à quelques fonctions phares de façade, cela dénote un certain état d'esprit de son concepteur et donne confiance dans le reste du toolkit.

Tous les autres services Jounce accrochés à l'objet Application jouent un rôle important. Plutôt qu'en parler des heures, je vous propose d'avancer vers d'autres exemples qui

mettront mieux en valeur les fonctionnalités du toolkit (après cette section sur les « essentiels »).

Dans la pratique, le service applicatif de Jounce se créée et s'instancie directement en Xaml dans le fichier App.xaml :

```
<Application …
      xmlns:Services="namespace:Jounce.Framework.Services" …>

     <Application.ApplicationLifetimeObjects>
        <Services:ApplicationService>
     </Application.ApplicationLifetimeObjects>

</Application>
```

C'est une approche « zéro code » totalement déclarative. On notera que Silverlight sait gérer plusieurs services et que le fait que Jounce insère le sien dans ApplicationLifetimeObjects n'empêche en rien que vous ajoutiez vos propres services à la liste.

Silverlight garantit que les services sont initialisés dans l'ordre de leur déclaration et qu'ils sont disposés dans l'ordre inverse ce qui permet d'envisager des relations de dépendance entre services sur une base prévisible.

Marquage d'une Vue

Une vue est marquée en utilisant l'attribut ExportAsView sur sa classe (dans le code-behind). Le tag doit être unique.

Il est préconisé d'utiliser des constantes plutôt que des chaînes de caractères pour définir le nom de contrat MEF lors de l'exportation.

ExportAsView supporte aussi l'exportation par type, on utilise alors typeof() au lieu d'un nom de contrat. Il s'agit là d'un comportement MEF.

Une seule Vue peut être marquée IsShell=true, je l'ai dit, mais il faut insister aussi sur le fait qu'il faut absolument qu'une Vue soit ainsi marquée, sinon Jounce déclenchera une exception (une application doit forcément posséder une Vue principale).

Utilisant les capacités de MEF qui autorisent la déclaration de métadonnées sur une exportation, Jounce définit Category, MenuName et ToolTip. Renseigner ces métadonnées est optionnel mais se révèle utile dans certaines situations (création automatique de menus, de pages ; requête sur les Vues existantes, affichage d'aide à l'utilisateur…).

Marquage d'un ViewModel

Un ViewModel est marqué en utilisant l'attribut ExportAsViewModel sur sa classe. Comme pour le marquage d'une Vue Jounce propose ici des attributs d'exportation MEF personnalisés qui simplifient à la fois le respect de MVVM et l'utilisation de MEF comme support. Pour apprendre plus en détail ce que sont ces attributs personnalisés de MEF, je ne peux que renvoyer le lecteur (one more time !) à mon article précédent traitant de MEF.

Tout comme les Vues, le tag utilisé pour un ViewModel doit être unique (le nom de contrat MEF).

Ici aussi on utilisera de préférence une constante plutôt qu'une simple chaîne de caractères.

Comme pour la Vue, on peut utiliser typeof() pour marquer le ViewModel en utilisant sa classe comme tag plutôt qu'un nom de contrat.

Un ViewModel hérite généralement de la classe BaseViewModel que nous allons étudier dans quelques lignes (mais cela n'est pas une obligation).

Un ViewModel peut être lié à plusieurs Vues, Jounce le supporte. J'entends par là que plusieurs Vues différentes peuvent utiliser tour à tour le même ViewModel, ou bien que plusieurs Vues peuvent utiliser le même ViewModel simultanément.

L'intérêt pratique peut s'avérer énorme puisqu'il suffit d'écrire un seul ViewModel qui pourra alimenter des Vues très différentes. Par exemple dans certains cas l'application (ou l'utilisateur) peut préférer une Vue flottante résumant certaines informations importantes, dans d'autre cas une Vue très détaillées sur les mêmes informations ou même une simple fenêtre dockée dans une région à l'intérieur d'une autre Vue. Toutes ces Vues exploitant les mêmes données et possédant les mêmes comportements (certaines variantes peuvent choisir de ne pas exposer toutes les données ni toutes les commandes) un seul ViewModel est nécessaire.

Bien utilisée, cette possibilité est très riche et laisse une grande part à la créativité du développeur et à celle du designer…

BaseViewModel, BaseNotify et IViewModel

La classe BaseViewModel est fournie par Jounce pour aider à l'écriture des ViewModels.

Le ViewModel le plus simple peut être créé en héritant juste de BaseNotify (pour la notification des changements de valeur des propriétés) et en implémentant IViewModel.

Toutefois, dans la pratique, on préfèrera hériter de BaseViewModel, plus complet, même si tous les ViewModels ne se servent pas forcément de toutes les possibilités offertes par cette classe mère.

Un ViewModel bénéficie alors d'un ensemble de services :

- L'EventAggregator (la messagerie)
- Le Logger (service de Log)
- Le ViewModel Router (dialogue rapide entre ViewModels) (propriété Router)

Mais BaseViewModel offre aussi des choses plus subtiles comme :

- La gestion du Visual State Manager
- La détection du mode Design
- Le cycle de vie de la Vue.

Concernant le VSM, il s'agit ici d'une possibilité vraiment très intéressante car elle permet de gérer notamment des transitions entre les Vues ou de modifier l'état visuel de ces dernières depuis le ViewModel, sans pour autant violer la loi de séparation qu'impose MVVM (le ViewModel ne doit pas connaître sa ou ses Vues) et sans utiliser un complexe ballet de messages asynchrones.

L'astuce de Jounce consiste, quand une Vue est bindée à son ViewModel, à placer un *delegate* sur le ViewModel pour gérer les transitions d'états via le VSM.

Ainsi rassuré par le toolkit que MVVM et sa séparation absolue entre ViewModel et ses Vues est totalement respectée, on peut le cœur léger écrire depuis le ViewModel du code comme celui-ci :

```
GoToVisualState("EtatVisuelXXX", true);
```

Ce qui permet de faire passer la Vue accrochée au ViewModel à l'état « EtatVisuelXXX » en utilisant les transitions du VSM (second paramètre à true). La communication entre ViewModel et Vue étant basée sur un *delegate* qui se charge aussi d'assurer que l'action est bien exécutée sur le thread de l'UI, on est assuré de la meilleure réactivité possible.

> *On pourrait philosopher des heures en se demandant si en agissant de la sorte on ne viole finalement pas MVVM. Techniquement il n'y a pas violation puisqu'il n'existe pas de couplage fort entre le ViewModel et sa Vue pour réaliser l'opération. Mais fonctionnellement il y a bien un ordre particulier envoyé à la Vue par le ViewModel, ordre compris par les deux parties, donc basé sur une connaissance mutuelle, même partielle et transitive (au travers du nom de l'état transmis)… Les puristes pourraient donc crier au scandale, mais les pragmatiques salueront au contraire cette*

merveilleuse fonction de Jounce qui évite, là encore, de faire se balader des messages asynchrones entre le ViewModel et la Vue avec tout ce que cela suppose de contraintes et de complexification inutile du code (expérience faite).

Jounce va même plus loin dans le viol apparent de MVVM puisque si plusieurs Vues sont bindées en même temps au même ViewModel (ce que Jounce autorise parfaitement et sans souci, le pattern MVVM aussi d'ailleurs) vous pourrez écrire un code comme le suivant permettant de piloter une Vue bien précise :

```
GoToVisualStateForView(NomDeLaVue, NomDeLEtat, UtiliserLesTransitions);
```

Poussant les limites au-delà même du côté obscur, BaseViewModel propose une propriété RegisteredViews qui permet au ViewModel de connaître la liste exacte de toutes les Vues qui lui sont connectées. S'en servir exposerait-il le développeur aux foudres des puristes ? Non, qu'il se rassure, il s'agit d'une simple liste de noms, des strings… Aucune connaissance des Vues par le ViewModel là encore, malgré les apparences ! Mais cette liste de noms permet, par exemple, de piloter des états visuels via GotoVisualStateForView.

Finalement Jounce est bien sage, et qu'apparence sont ces violations de MVVM…

En revanche il s'agit là de fonctionnalités particulièrement séduisantes car elles permettent un contrôle plus grand des Vues par les ViewModels ce qui s'avère souvent nécessaire, le tout en respectant MVVM et en se passant de la lourdeur qu'impose la messagerie.

BaseViewModel offre aussi la détection du mode Design en exposant la propriété InDesigner qui peut être testée à tout moment.

Si on applique correctement le principe des deux ViewModels exposé plus haut dans ce Livre Blanc, les « vrais » ViewModels ne devraient pas avoir à se servir de cette possibilité puisqu'ils ne sont jamais instanciés en mode design.

Néanmoins Jounce est assez permissif, et certains développeurs peuvent préférer n'utiliser qu'un seul ViewModel faisant la distinction lui-même entre design time et runtime. C'est une option tout à fait acceptable dans certains cas. D'autant plus qu'il reste possible de déclarer un binding sur le ViewModel en Xaml pour bénéficier des mêmes services (données de design et données réelles).

```
<Grid d:DataContext="{d:DesignInstance
vm:MyViewModel,IsDesignTimeCreateable=True}">
```

Comme le montre la déclaration ci-dessus, la méthode est rigoureusement identique qu'on utilise un seul ViewModel gérant les deux facettes (design / runtime) ou deux ViewModels distincts.

La méthode faisant intervenir deux ViewModels et une interface, méthode proposée d'emblée par le template de projet Jounce, peut s'avérer trop lourde dans certains cas. N'hésitez pas à utiliser un seul ViewModel pour les deux fonctions si cela vous fait gagner du temps. J'ai un faible pour la version à deux ViewModels plus interface, cela sépare réellement mieux le code de design du code de runtime, ce qui me semble largement préférable, mais je sais aussi que tout ce qui augmente le temps d'écriture du code n'est pas forcément compatible avec les impératifs du projet…

Enfin, BaseViewModel permet au ViewModel d'être averti du cycle de vie des Vues et de sa propre mise en service.

Les méthodes virtuelles InitializeVm(), ActivateView() et DeactivateView() peuvent être *overridées* pour savoir, respectivement :

- Quand le ViewModel est activé pour la première fois (ce qui permet de faire certaines initialisations)
- Quand une navigation vers la Vue est effectuée (si plusieurs Vues sont accrochées au même ViewModel son nom est disponible dans les arguments)
- Quand la Vue associée est quittée pour une autre.

Tous ces évènements font l'objet de l'émission d'un Log de niveau « information » comme la majorité des actions réalisées par Jounce (et qu'on retrouve grâce au Logger).

On notera que ces méthodes, comme celle concernant le Visual State Manager sont issues de l'implémentation de IViewModel par BaseViewModel.

Terminons le tour d'horizon de BaseViewModel en parlant de sa classe mère, BaseNotify (utilisable directement nous l'avons vu plus haut) qui propose le strict minimum pour un ViewModel, c'est-à-dire la notification des modifications de valeur des propriétés (INotifyPropertyChanged). En fait BaseNotify peut être utilisée pour créer toute classe qui nécessite le support de INotifyPropertyChanged (puisqu'elle ne fait que ça). Les Modèles (au sens MVVM) sont un bon exemple de classes pouvant utiliser BaseNotify.

La particularité qu'introduit Jounce ici est la possibilité de signifier le changement d'une propriété sans avoir à utiliser une chaîne de caractères pour son nom.

L'obligation de passer le nom de la propriété sous forme d'une string est une très mauvaise idée de .NET, l'une des rares qui me hérisse. Elle introduit un risque d'erreur et de bogue particulièrement sournois à détecter. D'autant que de marginal il y a quelques années, le

support de INotifyPropertyChanged est devenu une obligation pour presque toutes les classes dans les architectures récentes.

Il existe bien la possibilité de déclarer des constantes pour se prémunir contre ce problème, mais c'est fastidieux. Encore du code à taper et à contrôler dont on pourrait parfaitement se passer.

Le toolkit MVVM Light opte pour une solution assez sage : en mode Debug les noms passés sont systématiquement testés (via la Réflexion) et une exception est levée si le nom n'existe pas. Au runtime ce test est automatiquement déconnecté. C'est une parade intéressante mais hélas il y a des trous dans la passoire : On peut avoir commis l'erreur de faire un copier/coller non modifié du nom d'une autre propriété qui existe, aucune erreur ne sera levée ; pire l'erreur peut réellement exister mais on peut ne jamais être passé dessus en mode Debug, ce n'est que chez l'utilisateur, plus tard, que le comportement étrange et souvent difficile à reproduire aura lieu. Cette méthode, pour astucieuse qu'elle soit n'est donc pas parfaite.

Pour sa part, Jounce offre plusieurs façons de signaler le changement d'une propriété. La plus simple, la plus classique consiste à appeler la méthode RaisePropertyChanged() en passant le nom de la propriété. C'est la méthode de base, avec tous les désavantage d'utiliser une string, mais avec la rapidité maximale. Une variante existe en utilisant comme argument une liste de strings, ce qui permet de signaler le changement de plusieurs propriétés en un seul appel.

Les autres méthodes sont beaucoup plus fiables. On n'est pas même obligé de passer le nom de la propriété ! Bien entendu tout à un prix, Jounce utilise la Réflexion et des objets StackTrace pour analyser d'où vient l'appel et déterminer lui-même le nom de la propriété.

Pour une boucle de jeu il est fort probable que le prix à payer soit trop fort et que les performances ne soient trop dégradées. Pour une application de gestion, cible de Jounce, je considère que le temps « perdu » n'a ici pas de prix car il permet d'avoir l'assurance d'éliminer une source de bogue courante et difficile à déboguer. Et puis il ne faut pas exagérer, la Réflexion ne monopolise pas non 100% du processeur !

Ainsi, dans le setter d'une propriété, on peut appeler directement RaisePropertyChanged() sans aucun argument. L'analyse de l'objet StackTrace permet à Jounce d'extraire le nom de la propriété. Un contrôle est effectué pour s'assurer que l'appel a bien lieu depuis un setter.

Une autre façon de procéder est d'appeler RaisePropertyChanged(()=>laPropriété) c'est-à-dire d'utiliser une expression Lambda ne faisant que retourner la propriété (elle-même, pas son nom sous forme de string). Ici, l'analyse de l'expression permet d'extraire le nom de la

propriété. Avantage : si le nom passé n'existe pas le compilateur le rejettera. Second avantage, on peut aussi déclencher l'évènement à tout endroit dans le code sans être obligé d'être dans le setter de la propriété. On retrouve l'un des inconvénients de la méthode utilisée par MVVM Light : le trou dans la passoire existe puisque si la propriété existe (opération de copier / coller non modifiée en général) le code passera sans problème.

Personnellement je conseille la version sans paramètre à utiliser uniquement dans le setter des propriétés. C'est absolument fiable et évite radicalement la possibilité d'un bogue à ce niveau. Les quelques millièmes de secondes éventuellement perdus ne sont rien comparés à la fiabilité gagnée. En second lieu, je conseille la version avec expression Lambda lorsqu'on doit signaler le changement d'une propriété hors d'un setter. La méthode n'est pas fiable à 100% mais réduit grandement les risques malgré tout.

La méthode « classique » est à réserver aux plus aventureux d'entre les lecteurs ou à ceux qui ont une phobie irrépressible des millisecondes perdues, ou qui, cas plus rare, doivent absolument optimiser un code à la milliseconde près.

ViewModelRouter

La classe BaseViewModel dont héritent généralement les ViewModels importe une instance de ViewModelRouter qu'elle expose sous le nom de propriété Router.

Le routeur de ViewModels est un procédé très efficace proposé par Jounce pour établir une communication entre ViewModels.

```
var viewModel = Router.ResolveViewModel("HomePageVM");
```

Le code ci-dessus (supposé à l'intérieur d'un ViewModel héritant de BaseViewModel) demande au routeur de résoudre le nom "HomePageVM" (sensé être le nom de contrat MEF du ViewModel de page Home de l'application, simple exemple bien entendu). Si le routeur trouve l'objet demandé il est retourné et devient utilisable directement.

L'utilisation directe d'un ViewModel par un autre est un progrès énorme par rapport à l'approche « tout message » de MVVM Light par exemple. En effet, si un ViewModel doit en appeler un autre pour le piloter ou simplement échanger des données, la seule façon de respecter MVVM est de transmettre des messages.

S'en suit un ballet asynchrone pas très simple à coordonner et qui va alourdir inutilement le code surtout si on utilise souvent le procédé dans l'application. Par exemple, pour lire une valeur puis en modifier une autre on écrira dans un cadre classique :

```
var isOk = viewModelA.UneFonction(x) ;
if (!isOk) viewModelB.UneValeur = 0;
```

Ce code hypothétique lit un résultat booléen depuis une méthode de l'instance de ViewModel A et selon la valeur lue positionne une propriété du ViewModel B.

C'est simple (on pourrait l'écrire en une seule ligne d'ailleurs), direct, on est presque dans un cours pour débutant en programmation.

Malheureusement, dès qu'on applique MVVM écrire un tel code est totalement interdit ! Il est hors de question que le code en cours puisse connaître les classes des ViewModels A et B et encore moins qu'il puisse agir directement sur des instances de ces dernières !

Ne reste plus qu'à utiliser une messagerie, asynchrone le plus souvent (donc avec appel non bloquant). Le pseudo code de la même action hyper simple devient alors quelques chose du type :

- Envoi du message « GetLeResultat » avec un argument X, en passant l'adresse d'une méthode de callback. Ou directement en programmant un autre message pour la réponse.
- Programmation préalable d'un gestionnaire pour le callback ou du récepteur du message de retour selon le choix effectué.
- Attente de la réponse (qui peut ne jamais venir, gestion d'un timeout ?), pendant ce temps puisque les messages sont asynchrones, le code courant est passé à autre chose…
- Réception de la réponse, rétablissement du contexte original au moment de l'envoi du message de départ.
- Envoi d'un message « SetUneValeur » avec le paramètre 0, toujours à la volée.
- Programmation d'un gestionnaire de réponse au message SetUneValeur dans ViewModelB

Et encore j'ai fait simple…

C'est complexe, ce qui était déterministe devient statistique (pas d'assurance d'avoir une réponse à un message lancé « à la volée », pas d'assurance de l'ordre des messages traités par les récepteurs, l'ordre d'envoi des réponses…). Le code devient purement in-maintenable dès que le procédé se généralise dans l'application.

La gestion des messages c'est bien, mais il faut la limiter à quelques cas bien précis. Les effets de bords sont difficilement maîtrisables, le code s'allonge, la maintenance devient complexe, ce n'est vraiment pas une « amélioration ». On est très loin du chant des sirènes

de MVVM quand on se frotte à la réalité d'une telle situation ! Mais MVVM n'y est pour rien. Il faut juste trouver le bon moyen d'atteindre le but. Et la seule messagerie n'est pas ce moyen, tout simplement.

Avec le routeur de Jounce on revient au code initial : simple, déterministe.

```
var isOk = (Router.ResolveViewModel("ViewModelA") as IViewModelA).UneFonction(x);
if (!IsOk) (Router.ResolveViewModel("ViewModelB") as IViewModelB).UneValeur = 0;
```

Reste une question : que faire de la réponse du routeur ? En effet ce dernier ne peut retourner à la base qu'une instance d'interface IViewModel (ou héritant de IViewModel) ce qui peut être suffisant parfois mais pas dans un cas comme celui de notre exemple…

C'est en partie pourquoi par défaut le template de projet de Jounce incite le développeur à créer une interface pour chaque ViewModel. Et cela explique pourquoi j'ai utilisé les interfaces hypothétiques IViewModelA et IViewModelB dans l'exemple ci-dessus.

En disposant de ces interfaces il est possible d'utiliser le routeur en recevant uniquement une instance typée par l'interface du ViewModel réclamé. Piloter ce dernier via son interface ne viole pas MVVM car il n'y a plus besoin d'avoir connaissance de la classe réelle qui implémente le ViewModel accédé, juste la connaissance de l'interface (dont la définition se trouve généralement dans une DLL à part, partageable par l'ensemble des modules).

Certes on en revient à l'écriture des interfaces pour les ViewModels, et je l'accorde, cela fait un peu plus de code à écrire. Mais il s'agit d'un code léger (une interface n'est qu'une liste), clair, à la finalité bien établie, maintenable et ouvrant la possibilité d'utiliser pleinement le routeur pour écrire un code simple, direct et synchrone. Rien à voir avec le code spaghetti qu'imposent les messages asynchrones.

Mais la classe ViewModelRouter ne s'arrête pas à cette simple utilisation. Elle propose en réalité toute une panoplie de méthodes aidant à retrouver des ViewModels mais aussi des Vues, à obtenir les métadonnées de ces objets, etc.

Par exemple, nous avons vu que l'appariement des Vues et des ViewModels s'effectue de façon déclarative sous Jounce en exportant (au sens MEF) une « route ».

Le ViewModelRouter collecte l'ensemble de ces routes, elles sont disponibles au travers de sa propriété Routes.

Il est tout aussi possible de déclarer, après coup, de nouvelles routes, par exemple créer de nouveaux assemblages Vue / ViewModel de façon dynamique selon le contexte. Là encore

c'est par le ViewModelRouter qu'on passera en appelant la méthode RouteViewModelForView().

La classe gère ainsi deux listes de façon transparente : les routes importées automatiquement par MEF et les routes supplémentaires ajoutées par le code. Toutes les opérations travaillant sur les routes le font en fusionnant les deux listes.

On peut considérer le ViewModelRouter comme une plaque tournante centralisant l'information sur les Vues et les ViewModels. On peut grâce à cette classe connaître la liste de toutes les Vues découvertes (propriété Views, vue comme une liste de UserControl). On peut de la même façon accéder aux ViewModels recensés (ViewModels, retournés sous la forme d'une liste de IViewModel).

On accède aussi à la ViewFactory ou à la ViewModelFactory (qui permettent de créer de nouvelles instances des Vues et des ViewModels en dehors des instances créées par MEF).

Il est possible de requêter les Vues ou les ViewModels (par exemple ViewQuery(name) ou HasView(name)) et bien d'autres choses comme obtenir une Vue non partagée ou un ViewModel non partagé.

Les Vues ou ViewModels « non partagés » (*non shared*) sont des objets créés de façon dynamique.

En effet, lorsqu'une Vue ou un ViewModel sont exportés, ils sont recensés par MEF. Jounce utilise des importations de type Lazy<T> de telle façon à ce que les objets ne soient instanciés que lorsqu'ils sont réellement utilisés. Mais au final il n'y a qu'une instance par type exporté.

C'est un peu ce que fait le ViewModel Locator de MVVM Light avec ces propriétés statiques : l'instance est créée lors de la première utilisation et reste active ensuite (sauf si on demande un « *clear* » explicite).

Mais il y a des cas où il est nécessaire d'obtenir plusieurs Vues de même type en même temps. Par exemple l'affichage du détail des factures d'un client : chaque facture peut être une Vue avec son propre ViewModel. Si on s'en tient au comportement de base, il n'y a qu'une seule instance de la Vue « Facture » et de son ViewModel associé.

Ici il devient nécessaire de passer une factory pour être capable de créer plusieurs instances différentes, chacune affichant une facture différente dans notre exemple.

Le ViewModelRouter propose un ensemble de propriétés et de méthodes permettant de créer des Vues ou des ViewModels « non partagés », avec ou sans couplage automatique.

C'est-à-dire qu'on peut demander une instance non partagée de telle ou telle Vue toute seule, ou bien obtenir la Vue plus un ViewModel déjà connecté. Idem pour les ViewModels.

Pour créer une Vue non partagée on écrira :

```
var view = Router.GetNonSharedView("HomeView", null);
var view = Router.GetNonSharedView("HomeView", "HomeVM");
```

La première version créée une Vue non partagée à partir du nom de contrat MEF "HomeView", mais sans passer de nom pour le ViewModel. On obtient ainsi une Vue seule qu'on pourra ensuite connecter par exemple à un ViewModel déjà instancié.

La seconde version créée la route complète et retourne la Vue avec son ViewModel instancié, le tout en mode non partagé.

Pour les ViewModels la logique est la même :

```
var viewModel = Router.GetNonSharedViewModel(“HomeVM”);
```

Ce code créera une instance de "HomeVM" non partagée, et connectée à aucune Vue.

Une chose importante à se rappeler : ne pas utiliser le routeur dans le constructeur du ViewModel. En effet, à cet instant précis les importations MEF ne sont pas encore effectuées (MEF créé l'instance pour ensuite résoudre les importations, c'est très logique mais il faut y penser !). Ainsi, les manipulations des propriétés héritées de BaseViewModel qui fonctionnent sur l'importation MEF (Router, Logger…) doivent être effectuées après s'être assuré que les importations ont été satisfaites.

Pour cela la classe peut supporter l'interface IPartImportsSatisfiedNotification et sa méthode OnImportsSatisfied() qui est un point fiable, appelé par MEF, pour indiquer que toutes les importations de la classe sont satisfaites.

Une autre stratégie consiste à faire un override de la méthode Initialize() héritée de BaseViewModel, en toute logique (et tests effectués) les importations sont satisfaites lorsque Jounce la déclenche.

Un mot sur la documentation :

Malgré les efforts de Jeremy en matière de documentation de Jounce, la meilleure source de documentation pour ViewModelRouter, et d'autres classes du toolkit d'ailleurs, reste le code

source… Je vous conseille de le télécharger en même temps que le binaire. Le code étant propre, court et (un peu) documenté il est « la » référence absolue et à jour.

Créer des routes

La création des routes est un élément essentiel de la programmation MVVM avec Jounce. Une route n'est qu'un couple Vue / ViewModel.

L'exemple du template de projet Jounce nous a permis de voir que la vue exportait sa propre route de la façon suivante :

```
[Export]
public ViewModelRoute Binding
{
   get { return ViewModelRoute.Create("MainViewModel", "MainPage"); }
}
```

C'est la Vue qui se charge de créer et d'exporter la route car sous MVVM une Vue a parfaitement le droit de connaître son ViewModel (puisqu'elle accède à son contenu) alors que l'inverse est interdit.

Toutefois, l'exportation se faisait ici sur des noms de contrat MEF (donc des strings), il est possible de déclarer la route n'importe où.

Dans une application réelle on préfère d'ailleurs regrouper toutes les exportations de routes dans une seule classe séparée.

La création des routes par ce procédé convient à la plupart des applications. On évitera juste d'utiliser des strings en préférant des constantes.

Mais dans certains cas le côté un peu figé de ces déclarations ne satisfait pas les contraintes de l'application. On peut ainsi avoir besoin de déclarer les routes de façon plus dynamiques, au runtime.

Le marquage d'exportation MEF étant codé sous la forme d'un attribut il n'est alors plus possible d'utiliser le principe montré plus haut. On fait alors appel au ViewModelRouter que nous avons étudié à la section précédente.

Une route peut ainsi se créer dynamiquement en suivant ces modèles :

```
Router.RouteViewModelForView(“HomeVM”, “Home”);
Router.RouteViewModelForView<HomeViewModel, HomeView>();
```

Point Intermédiaire

La découverte de Jounce se poursuit ! Les grands principes du toolkit ont été posés : créer une application avec Jounce (surtout en partant du template de projet) n'est pas très compliqué. Mais les subtilités de Jounce sont malgré tout assez nombreuses. Et c'est heureux car appliquer le pattern MVVM dans la réalité pose de nombreux problèmes qu'un toolkit trop « light » ne peut régler.

Jounce bien que construit pour rester léger est inspiré par des frameworks imposants offrant des possibilités très larges. Même en ne gardant que l'essentiel Jounce est marqué de l'empreinte de ses grands frères.

Jounce est simple, mais n'est pas évident pour autant.

Ses nombreuses possibilités nécessitent de comprendre le contexte dans lequel elles prennent tout leur sens. Ce contexte est celui d'applications de taille moyenne à grande. Tout le monde n'a pas forcément la même expérience de telles applications. D'ailleurs Jounce l'annonce clairement, il est conçu pour les applications de type « LOB » et « enterprise ». On n'est définitivement plus dans le monde du petit utilitaire ou de la petite application bricolée dans son coin.

Petit, mais costaud, Jounce nous réserve encore des surprises. Alors poursuivons la route !

BASEENTITYVIEWMODEL

Je parlais des surprises que réserve Jounce, en voici une d'assez belle taille.

BaseEntityViewModel est une alternative à BaseViewModel dont elle descend. On peut donc hériter de la première au lieu de la seconde lorsqu'on créée une classe pour un ViewModel.

On le sent poindre à la présence du mot « *entity* » dans son nom, BaseEntityViewModel est une classe mère qui se destine aux ViewModels ayant à gérer des opérations CRUD, c'est-à-dire à ceux qui manipulent des entités qui peuvent être créées, modifiées, lues ou supprimées.

Dans une application de gestion, cible de Jounce, 100% des écrans ne sont pas des écrans de saisie, mais ils sont très nombreux en général. Du coup, ceux qui utiliseront Jounce dans ce contexte auront plus souvent à hériter de BaseEntityViewModel que de BaseViewModel. C'est donc une classe essentielle, bien en comprendre les ajouts et atouts n'est ainsi pas superflu.

Ses principales caractéristiques :

- S'utilise pour créer des ViewModel gérant des données ;

- Dérive de BaseViewModel (donc propose déjà tous les services de cette dernière) ;
- Implémente INotifyDataErrorInfo qui permet d'utiliser le système intégré de validation des données de Silverlight ;
- Offre une méthode virtuelle ValidateAll() qui permet de valider l'ensemble des données (à *overrider*) ;
- Possède une propriété booléenne Committed (à positionner par programme)
- Offre une commande CommitCommand qu'on peut binder à un bouton et qui est en mode *disabled* automatiquement (tant que Committed est à false et que toutes les validations ne passent pas) ;
- Propose la méthode virtuelle OnCommit qui est appelée quand CommitCommand est exécutée et que le développeur peut *overrider* ;
- Expose la propriété HasErrors qui est à true tant que des erreurs de validation subsistent.

Le but du jeu étant ici de simplifier la validation des données dans les formulaires.

Pour cela Silverlight 4 a introduit l'interface INotifyDataErrorInfo qui permet de gérer la validation asynchrone des données. Le mécanisme complet est hors sujet dans ce Livre Blanc mais je conseille au lecteur de regarder les nombreux exemples qu'on peut trouver sur le Web. D'autant que ce mécanisme est un « touche à tout » puisqu'il y a d'une part l'interface à implémenter, mais aussi les binding à compléter et éventuellement les styles des champs de saisie à relooker (ils contiennent désormais des états indiquant si le champ est valide ou non). Le tout se complète d'un contrôle ValidationSummary qui peut afficher l'ensemble des erreurs, sans parler des Tooltip automatiques précisant la nature de l'erreur.

Jounce n'hésite donc pas à utiliser ce qu'il y a de plus récent (mais largement validé, SL 4 ne vient pas de sortir non plus) et intègre à la classe BaseEntityViewModel la base de la mécanique nécessaire au fonctionnement des validations.

On sent toujours ici cette « modernité » de Jounce cherchant à tirer profit des avancées récentes au lieu de rester calée sur des façons de faire qui remontent aux débuts de Silverlight. L'inconvénient, nous le savons, est que Jounce ne marche qu'avec Silverlight (WPF n'intègre par exemple pas les mêmes mécanismes de validation de données). Mais qu'il est agréable d'utiliser un toolkit qui sent la peinture fraîche !

Le plus simple reste de traiter cela par un exemple (BaseEntityVM dans la solution des exemples).

L'exemple BaseEntityVM

Le résultat final

J'aime bien commencer par la fin quand je commente un exemple. Il est à mon sens très frustrant de lire des pages d'explications pour découvrir à la fin « quelle tête cela fait ». Donc sans plus tarder voici l'application en cours de fonctionnement (voir la figure 9 ci-dessous).

Il s'agit d'une fenêtre de saisie comportant quatre champs, chacun ayant ses exigences propres de validation.

Lorsqu'un champ n'est pas valide il s'entoure d'un filet rouge (comportement par défaut de Silverlight), et si le champ est celui qui possède le focus, un Tooltip s'ouvre pour expliquer ce qui ne va pas.

L'exemple est complété d'un résumé de validation qui permet à l'utilisateur d'avoir sous les yeux la liste de toutes les erreurs de saisie.

Bien entendu, malgré le petit effort de présentation que j'ai fourni, il s'agit là d'une démonstration hyper basique, tant au niveau fonctionnement qu'au niveau du look.

Figure 78 - L'application BaseEntityVM en cours d'utilisation

Ce qu'il faut noter sur cette capture :

- Le filet rouge autour de « Nom » (le champ est obligatoire)
- Le filet rouge complété du Tooltip sur le champ Email en cours de saisie
- Le résumé des erreurs (les noms des propriétés ne sont pas traduits, mais cela est possible)
- La présence de deux boutons
    - Sauvegarder, actuellement à l'état disabled car il existe des erreurs sur la fiche
    - Annuler, actif, permettant d'effacer toute la fiche

Le code, généralités

Tout d'abord je suis parti d'un template de projet Jounce identique à celui que je vous ai présenté. Cela évitera les redites et nous permettra de nous concentrer sur ce qui change.

J'ai conservé le principe de l'interface pour le ViewModel ainsi que de deux implémentations, l'une de design, l'autre de runtime. Nous avons pu voir que cette stratégie, un peu plus lourde qu'un seul ViewModel assurant les deux fonctions, offre en réalité des avantages qui s'avèrent plutôt intéressants comme la possibilité de dialoguer entre deux ViewModels sans passer par des messages.

Je préfère donc payer le prix, assez faible, d'un code un peu plus complexe mais qui me laisse des degrés de liberté qui faciliteront le travail à un moment ou un autre. Bien entendu dans cet exemple simplifié ce moment ne viendra jamais… Mais dans une véritable application il finirait par arriver presque à coup sûr !

Donc une interface, deux ViewModels, et un ingrédient nouveau : BaseEntityViewModel.

L'interface du ViewModel

L'exemple expose quatre champs et une commande d'annulation. La commande de Commit n'est pas incluse dans l'interface puisqu'elle est fournie directement par la classe BaseEntityViewModel.

```
namespace BaseEntityVM.Common
{
   public interface IMainViewModel
   {
      IActionCommand CancelCommand { get; set; }
      string FirstName { get; set; }
      string LastName { get; set; }
      string PhoneNumber { get; set; }
      string Email { get; set; }
   }
}
```

Rien de particulier ici, sauf la commande Cancel qui est de type IActionCommand, un type fourni par Jounce et qui supporte ICommand.

Le ViewModel de conception

Comme vous le savez maintenant, ce ViewModel est utilisé uniquement en conception sous Blend ou Visual Studio pour fournir des données factices simplifiant la mise en page.

La classe n'a pas besoin d'hériter de quoi que ce soit, au contraire, elle doit rester le plus simple possible et se contenter d'implémenter l'interface en retournant des données de test.

```
using System;
using BaseEntityVM.Common;

namespace BaseEntityVM.SampleData
{
  public class DesignMainViewModel : IMainViewModel
  {

    #region IMainViewModel Members

    public Jounce.Core.Command.IActionCommand CancelCommand
    {
      get { return null; }
      set { throw new NotImplementedException(); }
    }

    public string FirstName
    {
      get { return "Olivier"; }
      set { throw new NotImplementedException(); }
    }

    public string LastName
    {
      get { return "Dahan"; }
      set { throw new NotImplementedException(); }
    }

    public string PhoneNumber
    {
      get { return "01-23-45-67-89"; }
      set { throw new NotImplementedException(); }
    }

    public string Email
    {
```

```
        get { return "odahan@gmail.com"; }
        set { throw new NotImplementedException(); }
      }

      #endregion
    }
}
```

Comme j'utilise *Resharper*, un add-on de Visual Studio dont je ne saurai plus me passer, écrire « tout » ce code n'est en réalité qu'une question de quelques clics.

Par sécurité, les commandes sont retournées à null. Aucun code d'aucune sorte n'a besoin ni ne doit être déclenché lors de la conception. En revanche la propriété sera utilisée pour binder un bouton à la commande.

Les propriétés elles-mêmes ne font rien d'autre que retourner des valeurs valides. Par sécurité le setter de chacune déclenche une exception de type NotImplementedException qui normalement ne sera pas levée. Mais sa présence nous permettra d'être avertis si quelque chose, au design, cherchait à modifier une valeur. Cela est plus fiable qu'un setter vide (et *Resharper* sait le faire sans que j'ai eu besoin de taper le code).

Il n'y a rien de plus à dire pour ce ViewModel de conception.

Le ViewModel de runtime

Il y a en revanche plus à dire ici. Pour simplifier j'ai découpé le code en quatre régions que nous étudierons séparément.

La première chose importante à noter est que la classe MainViewModel descend de BaseEntityViewModel au lieu de BaseViewModel et qu'elle implémente l'interface IMainViewModel.

```
namespace BaseEntityVM.ViewModel
{
  [ExportAsViewModel("MainViewModel")]
  public class MainViewModel : BaseEntityViewModel, IMainViewModel
  {
    . . .
```

L'exportation se fait de la même façon, par l'attribut ExportAsViewModel en passant le nom de contrat MEF (il est préférable dans un véritable projet d'utiliser une constante qu'une chaîne de caractères).

La région IMainViewModel

Partons du plus simple qui permettra de remonter le fil, l'implémentation de IMainViewModel, notre interface spécifique à ce ViewModel de la page Main.

```
#region Implementation of IMainViewModel

private string firstName;
private string lastName;
private string phoneNumber;
private string email;

public IActionCommand CancelCommand { get;  set; }

public string FirstName
{
   get { return firstName; }
   set
   {
      firstName = value;
      RaisePropertyChanged();
      ValidateName(ExtractPropertyName(() => FirstName), value);
   }
}

public string LastName
{
   get { return lastName; }
   set
   {
      lastName = value;
      RaisePropertyChanged();
      ValidateName(ExtractPropertyName(() => LastName), value);
   }
}

public string PhoneNumber
{
   get { return phoneNumber; }
   set
   {
      phoneNumber = value;
      RaisePropertyChanged();
      ValidatePhoneNumber();
   }
}

public string Email
```

```
    {
      get { return email; }
      set
      {
        email = value;
        RaisePropertyChanged();
        ValidateEmail();
      }
    }

    #endregion
```

On voit que la commande d'annulation est simplement déclarée. Elle est initialisée dans le constructeur que nous verrons plus loin. Ce n'est donc qu'une simple propriété automatique (sans « *backing field* » donc).

Les quatre propriétés sont implémentées de façon très proche pour ne pas dire identique : le getter retourne le champ interne, le setter se décompose comme suit :

- Affectation de la valeur
- Avertissement du changement de valeur
- Validation de la valeur

Pour simplifier j'ai omis une étape préliminaire qui reste essentielle dans une véritable implémentation : les tests d'entrée classiques d'un « bon » setter :

- Vérifier si la valeur n'est pas identique à celle en cours et faire un return dans l'affirmative.

C'est peu de chose mais comme la modification d'une valeur entraîne des traitements (validation, propagation des bindings, etc.) il est toujours judicieux de stopper cette séquence s'il n'y a pas de véritable changement de la valeur.

Si nous n'utilisions pas le système de validation, les tests d'entrée de la valeur devraient aussi s'assurer que la valeur transmise est acceptable. Dans la négative une exception serait levée.

L'exemple est simplifié à l'extrême et n'implémente pas forcément toutes les bonnes pratiques, mais cela ne veut pas dire qu'il faut les oublier !

Quant à l'aspect « validation » de ce code, il se limite à appeler dans chaque setter une méthode particulière comme par exemple ValidatePhoneNumber() pour le numéro de

téléphone. Ce sont ces méthodes qui vont interagir avec BaseEntityViewModel pour faire jouer les mécanismes de validation de Silverlight.

La région des validations

C'est ici que les données sont validées et que les services de BaseEntityViewModel sont utilisés.

```
#region validations
private void ValidateName(string prop, string value)
{
    ClearErrors(prop);
    if (string.IsNullOrEmpty(value))
    {
        SetError(prop, "Ce champ est obligatoire.");
    }
}

private void ValidatePhoneNumber()
{
    var prop = ExtractPropertyName(() => PhoneNumber);
    ClearErrors(prop);
    if (string.IsNullOrEmpty(phoneNumber) ||
        !Regex.IsMatch(phoneNumber, @"^([-. ]?[0-9]{2}){5}$"))
    {
        SetError(prop,
"Numéro de téléphone français de type 01-23-45-67-09. Tiret, point ou aucun séparateur.");
    }
}

private void ValidateEmail()
{
    var prop = ExtractPropertyName(() => Email);
    ClearErrors(prop);
    if (string.IsNullOrEmpty(email) ||
        !Regex.IsMatch(email, @"^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$",
                RegexOptions.IgnoreCase))
    {
        SetError(prop, "Adresse email non valide.");
    }
}

protected override void ValidateAll()
{
    ValidateName(ExtractPropertyName(() => FirstName), firstName);
    ValidateName(ExtractPropertyName(() => LastName), lastName);
    ValidatePhoneNumber();
    ValidateEmail();
```

```
    }

    #endregion
```

La méthode ValidateName() est utilisée deux fois, pour le prénom et nom de famille, car ces deux champs ont les mêmes contraintes. Cela est conjoncturel et serait différent dans une autre application. C'est en tout cas ce qui explique que la méthode reçoit le nom du champ à valider et sa valeur.

On a vu que toutes les méthodes de validations sont appelées à la fin du setter de chaque propriété. C'est une méthode simple qui permet d'être assuré que les données sont valides.

Tant qu'une donnée retourne des erreurs la propriété HasError héritée de BaseEntityViewModel reste à true. Ce qui peut être utilisé partout dans le code ou par un autre ViewModel pour connaître l'état de la saisie en cours.

De même cette classe maintient le champ booléen Committed qui passe automatiquement à false dès qu'une propriété est modifiée (sauf elle-même). Le code doit réinitialiser ce champ lorsqu'il charge des données nouvelles et il peut aussi le forcer dans certaines conditions. Lorsque la commande validation fournie par BaseEntityViewModel se déroule correctement, et après appel à la méthode de persistance (OnCommitted) le drapeau est remis à true.

Les méthodes de validation, en dehors du cas particulier de ValidateName, sont spécifiques à un champ donné : ValidatePhoneNumber pour le numéro de téléphone et ValidateEmail pour l'Email.

La structure de ces méthodes est, comme pour les propriétés, très répétitives. Ceux qui sont adeptes des générateurs de code pourront certainement automatiser beaucoup de choses pour produire encore plus vite leurs applications !

Globalement, une validation s'opère de la façon suivante :

- Un appel à ClearErrors() pour effacer de la mémoire les éventuelles erreurs déjà enregistrées pour la propriété concernée. La séquence qui va suivre rétablira de toute façon les erreurs qui persistent.
- Une série de tests qui, lorsqu'ils détectent une condition d'erreur, font appel à SetError() pour ajouter le message d'erreur.

C'est vraiment très simple. On peut utiliser SetErrors() (au pluriel) si on souhaite indiquer plusieurs erreurs pour le même champ. Dans ce cas on créée une List<string> dans la méthode de validation, liste à laquelle on ajoute chaque erreur (un message = une ligne =

une entrée dans la liste). En fin de méthode s'il y a au moins une erreur (test du Count de la liste par exemple), on appelle SetErrors() en passant la liste.

Chacun doit voir selon les besoins de l'application et en fonction de l'expérience utilisateur (l'UX). Si un champ déclenche 10 erreurs tout simplement parce qu'il est vide ou en cours de saisie, cela peut agacer l'utilisateur et le surcharger d'informations inutiles. Un seul message est alors suffisant. Dans d'autres circonstances le fait que toutes les erreurs soient détaillées est au contraire une aide précieuse, l'utilisateur n'ayant pas besoin de corriger plusieurs fois, découvrant une nouvelle erreur à chaque correction. Il peut d'emblée saisir une donnée correcte. Le choix d'une stratégie ou de l'autre dépend vraiment du contexte.

Je viens d'évoquer des erreurs qui auraient lieu « en cours de saisie ». Ceux qui connaissent déjà le système de validation de Silverlight 4 auront peut-être réagi : de base cela n'arrivera pas. En effet, le mécanisme de Silverlight ne fonctionne que sur la perte de focus…

A cela une bonne raison : de nombreuses données ne peuvent être validées que lorsqu'elles sont finies de saisir. Envoyer des messages d'erreurs incessants à l'utilisateur qui tape un numéro de téléphone pour lui dire qu'il manque des chiffres n'est pas une méthode en adéquation avec la recherche d'une bonne UX.

Mais d'un autre côté, si l'utilisateur ne quitte pas le champ en cours (le dernier saisi par exemple), aucune validation n'a lieu. Elle peut se déclencher plus tard (lors du clic sur un bouton de validation s'il y en a un) mais ce décalage créée un manque de réactivité désagréable, donc une mauvaise UX !

Cornélien…

L'équipe de Silverlight a tranché (pour l'instant), c'est l'option la plus logique qui l'emporte : une donnée est validée une fois qu'elle est totalement saisie, donc sur la perte de focus.

Dans certains cas cela sera suffisant, dans d'autres vous désirerez utiliser une technique plus réactive. L'application exemple montre comment mettre en œuvre un Behavior qui force la validation des TextBox à chaque changement de caractère. Avec les avantages et les inconvénients évoqués plus haut…

Mais revenons aux validations. Il reste une méthode qui n'a pas été présentée : ValidateAll().

Elle est importante puisqu'il s'agit d'un *override*, donc d'une méthode proposée par Jounce dans la classe mère.

ValidateAll(), comme son nom peut le laisser entendre, s'attache à valider l'ensemble des propriétés. Elle est appelée automatiquement par la commande de validation (elle aussi

fournie par la classe mère). S'il y a des erreurs la commande s'arrête. S'il n'y a pas d'erreur la méthode OnCommitted() sera appelée et les données pourront être traitées par l'application. On le verra un peu plus loin.

Le code de ValidateAll() est généralement trivial si on a utilisé la méthode proposée ici (à savoir une méthode spécialisée par champ ou type de champ) : il suffit d'appeler l'une derrière l'autre toutes les méthodes de validation.

Comme on le voit ici, valider les données et retourner des messages clairs à l'utilisateur est largement simplifié par Jounce et son support des validations de Silverlight 4.

Mais regardons maintenant le constructeur…

La région du constructeur

Le constructeur de notre ViewModel contient un peu de code, ce n'est pas énorme mais cela est essentiel au bon fonctionnement de l'ensemble.

```
#region constructor

public MainViewModel()
{
   CancelCommand =
     new ActionCommand<object>(obj => Confirm(), obj => !Committed);
   var committedProp = ExtractPropertyName(() => Committed);
   PropertyChanged += (o, e) =>
                  {
                     if (e.PropertyName.Equals(committedProp))
                        CancelCommand.RaiseCanExecuteChanged();
                  };
}

#endregion
```

Le code du constructeur commence par attribuer une valeur à la commande « CancelCommand », c'est celle qui sera bindée au bouton « Annuler » de la fiche.

La création d'une commande sous Jounce consiste à créer une nouvelle instance de ActionCommand qui généralement se complète par deux expressions Lambda, selon un principe si ce n'est identique au moins très proche de la classe RelayCommand de MVVM Light par exemple.

La première expression fixe le code à exécuter par la commande. En général on préfèrera, comme ici, créer une méthode à part et l'appeler plutôt que de « fourrer » tout un tas de code dans l'expression elle-même. Cela clarifie beaucoup les choses et facilite le débogue.

La seconde expression est une fonction booléenne de type « *Can Execute* ». Si elle retourne false la commande est désactivée. Quand elle est à true, la commande peut s'exécuter.

Je reviendrai plus loin sur les commandes.

Ensuite, le constructeur extrait le nom de la propriété Committed (ce qui évite d'utiliser les noms des propriétés sous forme de chaînes sujettes aux erreurs mais au prix de quelques cycles CPU utilisés par la Réflexion). Ce nom est alors utilisé dans un gestionnaire de l'évènement PropertyChanged. L'expression Lambda utilisée est toute simple, elle sert uniquement à rafraichir le binding du bouton Cancel (de la commande CancelCommand donc) en invoquant RaiseCanExecuteChanged.

La commande de validation des données est fournie automatiquement par BaseEntityViewModel, mais pas la commande d'annulation qui est laissée à la discrétion du développeur.

C'est pourquoi la première n'apparait pas dans notre code alors que nous avons défini la seconde (dans l'interface du ViewModel, IMainViewModel). Jounce se charge d'activer ou désactiver la commande de validation qu'il fournit. Pour la commande d'annulation que nous avons ajouté, c'est à nous de faire ce travail. Ici la commande Cancel sera activée dès que l'état du drapeau Committed aura changé. Peu importe le sens de ce changement. Si Committed change c'est que des propriétés ont été modifiées. On doit alors pouvoir tout annuler.

Il s'agit bien entendu ici d'une implémentation de démonstration, très simplifiée. Dans une application réelle on pourra sophistiquer un peu plus le traitement. Par exemple dès qu'une fiche est lue depuis la base de données une copie peut en être faite. La commande Cancel pourrait alors rétablir les valeurs originales. Ce n'est qu'un exemple, à vous d'utiliser ce mécanisme comme vous le désirez.

La région du code privé

Cette région que je nomme souvent « private stuff », c'est-à-dire « trucs privés » ou « fatras privé », contient tout ce qui n'entre pas dans les autres régions et qui n'est pas public. L'exemple contient un peu de code de ce type :

```
    #region private stuff

    private void Confirm()
    {
       // don't use MessageBox in a ViewModel in a real program !
       var result = MessageBox.Show(
  "Êtes-vous sûr ?", "Confirmez l'annulation svp", MessageBoxButton.OKCancel);
```

```
        if (result == MessageBoxResult.OK) Reset();
    }

    protected override void OnCommitted()
    {
        // don't use MessageBox in a ViewModel in a real program !
        MessageBox.Show("Données enregistrées.");
        Reset();
    }

    private void Reset()
    {
        PhoneNumber = string.Empty;
        Email = string.Empty;
        FirstName = string.Empty;
        LastName = string.Empty;
        Committed = true;
    }

    #endregion
```

Ce code n'est pas le plus beau de l'exemple, par deux fois il utilise une MessageBox pour afficher des messages ce qui est une hérésie sous MVVM. Un ViewModel ne doit rien afficher du tout. Mais pour cet exemple il est évident que je n'allai pas ajouter la complexité d'un service de dialogue.

La méthode Confirm() est appelée dans le code d'exécution de la commande d'annulation (bouton Cancel). Après une demande de confirmation à l'utilisateur les données sont effacées en appelant la méthode Reset(). Cette dernière remet les champs à vide et se termine, chose à noter, par la remise à true du drapeau Committed. Ce qui désactivera automatiquement le bouton d'annulation. Cette remise à vide des champs déclenchera la validation de chaque propriété, et produira des erreurs de saisie (par exemple pour les champs obligatoires qui ne peuvent être vides), ce qui entraînera la désactivation automatique de la commande de validation donc du bouton « Sauvegarder » …

Reste la méthode OnCommitted(). Elle fait partie des méthodes de la classe mère et on l'utilise par un *override*. OnCommitted() est appelée automatiquement par la classe mère quand la commande da validation est exécutée (et après un appel à ValidateAll()).

C'est dans le corps de cette méthode que se trouvera le code qui va persister ou traiter les données de la fiche. Notre exemple se contente d'afficher un message montrant que nous sommes bien passés par la séquence de persistance, puis la fiche est vidée (appel à Reset()).

Tout cela n'est pas bien compliqué, mais il y a pas mal de petites choses à bien comprendre et à bien utiliser.

Avant de passer à la suite, il nous reste à étudier la Vue.

La Vue – Code-behind

Le code de la Vue, MVVM oblige, ne contient pas beaucoup de choses :

```
namespace BaseEntityVM
{
   [ExportAsView("MainPage", IsShell = true)]
   public partial class MainPage : UserControl
   {
      public MainPage()
      {
         InitializeComponent();
      }

      [Export]
      public ViewModelRoute Binding
      {
         get { return ViewModelRoute.Create("MainViewModel", "MainPage"); }
      }
   }
}
```

On retrouve l'exportation avec ExportAsView qui sert à la fois à la modularité de MEF et à Jounce pour recenser les Vues puis l'exportation de la route qui connecte MainViewModel à la Vue (MainPage).

Les remarques déjà faites s'appliquent toujours, à savoir que les routes sont généralement regroupées dans une classe à part et qu'il est préférable d'utiliser des constantes que des strings pour définir les noms de contrat MEF des exportations (et des importations).

Passons ainsi tout de suite à la partie Xaml…

La Vue – Xaml

Le code Xaml de la Vue n'est pas très long et beaucoup de celui-ci concerne la mise à page ce qui ne nous intéresse pas directement ici.

Je vous propose de voir immédiatement les extraits essentiels. Comme par exemple le paramétrage des TextBox qui permettent la saisie des propriétés. Comme ils sont tous bâtis sur le même modèle, prenons le premier :

```
<TextBox Text="{Binding FirstName, Mode=TwoWay, NotifyOnValidationError=True,
ValidatesOnDataErrors=True}" >
        <i:Interaction.Behaviors>
<BaseEntityVM_Behavior:TextBoxChangedBehavior/>
        </i:Interaction.Behaviors>
</TextBox>
```

J'ai bien entendu débarrassé ce code de tout ce qui concerne la mise en page. Reste le binding au champ FirstName (le prénom). On note le mode TwoWay mais surtout l'ajout de NotifyOnValidationError=true autant que ValidatesOnDataError=true.

En soi il ne s'agit pas d'un fonctionnement spécifique à Jounce mais propre au système de validation des données de Silverlight.

Pour rappel (car je ne traiterai pas ce sujet ici), le premier drapeau indique à Silverlight que l'évènement BindingValidationError doit être déclenché sur les erreurs de validation, et le second drapeau transforme certaines erreurs (notamment celles qui pourraient venir d'une implémentation de IDataErrorInfo) en erreurs de validations qui peuvent ainsi être gérées de façon identique.

Ces drapeaux sont importants pour la prise en charge de la validation des données et pour assurer la compatibilité avec du code existant qui utiliserait IDataErrorInfo.

Le code de la TextBox montre aussi le support du Behavior déjà évoqué. Ce Behavior déclenche le rafraichissement du binding à chaque caractère tapé au lieu que cela soit fait sur la perte de focus. Il y a des avantages, mais aussi des inconvénients, comme nous l'avons déjà vu.

Au final la Vue n'a pas à faire beaucoup de choses pour supporter la validation, à la fois parce que Silverlight 4 intègre des mécanismes évolués et que Jounce aplanit les difficultés. Il n'y a donc aucune raison de ne pas utiliser BaseEntityViewModel dès qu'une fiche supporte de la saisie.

On pourra objecter que les fiches qui font de la saisie font aussi souvent pleins d'autres choses et que par exemple le système du drapeau Committed tombe à l'eau si on ajoute une seule propriété qui ne serait pas liée directement à la fiche saisie. C'est un peu vrai. Peut-être le procédé est-il un peu trop simple encore … Mais êtes-vous prêt à vous lancer dans des frameworks encore plus complexes ? Ou préférez-vous adapter votre architecture pour qu'elle fonctionne bien avec un toolkit assez léger comme Jounce ? Ce sont de vraies questions ! Et je ne peux y répondre à votre place. Mais par exemple on peut se débrouiller pour que les saisies soient effectivement faites uniquement par des fiches très simples,

fiches enchâssées dans d'autres, gérant les autres aspects. Et Jounce le permet car il intègre la gestion des régions que nous verrons plus loin…

Il y a donc, au-delà de ce simple Livre Blanc, un travail personnel de réflexion et de pratique pour arriver à trouver la juste façon d'utiliser Jounce. Un Livre Blanc n'est pas une fin en soi, j'aime penser que ce n'est que le début d'une histoire… et cela correspond exactement à sa définition, il y a de belles coïncidences dans la vie !

Point Intermédiaire

Après avoir vu les mécanismes de base Jounce au travers de son template de projet, nous venons d'étudier la façon de mettre en œuvre des mécanismes de validation des données très efficaces.

Arrivés ici nous pouvons constater que Jounce offre une surface bien plus large qu'un toolkit comme MVVM Light, et il nous reste encore beaucoup de choses à voir.

C'est tout ce qui fait l'intérêt de Jounce. Mais il y a toujours deux façons de voir les choses ; certains penseront que Jounce est le génial chaînon manquant entre Prism et MVVM Light, d'autres se diront qu'il en fait tellement que le pas à sauter pour utiliser Prism ou Caliburn.Micro n'est finalement pas si grand… Poser des questions de ce genre me plaît beaucoup plus que d'y répondre, je laisserai ainsi cette interrogation en suspens, préférant largement vous imaginer en train d'y réfléchir ☺

LES COMMANDES

Un des mécanismes essentiels à mettre en œuvre pour respecter le pattern MVVM est d'arriver à transformer la gestion classique des évènements en une gestion de propriétés… Et ce n'est pas une mince affaire.

Traditionnellement, et depuis des langages comme Delphi, Visual Basic ou Java, la gestion des évènements fait partie « du décor », de la panoplie de base du développeur.

La classe Button expose un évènement comme Click, on y connecte un bout de code et l'affaire est jouée. Les environnements de développement eux-mêmes ont évolué pour faciliter au maximum ce style de programmation.

Or, tout chose à une fin. Ce qui semblait répondre à tous les besoins à un moment donné finit plus ou moins rapidement par rencontrer un mur infranchissable : celui de l'explosion des fonctions et de la taille des logiciels, obligeant les informaticiens à revoir leur copie, à inventer de nouvelles méthodes pour maîtriser cette complexité galopante.

MVVM, bien qu'assez récent, est issu d'une mouvance ancienne. MVC, pattern ancêtre de cette même mouvance, a été mis au point en 79 ! Mais tant que le problème réglé restait hypothétique, personne ne s'y intéressait.

Le problème de la gestion des évènements est que cela créée une dépendance codée en dur entre les intervenants. L'essentiel de la nouvelle démarche n'est pas tant de pouvoir changer facilement tel ou tel bout de la chaîne créé par l'évènement, mais de pouvoir cloisonner et isoler l'interface utilisateur aussi radicalement que le sont les objets eux-mêmes, pour en tirer le même bénéfice : conserver la maintenabilité des applications.

L'accroissement de la complexité du code est venu s'immiscer dans la gestion des interfaces utilisateur. Ce mouvement a appelé au même type d'évolution que le code lui-même avait connu : la création de nouveaux langages (comme Xaml), de nouveaux outils (comme Expression Blend), de nouveaux paradigmes (l'UX) et de nouvelles méthodes pour organiser cette montée en puissance.

MVVM est une de ces méthodes.

Et parmi ses principes, tout comme le code a dû se plier à l'encapsulation et l'objectivation pour se modulariser et rester maintenable, la gestion des interfaces se doit d'évoluer vers une même modularisation, un même idéal de découplage.

Mais quoi de plus naturel que de connecter un bout de code sur l'évènement Click d'un Button !

Hélas cela n'est plus envisageable dans une démarche de séparation totale de l'UI.

MVVM, en imposant clairement la fin de l'intrication code / UI impose une nouvelle façon de programmer.

Par exemple, Silverlight a introduit dans sa version 3 la gestion des commandes. C'est-à-dire une simple interface, ICommand. Toutefois le véritable support de cette interface (dans les classes ButtonBase et Hyperlink) est une nouveauté de Silverlight 4.

Mais l'idée est posée : au lieu qu'un bouton déclenche un code pointé par son évènement Click, il expose une propriété de type ICommand à laquelle peut être reliée, *bindée*, une propriété de même type implémentée par une autre classe. Les classes n'ont plus besoin de se connaître, Xaml s'interposant entre elles par le biais du binding.

Cette avancée permet de faire rentrer dans le rang de MVVM la délicate question de la gestion des évènements, en tout cas ceux liés à l'UI et qu'on appelle le plus souvent des

« commandes » (le code pur utilisant plutôt la modularisation via l'injection de dépendances, ce pourquoi MEF est fait et sur lequel s'appuie Jounce).

Tout irait pour le mieux si les besoins en matière de commande depuis l'UI vers le code se limitaient à cliquer sur des boutons et exécuter un bout de code…

Il existe en réalité des dizaines, des centaines d'évènements différents qui n'ont pas pour finalité exclusive l'exécution d'une commande comme le clic d'un bouton.

Par exemple le code peut vouloir suivre les variations de la valeur courante d'un Slider.

Il faudrait une autre interface adaptée et capable de véhiculer la valeur courante. Mais cela n'existe pas. De même que ICommand ne peut être supporté qu'une fois par une classe et qu'il faut ainsi, pour chaque commande, créer une nouvelle classe.

L'une des tâches les plus complexes des toolkits MVVM du point vue technique et méthodologique n'est donc pas de fournir des classes mères pour créer des ViewModels, ni même d'inventer des systèmes de gestion de régions d'affichage. Non, le véritable défi, et la véritable faiblesse de tous les toolkits est bien la gestion des évènements de l'interface…

Comme je viens de le dire, Silverlight a introduit ICommand dans sa version 3. Ce sont les prémices d'une solution au problème des commandes avec MVVM.

Reste trois problèmes à régler pour un toolkit :

- ICommand n'est pas supporté par tous les contrôles ;
- ICommand n'est pas supporté pour tous les évènements même quand la classe supporte la notion de commande ;
- Créer une classe pour chaque commande est très lourd.

Chaque toolkit essaye ainsi, avec plus ou moins d'adresse, de répondre à ces trois problèmes.

Pour éviter d'avoir à créer une classe pour chaque commande la plupart des toolkits expose un type dont le principe consiste à prendre dans son constructeur des délégués ou des expressions Lambda qui prennent en charge les méthodes de ICommand (dont la principale est Execute() qui effectue l'action). Il n'y a plus besoin d'écrire une classe spécifique pour chaque commande, une seule classe (générique dans la plupart des toolkits) suffit.

MVVM Light propose la classe RelayCommand, Jounce offre ActionCommand. Deux solutions très proches qu'on retrouve dans Prism (DelegateCommand).

Cela règle l'un des problèmes. Il en reste deux autres…

Pour ceux-là il n'y a pas de solution simple. Il ne semble pas raisonnable de réécrire tous les contrôles de Silverlight pour rajouter une propriété ICommand équivalente à chaque évènement et spécialisée pour le type des arguments de l'évènement.

Mais à problèmes nouveaux, solutions nouvelles. Et pour cela il faut se forcer à penser dans le contexte des nouveaux outils et langages.

Xaml offre ainsi la possibilité d'étendre le comportement des contrôles par l'ajout de Behaviors. Ce n'est pas un artifice, cela répond à un vrai besoin. En utilisant intelligemment cette possibilité il devient possible de trouver une solution globale aux deux problèmes sus-évoqués.

MVVM Light ajoute le Behavior EventToCommand qui permet de capter un évènement et de déclencher une commande en lui passant éventuellement les arguments originels. C'est une solution certes pratique mais certains pensent que, malgré tout, transporter les arguments des évènements de l'UI vers le ViewModel est une façon détournée de violer MVVM. En effet, s'il n'est pas nécessaire au ViewModel de connaître la classe de l'émetteur cela lui impose d'avoir connaissance du type de l'argument ainsi transmis. Pour être franc je suis assez d'accord avec cette vision des choses.

Jounce propose une approche similaire mais en se basant sur le Behavior InvokeCommandAction qui a été ajouté à la librairie System.Window.Interactivity (qui notamment est celle qui apporte le support des Behaviors).

Ce Behavior agit comme EventToCommand mais se limite à invoquer une commande, avec passage d'un éventuel paramètre de type objet. Le découplage reste donc total. Le Behavior possède une partie Trigger, le déclencheur qui peut intercepter tout évènement, et une propriété Command qui peut être bindée à l'une des commandes du ViewModel. La propriété CommandParameter permet de transmettre un paramètre de type object à la commande.

Cette solution a plusieurs avantages : elle repose sur un Behavior fourni avec Silverlight, il n'y a pas le choix de passer des arguments qui pourraient briser le découplage mais il reste possible de passer un paramètre.

La gestion des commandes peut ainsi se scinder en deux cas : d'un côté les contrôles offrant une propriété Command pour lesquels il suffit de fournir une autre propriété supportant ICommand, de l'autre tous les évènements qui ne sont pas accessible via ICommand.

Dans le premier cas Jounce propose de simplifier l'écriture du code par le biais de la classe ActionCommand que nous verrons à l'œuvre dans quelques instants.

Pour le second cas Jounce nous propose de régler le problème en deux étapes :

- Créer une commande avec ActionCommand ;
- Utiliser le Behavior de Silverlight InvokeCommandAction au sein de l'interface pour se brancher sur l'évènement souhaité et atteindre la commande créée à l'étape précédente.

Nous allons voir un exemple de ActionCommand de Jounce, mais avant je voudrais vous montrer comment régler le cas des évènements qui ne sont pas gérables directement via ICommand et cela sans passer par un toolkit externe pour ne pas mélanger les problématiques. Ajouter Jounce ensuite pour simplifier les choses ne sera qu'une formalité.

Je vais procéder en deux étapes : d'abord un exemple très simple qui va montrer comment gérer le clic (qui n'existe pas) sur un Rectangle, puis en compliquant un peu plus, comment gérer le changement de valeur d'un Slider.

L'exemple du clic sur un Rectangle

Faire plus simple sera malgré tout très difficile… Créez une nouvelle application Silverlight (sans application Web), ouvrez MainPage.xaml et ajoutez un Rectangle sur la grille LayoutRoot.

C'est tout.

Le but du jeu : simuler toute une gestion MVVM avec ViewModel qui pourra intercepter le clic sur le Rectangle. Sans utiliser autre chose que Silverlight.

Cela pose quelques petits problèmes qu'il va falloir régler :

- La simulation d'un ViewModel en quelques lignes de code (mais cela est anecdotique, l'exemple ne sert pas à cela) ;
- L'écriture de la commande ;
- L'utilisation du Behavior de Silverlight pour simuler le clic qui n'existe pas sur le Rectangle et invoquer la commande de notre mini ViewModel.

Ouvrons le code-behind de MainPage et regardons ce que j'y ai ajouté :

```
namespace SilverlightApplication6
{
	public partial class MainPage : UserControl
	{
		public MainPage()
		{
			InitializeComponent();
			LayoutRoot.DataContext = new MiniVM();
		}
```

```
        }

        public class MiniVM
        {
                public ICommand TheCommand {get; private set; }

                public MiniVM()
                {
                        TheCommand = new MyCommand();
                }
        }

        public class MyCommand : ICommand
        {
                public void Execute(object parameter)
                {
                        MessageBox.Show("Clic !");
                }

                public bool CanExecute(object parameter)
                { return true; }

                public event EventHandler CanExecuteChanged;
        }
}
```

J’ai créé une classe MiniVM qui simule le ViewModel. Elle possède une seule propriété de type ICommand, TheCommand. Cette propriété est initialisée dans le constructeur en créant une nouvelle instance de la classe MyCommand.

Cette dernière ne fait rien d’autre qu’implémenter l’interface ICommand *a minima*. Seule la méthode Execute() est intéressante, et encore elle ne fait qu’afficher un message à l’écran.

Mais en une poignée de lignes nous venons de simuler une Vue et son ViewModel !

Cela va même jusqu’à la blendabilité puisque le code Xaml fait pointer le DataContext de LayoutRoot sur une instance de design de MiniVM. Il est donc possible de voir en mode conception les propriétés du ViewModel, comme avec un toolkit super compliqué…

Ce que je veux montrer n’est pas que ces toolkits sont inutilement compliqués mais que *l’essentiel de MVVM se situe dans la façon de penser le code bien plus que dans les toolkits*.

Il nous reste à voir ce que MainPage.xaml contient :

```
<UserControl  …>
```

```
  <Grid x:Name="LayoutRoot"
        d:DataContext="{d:DesignInstance VM:MiniVM, IsDesignTimeCreatable=True}" >
    <Rectangle x:Name="rectangle" Fill="Blue">
      <i:Interaction.Triggers>
        <i:EventTrigger EventName="MouseLeftButtonDown">
          <i:InvokeCommandAction
            Command="{Binding TheCommand, Mode=OneWay}"/>
        </i:EventTrigger>
      </i:Interaction.Triggers>
    </Rectangle>
  </Grid>
</UserControl>
```

J’ai supprimé les déclarations des namespaces et tout ce qui concerne la mise en page. Reste l’essentiel :

- La déclaration du DataContext de conception
- Le Rectangle et son Behavior InvokeCommandAction dont la propriété Command est bindée sur la propriété TheCommand du mini ViewModel.

Simple. Le Behavior nous permet ici de lier MouseLeftButtonDown qui n’est absolument pas compatible avec la gestion des commandes à une commande d’un ViewModel.

Il faut noter que cet exemple a été entièrement codé sous Expression Blend et que le code Xaml a été directement produit par ce dernier en plaçant par drag’n drop le Behavior sur le Rectangle.

Le code est court mais il ne fait pas grand-chose non plus. Si on avait directement programmé l’évènement MouseLeftButtonDown dans le code-behind cela n’aurait pris qu’une ligne et sans rien de particulier dans le code Xaml.

Ce qui compte ici est bien d’avoir réussi le tour de force d’arriver au même résultat mais sans utiliser directement l’évènement et en simulant une séparation totale du code et de l’UI via un ViewModel et du binding.

Mais nous pouvons aller un cran plus loin en utilisant la même simulation MVVM. Nous allons ajouter un Slider qui fera varier l’opacité du Rectangle dont la valeur sera affichée par un TextBlock, mais pas directement. En passant par le ViewModel pour remonter sur le TextBlock. Toujours dans le but d’être au plus proche de MVVM et de l’architecture que le pattern implique.

Pour mieux comprendre la dynamique de cet exemple, je pourrais préciser que tout cela pourrait être réalisé sans aucune ligne de code, uniquement en reliant le Text du TextBlock et Opacity du Rectangle à Value du Slider en utilisant de l'Element Binding. Bien entendu le but recherché n'est pas là et *il ne faut pas se focaliser sur ce que fait l'exemple mais sur la façon dont il le fait*.

L'exemple du Slider

Un Slider est un objet d'interface assez courant. Son principal avantage est de retourner une valeur de type double lorsque l'utilisateur fait bouger le curseur. Mais le Slider, s'il expose des évènements dont ValueChanged qui nous intéresse plus particulièrement, ne propose aucune propriété qui pourrait être bindée à une ICommand.

C'est dans ce type de cas qu'entrent en scène les extensions de la librairie Interactivity. Grâce au Behavior InvokeCommandAction nous allons, *in situ*, dans le code Xaml, pouvoir « attraper » l'évènement ValueChanged pour qu'il déclenche une commande ICommand dans le ViewModel. Cette commande modifiera une propriété du ViewModel, propriété qui sera ensuite bindée au TextBlock. Et comme ce qui nous intéresse principalement c'est la valeur courante du Slider, nous pourrons même, grâce à un Element Binding, récupérer sa propriété Value qui sera passée en paramètre à la commande.

Voici le code-behind qui reprend la même structure que l'exemple précédent, mais adapté au cas du Slider.

```
namespace SLInvokeActionDemo
{
   public partial class MainPage : UserControl
   {
      public MainPage()
      {
         // Required to initialize variables
         InitializeComponent();
         LayoutRoot.DataContext = new MiniVM();
      }
   }

   public class MiniVM : INotifyPropertyChanged
   {
      public ICommand SliderValueCommand { get; private set; }

      private double rectangleOpacity = 0.8d;
      public double RectangleOpacity
      {
         get { return rectangleOpacity; }
         set
```

```
        {
          if (value == rectangleOpacity) return;
          rectangleOpacity = value;
          if (PropertyChanged != null)
          PropertyChanged(this,
                new PropertyChangedEventArgs("RectangleOpacity"));
        }
      }

      public MiniVM()
      {
        SliderValueCommand = new MySliderCommand(this);
      }

      public event PropertyChangedEventHandler PropertyChanged;
    }

    public class MySliderCommand : ICommand
    {
      private MiniVM theVM;
      public MySliderCommand(MiniVM vm)
      {
        theVM = vm;
      }

      public void Execute(object parameter)
      {
        theVM.RectangleOpacity = Convert.ToDouble(parameter);
      }

      public bool CanExecute(object parameter)
      { return true; }

      public event EventHandler CanExecuteChanged;
    }
}
```

On notera :

- La commande MySlideCommand qui implémente ICommand
  - Le constructeur de cette commande qui reçoit en paramètre le ViewModel
  - Sa méthode Execute() qui modifie la propriété RectangleOpacity du ViewModel
  - La façon dont cette dernière extrait la valeur du paramètre qui lui est passé
- La propriété RectangleOpacity du ViewModel qui déclenche PropertyChanged

Si ce code utilisait Jounce, et abstraction faite de la structure minimaliste de l'exemple, on pourrait utiliser ActionCommand en place et lieu de la classe MySliderCommand ce qui raccourcirait d'autant le code.

Le code Xaml de MainPage est le suivant :

```
<UserControl ... >

  <Grid x:Name="LayoutRoot"
   d:DataContext="{d:DesignInstance VM:MiniVM, IsDesignTimeCreatable=True}" >
    <Rectangle x:Name="rectangle" Opacity="{Binding Value, ElementName=slider}"/>
    <Slider x:Name="slider" Maximum="1" Value="0.8">
      <i:Interaction.Triggers>
        <i:EventTrigger EventName="ValueChanged">
          <i:InvokeCommandAction
             Command="{Binding SliderValueCommand, Mode=OneWay}"
             CommandParameter="{Binding Value, ElementName=slider}"/>
        </i:EventTrigger>
      </i:Interaction.Triggers>
    </Slider>
    <TextBlock Text="{Binding RectangleOpacity}" VerticalAlignment="Bottom"/>
  </Grid>
</UserControl>
```

Comme précédemment j'ai supprimé les déclarations de namespaces ainsi que le code de présentation.

On notera :

- La propriété Opacity du Rectangle qui est bindée à la propriété Value du Slider (Element Binding)
- Le Behavior InvokeCommandAction accroché au Slider sur son évènement ValueChanged
  - Le binding de la commande sur SliderValueCommand du ViewModel
  - Le binding de CommandParameter sur la valeur du même Slider ce qui transmet la valeur à la commande
- Le TextBlock dont la propriété Text est bindée à la propriété RectangleOpacity du ViewModel.

Comme pour l'exemple précédent, le projet a été construit sous Expression Blend et le Behavior a été déposé par drag'n drop. Si vous voulez travailler vite et bien avec Xaml, utilisez Expression Blend.

Visuellement le résultat est le suivant (figure 10) :

Figure 79 - InvokeCommandAction et le Slider

ActionCommand

ActionCommand est la classe proposée par Jounce pour créer des commandes.

Elle évite la création d'une classe spécialisée pour chaque commande et offre la possibilité de transmettre un paramètre typé à la commande.

Cette classe est générique, ce qui permet le typage du paramètre.

Je vais maintenant vous proposer une application typiquement Jounce, bâtie en partant du template de projet Jounce.

Elle va mettre en œuvre deux listes :

- Une liste de personnes
- Une liste de personnes sélectionnées

Au départ de l'application on suppose qu'il existe déjà des données dans la première liste.

Deux boutons placés chacun sous chaque liste permettent de faire passer la personne sélectionnée d'une liste à l'autre.

Les boutons ne sont actifs que s'il existe une sélection non nulle dans la liste source et si les données connectées à cette liste ont un compte supérieur à zéro.

On ne gère ici que des commandes simples sans passer le Behavior étudié plus haut. Comme je l'expliquais, une fois ActionCommand et le Behavior compris séparément, marier les deux solutions est une formalité. Je vous laisse régler cette dernière à titre d'entraînement.

Pour mieux comprendre avec de tels exemples je préfère toujours montrer une séquence d'utilisation. Les explications sur le code qui suivent sont alors plus faciles à comprendre.

Voici donc quelques captures de l'application exemple CommandDemo en cours d'utilisation :

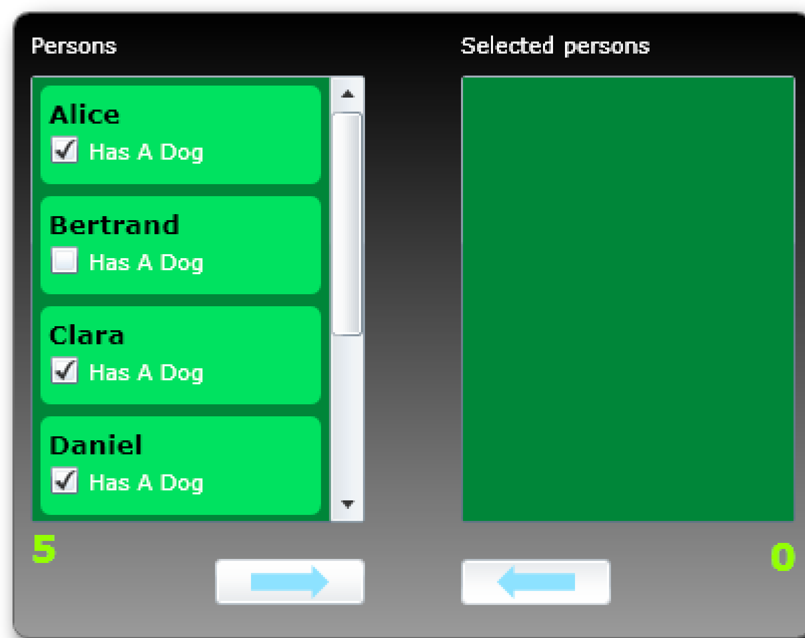


Figure 80 - ActionCommand. Affichage de l'application

Au lancement de l'application la liste des personnes, à gauche, est pré-remplie. On pourrait supposer les données lues depuis un service Web, un fichier XML ou autre.

La liste des personnes sélectionnées à droite est vide.

Sous chaque liste est affiché le nombre de personnes qu'elle contient.

Les deux boutons fléchés permettent, en toute logique, de faire passer une personne d'une liste à l'autre. A l'état initial, les deux boutons sont désactivés par le jeu des ActionCommand (le code qui va suivre permettra de comprendre ce mécanisme).

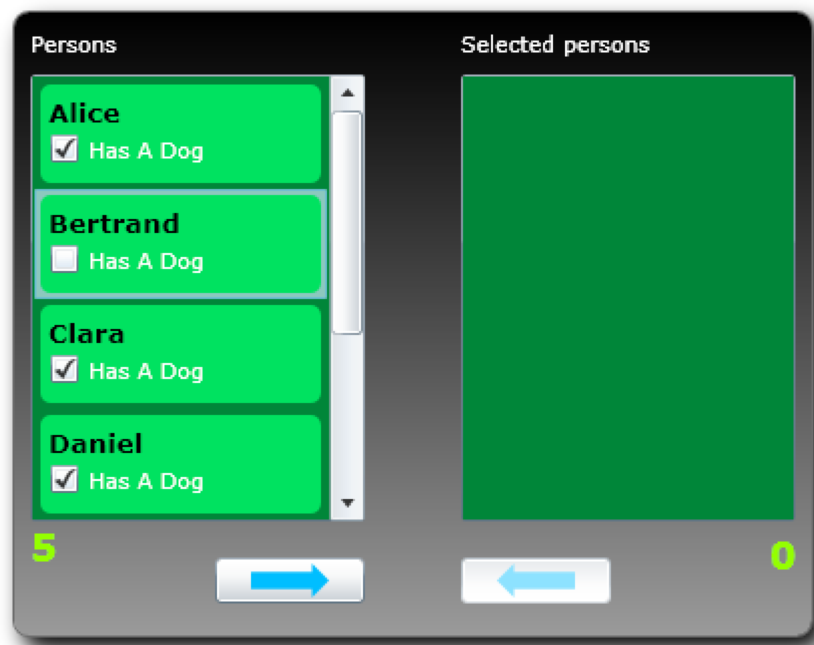


Figure 81 - ActionCommand. Sélection d'une personne

Ici je viens de sélectionner « Bertrand » dans la première liste. Au même instant le bouton permettant de le placer dans la liste de droite s'est activé.

Certains effets visuels comme la sélection d'un item ne sont pas très lisibles, je l'accorde. J'ai créé un look ultra minimum pour faire moins triste que les démonstrations habituelles utilisant les composants bruts de fonderie. Mais je n'ai pas non plus passé des heures à peaufiner tout ça ! Le lecteur saura me le pardonner j'en suis convaincu.

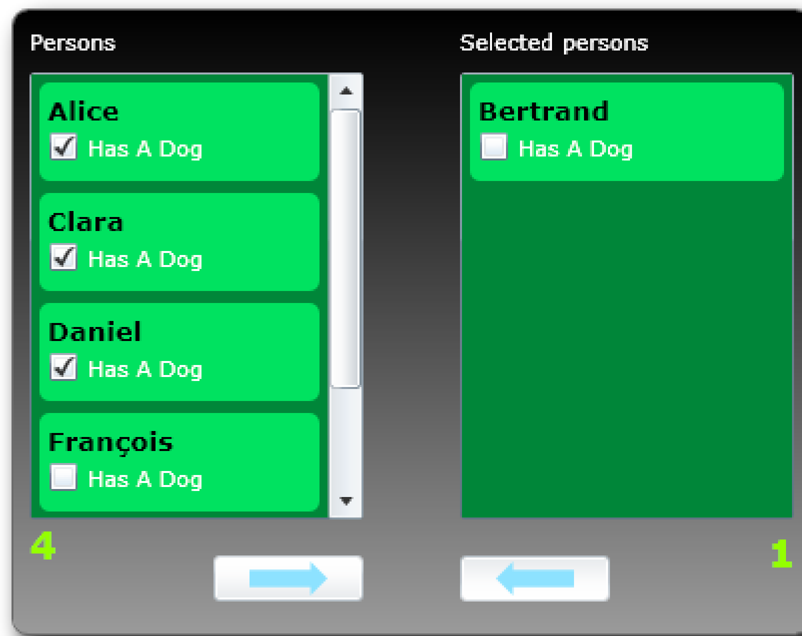


Figure 82 - ActionCommand. Le bouton a été cliqué

Le bouton de droite a été cliqué. La personne sélectionnée (Bertrand) se trouve maintenant dans la liste de droite et a disparu de celle de gauche.

Dans le même temps, les compteurs ont été mis à jour. Il y a bien 4 personnes dans la liste de gauche, et une seule dans celle de droite.

Les listes n'ayant plus aucun élément sélectionné, les deux boutons sont à l'état désactivé.

Fin du film !

Voyons maintenant comment tout cela fonctionne…

Le code de l'exemple

Comme indiqué, cet exemple se base lui aussi sur le template de projet Jounce. La structure globale de ce dernier ayant été présentée tout au long du présent document, je vous propose de nous focaliser sur l'aspect Commande de l'exemple. Le code source étant fourni vous aurez tout le loisir d'éplucher le code !

Mais pour vous aider dans ce voyage, voici les grandes lignes de l'application :

- Un sous répertoire Model contient les données
  - La classe Person est définie. Elle hérite de BaseNotify de Jounce ce qui simplifie le support de PropertyChanged.
  - La classe PersonList gère une liste de Person et fournit une méthode Add() qui permet d'ajouter des personnes à la liste ainsi qu'une méthode

CreateTestData qui peut être appelée pour fabriquer quelques données de test.

- L'interface du ViewModel déclare deux PersonList, la liste de base, à gauche à l'écran, et la liste des personnes sélectionnées, à droite.
    - Elle prévoit aussi deux propriétés retournant le nombre d'éléments de chaque liste.
    - Elle définit deux actions que nous allons voir plus en détail, chaque commande correspondant à l'un des boutons de l'UI.
- Deux ViewModels sont créés, le ViewModel de conception qui ne retourne que des données, et le ViewModel opérationnel qui assure le fonctionnement de la Vue, notamment en implémentant les deux commandes.
- L'UI est très simple, deux ListBox utilisant un DataTemplate pour la mise en forme des données, deux TextBlock pour afficher les comptes, deux boutons pour agir sur les personnes des listes.

Voici l'interface du ViewModel, on y voit la définition des deux commandes :

```
using CommandDemo.Model;
using Jounce.Core.Command;

namespace CommandDemo.Common
{
  public interface IMainViewModel
  {
    PersonList Persons { get; }

    PersonList SelectedPersons { get; }

    int PersonsCount { get; }

    int SelectedCount { get; }

    IActionCommand<Person> ToSelectedListCommand { get; set; }

    IActionCommand<Person> ToPrincipalListCommand { get; set; }
  }
}
```

ToSelectedListCommand est la commande qui envoie une personne de la liste de base vers la liste des personnes sélectionnées.

ToPrincipalListCommand est la commande qui envoie une personne de la liste des personnes sélectionnées vers la liste de base.

On notera :

- Les commandes sont exposées comme des interfaces IActionCommand<>
- Cette interface est générique et supporte un type qui n'est autre que celui de l'éventuel paramètre qui peut être passé à la commande.

Comme notre exemple manipule des objets Person les commandes ont été typées pour recevoir des instances de ce type en paramètre. Cela est bien plus fiable que d'avoir un paramètre de type object non typé (bien que object soit déjà un type précis, mais vous voyez ce que je veux dire). Ici le code des commandes recevra des instances de Person et rien d'autre, utilisant les mécanismes propres de C# pour éviter toute erreur humaine de codage.

L'interface IActionCommand<> existe en version non générique lorsqu'on ne souhaite pas passer de paramètre.

Enfin, cette interface hérite de IActionCommand (non générique) et ICommand, l'interface Silverlight. On se rappelle ici que si l'héritage multiple à la C++ n'existe pas sous C# pour les classes (ce qui est un bien) mais que le principe existe pour les interfaces et qu'il est utilisé ici par exemple.

Si les commandes sont préférablement exposées sous la forme d'interfaces c'est que les interfaces sont déjà un moyen fiable et bien connu d'isoler l'implémentation d'un comportement (ou d'une série de comportements). Les interfaces évitent d'avoir à connaître les véritables instances qui les implémentent et ce vieux procédé retrouve aujourd'hui, et surtout sous MVVM, toute sa raison d'être par le découplage assez fort qu'il propose sans gros efforts à fournir.

Bien entendu, Jounce offre une implémentation de IActionCommand qui peut être utilisée pour créer des commandes. Mais rien n'interdit au développeur de concevoir sa propre classe qui prendrait en charge d'autres aspects. Jounce l'autorise parfaitement. Par exemple, on pourrait supposer une classe implémentant IActionCommand ajoutant l'écoute de certains messages qui désactivent certaines commandes automatiquement… Il n'y a donc rien d'exotique à fournir ses propres implémentations et à les utiliser, bien au contraire !

Quoi qu'il en soit, Jounce propose une implémentation par défaut, ActionCommand<>.

Cette classe créée par défaut une commande vide toujours exécutable. Il est évident qu'on utilise plutôt les constructeurs permettant de passer à la fois l'action à réaliser ainsi que la fonction retournant un booléen indiquant si la commande peut être exécutée ou non.

ActionCommand<> offre en plus une petite curiosité, OverrideAction() qui permet, à la volée, de remplacer l'action réalisée par la commande par une autre action. Dans ce cas un drapeau Overriden est positionné à true. Je ne sais pas dans quel contexte précis l'auteur s'est senti le besoin d'ajouter cet artifice et je ne suis pas convaincu qu'il serve à beaucoup de monde. Mais c'est une bonne illustration de ce que je disais à propos des implémentations personnalisées de IActionCommand, chacun peut ajouter ce qu'il veut, tant que la classe respecte le contrat de base, il n'y a aucune limitation quant aux options qu'elle peut offrir. De toute façon, pour Silverlight, ces classes seront vues au travers de ICommand et rien d'autre.

Voici maintenant le code efficace, celui du ViewModel :

```
[ExportAsViewModel("MainViewModel")]
public class MainViewModel : BaseViewModel, IMainViewModel
{
    #region private fields

    readonly PersonList persons = new PersonList();
    readonly PersonList selectedPersons = new PersonList();

    #endregion

    #region IMainViewModel Members

    public PersonList Persons { get { return persons; } }
    public PersonList SelectedPersons { get { return selectedPersons; } }

    public IActionCommand<Person> ToSelectedListCommand { get; set; }
    public IActionCommand<Person> ToPrincipalListCommand { get; set; }

    public int PersonsCount { get { return persons.Persons.Count; } }
    public int SelectedCount { get { return selectedPersons.Persons.Count; } }

    #endregion

    #region constructor

    public MainViewModel()
    {
        // create some data for demo purpose
        persons.CreateTestData();
        ToSelectedListCommand =
          new ActionCommand<Person>(person =>
                  {
```

```
                    selectedPersons.Add(person);
                    persons.Persons.Remove(person);
                    ToSelectedListCommand.RaiseCanExecuteChanged();
                    ToPrincipalListCommand.RaiseCanExecuteChanged();
                    RaisePropertyChanged(() => PersonsCount);
                    RaisePropertyChanged(() => SelectedCount);
                  },
                person => person != null && persons.Persons.Count > 0);

        ToPrincipalListCommand = new ActionCommand<Person>(person =>
                  {
                    persons.Add(person);
                    selectedPersons.Persons.Remove(person);
                    ToPrincipalListCommand.RaiseCanExecuteChanged();
                    ToSelectedListCommand.RaiseCanExecuteChanged();
                    RaisePropertyChanged(() => PersonsCount);
                    RaisePropertyChanged(() => SelectedCount);
                  },
            person => person != null && SelectedPersons.Persons.Count > 0);
    }

    #endregion
}
```

La seule chose vraiment intéressante dans ce code est la définition des deux commandes. Comme c'est le sujet de cette section, il ne s'agit peut-être pas d'un total hasard ☺

Les deux commandes sont similaires, seul le « sens » dans lequel l'objet Person est véhiculé d'une liste à l'autre change. Prenons alors la seconde commande et détaillons-là.

- La propriété ToPrincipalListCommand est affectée d'une nouvelle instance de ActionCommand<Person>. Nous avons prévu, en effet, de recevoir dans le paramètre des méthodes Execute() et CanExecute() une instance de Person, celle qui sera sélectionnée dans la ListBox correspondante. Nous verrons comment Xaml nous permet de le faire.
- Le premier paramètre passé au constructeur de ActionCommand est une Action<T>, le type étant celui déclaré, donc ici Person. On peut passer le nom d'une méthode qui répond à la signature ou, le plus souvent comme ici, directement coder une expression Lambda. Attention toutefois à ne pas « coller une tartine » de code dans une expression de ce type. Au-delà de quelques lignes il est préférable de créer une méthode séparée.
  - Le code de l'action est assez simple :
    - L'instance de Person passée en paramètre est ajoutée à la liste principale

- La même instance est supprimée de la liste des sélectionnés
- RaiseCanExecuteChanged() est appelé pour toutes les commandes qui sont susceptibles d'être impactées par l'action. Ce point est très important, il assure la synchronisation de l'état *enabled / disabled* des boutons dans l'UI.
- RaisePropertyChanged() est appelé pour toutes les propriétés qui se trouvent être modifiées par effet de bord sans que cela ne passe par leur setter. Les compteurs ici n'ont d'ailleurs pas de setter… Il est donc nécessaire d'avertir l'UI qu'elle doit rafraichir l'affichage de ses propriétés précises.

- Le second paramètre de ActionCommand est une Func<T,bool>, donc une méthode retournant un booléen et prenant en entrée un paramètre de type T, ici Person. Cette fonction assure le CanExecute() de ICommand. Elle est évaluée pour savoir si la commande est disponible ou non. Elle est aussi évaluée lors d'un appel direct à RaiseCanExecuteChanged comme nous l'avons vu dans ce même code. Cela est nécessaire car Silverlight, à la différence de WPF, ne propose pas de CommandManager qui effectuerait automatiquement la mise à jour de l'état des commandes. Le code de l'exemple vérifie que le paramètre passé (l'instance de Person) n'est pas null et que la liste d'origine n'est pas vide.

Oublions l'aspect présentation de l'UI et regardons maintenant dans le code Xaml ce qui est important dans cet exemple : les commandes et leur paramètre.

Le bouton permettant d'envoyer une personne de la liste des sélectionnés vers la liste de base (donc le bouton en bas à droite) est défini comme cela (en supprimant le code de présentation) :

```
<Button
    Command="{Binding ToSelectedListCommand}"
    CommandParameter="{Binding SelectedItem, ElementName=listBox1}" />
```

La propriété Command du bouton est bindée à la propriété ToSelectedListCommand du ViewModel. Pour le paramètre nous désirons envoyer à la commande l'instance de Person qui est sélectionnée dans la liste.

Cela se fait très simplement en bindant la propriété CommandParameter du bouton au SelectedItem de la listbox1 (celle de droite).

Le même mécanisme est utilisé pour le premier bouton.

Point intermédiaire

La gestion des commandes n'est pas très compliquée à implémenter une fois qu'on a compris l'ensemble des mécanismes en jeu. Mais ils sont nombreux pour les raisons évoquées en introduction de cette section.

Les commandes sont essentielles, sans elles aucune interactivité ne serait possible entre l'utilisateur et l'application. La réactivité de l'application, la façon dont elle guide implicitement et visuellement l'utilisateur dépend de la mise en œuvre des commandes (par exemple le signalement de leur état et la prise en compte visuelle de celui-ci). Forcément, tout cela s'accompagne côté Xaml de DataTemplate adaptés, d'animations, de styles et templates de contrôles mettant en valeur les données et leurs états. C'est un tout.

MVVM nous parle d'applications qui *englobent à valeur égale le code et l'UI*, *la maintenabilité et l'UX*. L'étude d'un toolkit est par force assez focalisée sur le code ; mais ne vous y trompez pas, si je ne parle pas de design dans ce long papier, ce n'est qu'en apparence. Tout ce que nous voyons ici, Jounce et ses facilités, tout cela n'a qu'un but : aider au mariage du code pur et dur et de l'UI au service de l'UX. Il ne faut jamais perdre cela de vue.

LA MESSAGERIE EVENTAGGREGATOR

Nous avons vu jusqu'à maintenant beaucoup de moyens et de stratégies qui permettent aux différents modules de l'application de communiquer au sens large du terme. La gestion des commandes en est un exemple. Qu'est-ce que l'envoi d'une commande à un ViewModel si ce n'est une forme de communication ?

De même que les principes d'exportation et d'importation de MEF sur lequel se base Jounce ne sont que des moyens particuliers de communiquer une information (relier une Vue et un ViewModel est un échange d'information et cela permet d'établir des bindings qui sont des moyens de faire voyager l'information, donc de communiquer).

Tous les moyens étudiés jusqu'ici servent un but particulier (binding, relation Vue/ViewModel, Commandes…). Mais il y existe des besoins plus génériques où l'établissement d'une communication s'avère tout aussi indispensable, surtout dans un modèle architectural où le cloisonnement interdit les « contacts » directs. Pour tous ces cas où un « émetteur » doit transmettre des informations à un ou plusieurs « récepteurs » il existe un procédé simple et largement utilisé en programmation : la mise en place d'un service de messagerie.

MVVM Light propose par exemple une classe Messenger, Prism ou Jounce utilisent un procédé identique (bien qu'implémenté de façon sensiblement différente) auquel ils donnent le nom d'Event Aggregator qu'on peut traduire par « agrégateur d'événements ».

En voilà une expression bien savante pour si peu de chose… Je vous avoue que c'est comme dire qu'on possède un *Felis silvestris catus*, au lieu de dire tout simplement qu'on a un chat … Dans notre métier on aime bien donner des noms compliqués ou incompréhensibles à des choses simples, comme si cela rendait le contenu plus sérieux (ou celui qui en parle ? *Vanitas Vanitatum, et omni vanitas*[23] sont les premiers de l'Ecclésiaste…).

Bref vous voilà rassuré, un Event Aggregator sous Prism ou Jounce ce n'est qu'une simple messagerie, vraiment. Rien de plus.

Reste à savoir comment cette messagerie fonctionne sous Jounce.

Les principes sont très simple :

- La classe EventAggregator fournit un service de messagerie qu'on peut importer dans son code (BaseViewModel le fait directement et expose une propriété de même nom : EventAggregator) ;
- Tout code peut être un émetteur de message
- Un message peut être de n'importe quel type (le procédé utilise les génériques)
- Pour recevoir des messages une classe doit, en plus d'importer l'Event Aggregator et de s'abonner aux messages, implémenter l'interface IEventSink<T>

Comparée à la messagerie de MVVM Light, Jounce propose une solution un peu plus contraignante. Sous MVVM Light il suffit de s'abonner à la messagerie ou d'envoyer un message, sans autre condition. Jounce oblige à implémenter une interface puis à s'abonner pour écouter les messages ce qui donne un peu l'impression d'avoir à faire deux fois la même chose.

Toutefois la messagerie de Jounce est assez bien faite. Par exemple elle n'utilise que des WeakReference pour référencer les récepteurs et fait le ménage automatiquement. Elle utilise aussi un système automatique d'invocation qui assure que les messages sont traités par le thread de l'UI, le thread principal (et cela peut se choisir à l'abonnement). La messagerie de MVVM Light ne le fait pas ce qui oblige à le traiter dans le code de chaque récepteur.

[23] « Vanité des vanités, et tout est vanité ! », l'Ecclésiaste est un livre de la Bible hébraïque.

Avantages et inconvénients se balancent, c'est tout simplement une façon différente de faire. Ce qu'on perd en légèreté avec Jounce dans la déclaration des récepteurs, on le gagne ailleurs.

Et à ce titre finalement, la messagerie de Jounce, et bien qu'elle s'en inspire, est assez simple et efficace au regard de l'Event Aggregator de Prism qui est d'une inutile complexité[24].

Pour illustrer le fonctionnement de la messagerie de Jounce je vous propose un scénario très simple : une MainPage, le *shell*, dans laquelle deux vues sont imbriquées de façon simple (ce sont des UserControl simplement posés sur la grille de la fiche mère), la première affiche un fond d'une certaine couleur, la seconde pouvant émettre des messages de changement de couleur.

L'exemple

Le code de cet exemple est une adaptation d'une des démonstrations fournies avec le code source de Jounce. Il est très complet et montre diverses techniques avec un style de programmation différent du mien, et c'est toujours bien de voir des implémentations qui diffèrent un peu. L'exemple rajoute la gestion des exceptions non gérées, toujours par le biais de la messagerie, ce qui est un point intéressant à connaître.

Le visuel

La figure suivante (figure 14) montre le programme en cours d'exécution juste après les deux actions suivantes :

- Sélection de la couleur « Orange » dans la combo de gauche
- Clic sur le bouton « Envoyer »

[24] On trouve d'ailleurs de nombreux messages ou billets sur le Web qui abordent ce sujet épineux.

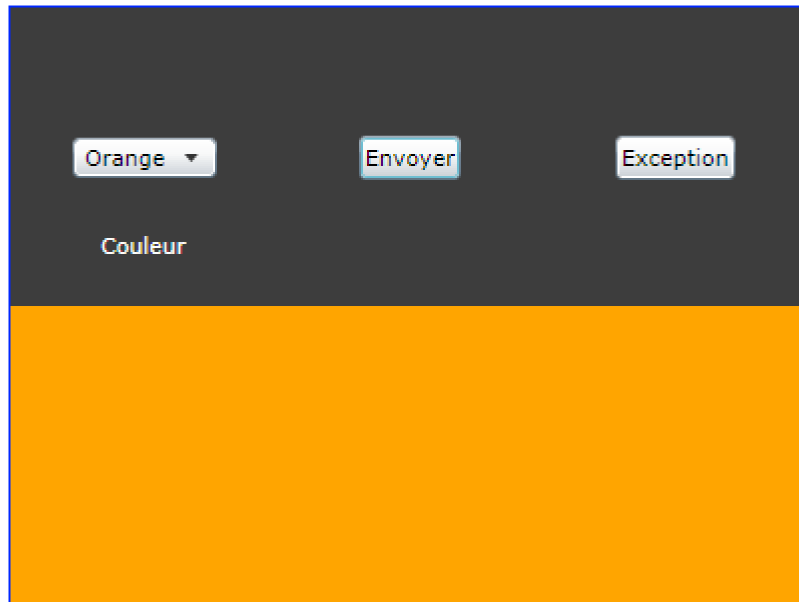


Figure 83 - Event Aggregator - Envoi de message

Ce qu'il s'est passé :

- Le UserControl de la partie supérieure (une Vue) a transmis via l'Event Aggregator le choix de la couleur.
- Le second UserControl de la partie inférieure (une autre Vue) a reçu le message et a changé la couleur de son fond.

Si on clique sur le bouton « exception », une ChildWindow apparait alors (figure 15 ci-dessous).

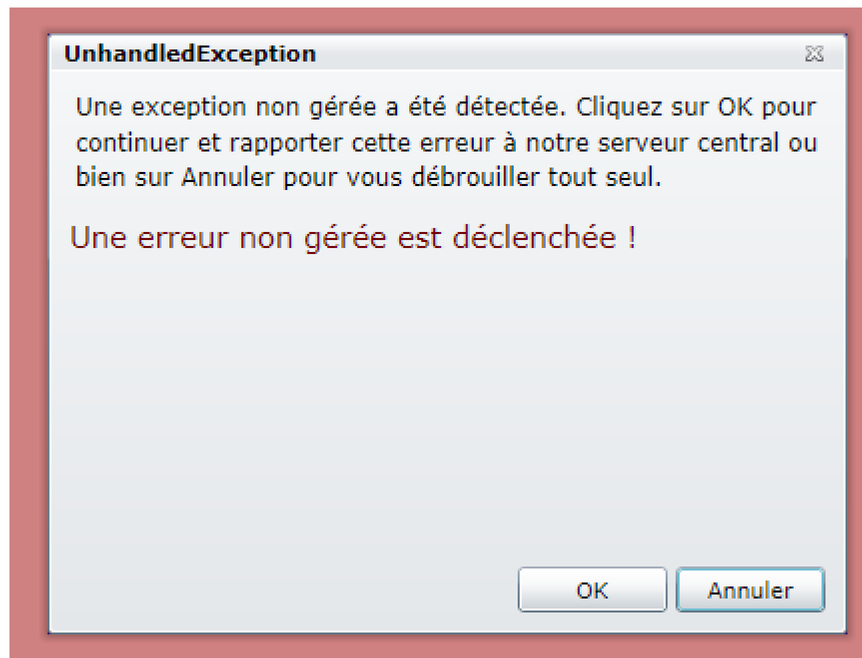


Figure 84 - Event Aggregator - Exception non gérée

Ce qu'il s'est passé :

- Le code de la Vue supérieure a provoqué une exception qui n'est pas gérée par un try/catch (en faisant un throw).
- La gestion interne de Jounce (*l'Application Service*) a intercepté l'erreur et l'a transformée en un message particulier
- La ChildWindow, une troisième Vue, s'est abonnée à ce message particulier et s'est activée dès sa réception.
  - Le message affiché est celui de l'exception (ce qui est ici un choix du développeur)

Si on clique sur OK le message est considéré comme géré et l'utilisateur retourne à l'application. Si on clique sur Annuler, la ChildWindow arrête d'écouter les messages et relance l'exception, ce qui fait planter l'application.

Le code

Je ne vais pas publier tout le code de l'exemple, il est fourni en accompagnement de ce Livre Blanc, car il est malgré tout assez long puisqu'il y a trois Vues secondaires et une Vue principale. Tout cela tourne autour de la messagerie et mieux vaut ici nous concentrer sur ce seul point. Je laisse le lecteur étudier le code directement sous Visual Studio ou Blend pour en sonder l'esprit.

L'émetteur

Comme je l'ai déjà dit, la classe BaseViewModel importe une instance de l'Event Aggregator sous le nom d'EventAggretator. Lorsqu'on créée un ViewModel en héritant de cette classe (ou de BaseEntityViewModel) on dispose donc immédiatement d'un accès à la messagerie.

Si on désire utiliser la messagerie dans un autre code, ce qui est tout à fait possible et même courant, il suffit d'importer l'Event Aggregator de la façon suivante :

```
[Import]
public IEventAggregator EventAggregator { get; set; }
```

C'est exactement ce que fait BaseViewModel d'ailleurs.

Emettre un message est la chose la plus simple dans le mécanisme de l'Event Aggregator :

```
EventAggregator.Publish("Jaune");
```

La méthode générique Publish détecte le type de son paramètre, il n'est donc pas nécessaire de le préciser. L'exemple de code utilise cette technique pour envoyer le nom de la couleur sous la forme d'une string, reconnue comme telle. On peut aussi préciser le type pour lever toute ambiguïté car les récepteurs devront s'abonner avec l'exact même type.

On peut ainsi émettre des messages de tout genre :

```
public class MyParams
{
  public bool ParamBool {get; set; }
  public double ParamDouble {get; set;}
}

…
EventAggregator.Publish<MyParams>(new MyParams{ParamBool=true,ParamDouble=5d});
```

Emettre des messages est donc chose simple, même l'envoi de paramètres complexes. On peut supposer ainsi transmettre des données et le contexte WCF Ria Services qui permettra de les mettre à jour, ou bien l'envoi de données accompagnées d'un callback que le récepteur pourra appeler en fin de traitement du message, etc. Il suffit simplement de déclarer une classe pour chaque type de message complexe dont l'application aura besoin.

Le récepteur

La classe qui souhaite recevoir des messages doit comporter deux choses :

- Elle doit supporter l'interface IEventSink<T> pour chaque type T de message qu'elle souhaite recevoir.
- Elle doit implémenter cette interface autant de fois qu'il y a de types de messages différents (méthode HandleEvent)
- Elle doit s'abonner à la messagerie pour écouter les messages.

Le principe est assez simple, lorsque la classe s'abonne à l'écoute des messages par la commande suivante la messagerie enregistre l'instance du demandeur :

```
EventAggregator.SubscribeOnDispatcher(this) ;
```

Lorsqu'un message doit être transmis, la messagerie balaye une liste de tous les demandeurs inscrits, elle en profite pour supprimer ceux qui n'existent plus (la messagerie utilise des WeakReference) et pour chaque demandeur implémentant IEventSink<T> du type du message à transmettre elle appelle sa méthode HandleEvent(T param). Celle-ci étant le contrat IEventSink<T>.

Cette façon de faire permet à la messagerie de signaler un message à n'importe quelle classe pourvue qu'elle supporte l'interface IEventSink<T>. Et l'abonnement permet d'enregistrer l'instance du demandeur dans la liste de la messagerie.

Le procédé pourrait être simplifié s'il était possible de transmettre une méthode de type Action<T> lors de l'abonnement. La messagerie appellerait cette dernière directement au lieu d'obliger à supporter IEventSink<T>. C'est l'approche choisie par MVVM Light notamment.

Jounce propose ici une façon de faire un peu plus alambiquée mais plus rigoureuse. Au lieu d'autoriser un message à « plonger » dans une méthode quelconque d'une instance, il ne fait « confiance » qu'aux classes implémentant l'interface IEventSink<T>. De même, lors de l'abonnement il est possible de préciser si on souhaite recevoir les réponses directement ou bien via le Dispatcher. Cela est plus souple. La messagerie fonctionne en se protégeant par un Mutex ce qui garantit qu'elle fonctionne correctement en multitâche.

Pour s'abonner à des réponses directes sans utiliser le Dispatcher il suffit d'utiliser la méthode Subscribe() au lieu de SubscribeOnDispatcher(). Pour arrêter l'écoute et se désabonner de la messagerie, il faut utiliser la méthode Unsubscribe().

Il existe un message particulier transmis par Jounce, il s'agit de UnhandledExceptionEvent. C'est *l'Application Service* initialisée dans App.Xaml qui prend en charge la surveillance des exceptions non gérées et qui, via la messagerie, les transforme en message facilement

gérable par l'application (comme l'exemple le montre). Une classe s'y abonne comme à n'importe quel autre message, en supportant IEventSink<UnhandledExceptionEvent> et en s'abonnant via Subscribe ou SubscribeOnDispatcher. Il reste bien entendu à implémenter la réponse dans la méthode HandleEvent comme le montre le code ci-dessous :

```
public void HandleEvent(UnhandledExceptionEvent publishedEvent)
{
    …
}
```

Cela reste malgré tout assez simple.

A noter que comme tous les autres services de Jounce, la messagerie émet des entrées dans le Log automatiquement. Lorsqu'on active ce dernier on peut ainsi savoir exactement quels messages sont transmis et traités et dans quel ordre. C'est un atout indéniable de Jounce sur d'autres toolkits comme MVVM Light car la gestion de message peut vite se transformer en cauchemar s'il n'existe pas un moyen fiable de tracer les événements dans l'ordre où ils interviennent. Le Logger étant disponible partout (soit par héritage de BaseViewModel, soit par simple importation MEF) il est possible d'insérer des messages depuis les modules à déboguer, ils seront ainsi placés correctement dans l'ordre de leur traitement, ce qui permet de savoir si un message intervient trop tôt ou trop tard notamment. Pour avoir eu à déboguer des applications utilisant MVVM Light qui ne dispose pas d'un tel mécanisme je peux vous affirmer qu'il s'agit là vraiment d'une aide précieuse.

NAVIGATION SIMPLIFIEE : EVENT AGGREGATOR ET VIEWROUTER

La messagerie de Jounce fait un peu plus de choses que ce que nous venons de voir. En effet, par son entremise, il est possible de naviguer très facilement dans les Vues.

Ce n'est bien sûr pas l'Event Aggregator qui traite le message, il ne fait que son travail, transmettre des messages à ceux qui s'y abonnent, mais c'est en envoyant un message bien particulier qu'il est possible de demander le chargement d'une Vue.

La classe en charge de traiter ces messages particuliers est instanciée par l'*Application Service*, il s'agit de ViewRouter. Comme son nom l'indique, cette classe est spécialisée dans le routage des Vues et elle offre plusieurs services importants comme le chargement des Vues même depuis des XAP externes.

Pour l'instant laissons cela de côté et regardons comment il est possible de naviguer simplement dans les Vues en envoyant un simple message.

Il faut noter que nous arrivons dans des possibilités de Jounce, qui bien que simples, s'appliquent à des applications possédant plusieurs Vues et ViewModels ce qui interdit des exemples très courts. C'est pourquoi, nous allons juste étudier les possibilités de cette navigation particulière, laissant l'exemple de code de côté. Vous aurez loisir de le consulter vous-mêmes (SimpleNavigation, issu des exemples fournis avec le code source de Jounce).

Naviguer ?

Tout d'abord « naviguer » cela peut vouloir dire beaucoup de choses, notamment sous Silverlight où il existe maintenant un mode de navigation faisant intervenir des classes de type Page au lieu de UserControl. Donc première précision, la navigation Jounce, à la base, s'applique aux Vues qui sont des UserControl, non des pages. Nous verrons que Jounce sait marier sa navigation à celle bien particulière de Silverlight, mais c'est un autre débat.

Dans le sens qui nous intéresse « naviguer » signifie qu'il est possible de charger de une Vue qui va ainsi être affichée.

Il est important qu'il existe un mode de navigation dans le toolkit car sous MVVM les ViewModels ne peuvent rien afficher, donc hors de question de charger une Vue… C'est justement ces petits problèmes d'implémentation qui justifient l'utilisation d'un toolkit.

Jounce supporte les régions d'affichage (que nous verrons plus tard), ce qui signifie qu'une Vue peut en contenir d'autres, obligeant presque les ViewModels à manipuler des Vues. Mais ce n'est pas respectueux de la séparation voulue par MVVM.

Il faut donc un mécanisme qui puisse permettre à un ViewModel de charger une Vue, soit dans le shell, soit dans la Vue qu'il pilote, et ce, sans avoir de « contact direct » avec la classe de cette Vue.

ViewRouter

La classe ViewRouter de Jounce s'occupe en partie de cette tâche, c'est un service de Jounce instancié au lancement de l'application et qui importe toutes les Vues pour offrir un moyen de les charger. Le ViewRouter sait répondre à un message particulier de type ViewNavigationArgs. C'est par le biais de ce message qu'un ViewModel (ou toute autre partie de code) peut indiquer son désir de charger une Vue.

Je dis que ViewRouter prend en charge cette tâche partiellement car s'il peut recevoir l'ordre d'instancier une Vue, voire d'instancier aussi son ViewModel et connecter tout cela ensemble, il lui est impossible de savoir « où » afficher la Vue… Doit-elle remplacer la Vue active dans le Shell ? Doit-elle être simplement « préparée » en vue d'un affichage différé ? Doit-elle être dockée dans une autre Vue et à quel endroit ? Etc.

Il y a bien trop de questions, et de degrés de libertés, pour qu'un toolkit, aussi subtil soit-il, puisse répondre seul à ce genre d'interrogations qui touchent directement à l'organisation et au fonctionnement même de l'application.

De fait, le ViewRouter va répondre au message évoqué plus haut en transmettant un autre message permettant d'accéder à la Vue qu'il a créée. Charge à la partie de code qui désire gérer ce message d'appliquer les mêmes recettes que celles étudiées à la section traitement de l'Event Aggregator.

Le mécanisme

Un code déclenche le chargement d'une Vue en transmettant un message de type ViewNavigationArgs comme cela :

```
EventAggregator.Publish(new ViewNavigationArgs("MaVue"));
```

Ce message est traité par le ViewRouter qui :

- Cherche la Vue et son ViewModel, les instancie et les connecte
- Appelle Initialize() et Activate sur le ViewModel
- Puis publie un message de type ViewNavigatedArgs

Il suffit pour un code quelconque (un ViewModel ou même une Vue) de s'abonner à ce dernier message pour récupérer la Vue et l'afficher comme bon lui semble.

Il existe aussi un Behavior de type Trigger, NavigationTrigger, qui peut être utilisé directement en Xaml pour déclencher le message de navigation ce qui permet d'automatiser encore plus le procédé sans code C#.

Le mécanisme est encore plus subtil que cela puisque via l'extension AddNamedParameter il est possible d'ajouter des paramètres nommés à l'instance de ViewNavigationArgs transmise au départ pour déclencher le chargement de la Vue. Par exemple on peut écrire :

```
EventAggregator.Publish(view.AsViewNavigationArgs().AddNamedParameter("Guid",
Guid.NewGuid()));
```

Ce code utilise deux extensions de Jounce : AsViewNavigationArgs qui transforme une simple string (ici dans la variable view) en ViewNavigationArgs, puis AddNamedParameter pour ajouter à cette dernière instance un paramètre nommé « Guid » dont la valeur est ici un nouveau Guid (pur exemple bien sûr).

On notera l'intelligence de ces extensions qui ont été conçues pour retourner l'instance qu'elles manipulent afin d'être chaînables… Il est ainsi possible de passer plusieurs paramètres en rajoutant des « .AddNamedParameter(…) » les uns derrière les autres.

Le ViewModel accroché à la Vue qui est activée peut overrider ActivateView (de BaseViewModel) pour être averti de cette activation et analyser les éventuels paramètres passés :

```
protected override void ActivateView(string viewName,
  System.Collections.Generic.IDictionary<string, object> viewParameters)
{
   Text = viewParameters.ParameterValue<Guid>("Guid").ToString();
   base.ActivateView(viewName, viewParameters);
}
```

Dans le code ci-dessus, le ViewModel override ActivateView pour traiter l'activation de la Vue. Le paramètre Guid de l'exemple de code précédant est ainsi extrait en utilisant ParameterValue qui est une extension de Jounce sur le type IDictionary<string,object>.

Deux messages sont ainsi utilisés :

ViewNavigationArgs

- On le publie pour notifier Jounce qu'on souhaite charger une Vue
- Le message peut s'accompagner de paramètres récupérables par la Vue et son ViewModel
- Le message peut être utilisé pour désactiver ou activer une vue en positionnant le drapeau Deactivate (appartenant à la classe du message).

ViewNavigatedArgs

- Ce message est publié par le ViewRouter lorsque le message précédent a été totalement traité
- Il suffit de souscrire à ce message pour être notifié de la demande de navigation et récupérer la Vue
- Le même procédé est utilisé par le *Region Manager* pour gérer des régions d'affichage (ce que nous verrons plus tard).

Tout cela est très simple mais est subtil. En peu de code Jounce arrive à proposer des solutions très élégantes qui réclament malgré tout un peu de temps de compréhension pour

être capable de les manipuler toutes dans une véritable application. On est toujours à la limite de la complexité conceptuelle de Prism ou Caliburn, mais en plus light.

De ce point de vue, Jounce est largement plus sophistiqué de MVVM Light et adresse beaucoup plus de cas. Il traite de façon assez complète les différents problèmes posés par l'implémentation de MVVM et s'avère être un toolkit très efficace mais plus complexe qu'on le croirait au départ.

L'exemple

L'application SimpleNavigation est issue des exemples fournis avec le code source de Jounce. Par souci pratique je l'ai intégrée à la solution ODJounceSamples. Cette dernière contient à la fois mes propres exemples et certains issus de Jounce, bruts ou modifiés.

Pour restreindre la taille du document final je ne publierai pas le code source complet de SimpleNavigation ni une analyse trop poussée de celui-ci.

Le principe est assez simple mais montre combien il est difficile de faire des démonstrations courtes dès lors qu'on doit simuler de grosses applications ayant plusieurs Vues et ViewModels.

Néanmoins, le code de SimpleNavigation mérite largement que vous vous y arrêtiez, il utilise une approche vraiment intéressante qui vous permettra de vous imprégner de l'esprit Jounce.

L'application ne fait pas grand-chose, elle expose une Vue qui joue le rôle de menu de navigation (je vous recommande ce passage du code source, la construction est vraiment habile) qui se construit automatiquement en se basant sur les Vues marquées « Navigation » dans leur catégorie (une métadonnée qu'on peut préciser lors de l'exportation). Les boutons de commande qui permettent la navigation sont construits au chargement en utilisant eux aussi les métadonnées de chaque Vue. Ainsi chaque bouton connaît le nom de la Vue qu'il doit charger, le texte qu'il doit afficher (MenuName des métadonnées) et propose même un Tooltip.

On notera accessoirement l'utilisation intéressante de la classe Tuple<>, une nouveauté de Silverlight 4 qui a été moins commentée sur les blogs que les autres.

Ensuite tout le reste est assez simple : il y a trois Vues pilotables par le menu, une qui affiche un cercle vert, une autre un rectangle rouge et une troisième juste un texte.

C'est bien entendu le ballet qui s'exécute autour des messages de navigation et l'architecture globale de l'exemple qui compte. On appréciera lors de la promenade dans le code le travail intelligent à la fois de Jounce et de l'exemple écrit par son auteur.

Je vous laisse vous plonger dans le code !

LES REGIONS

Une application riche, et donc souvent complexe, ne fait pas qu'afficher des pages entières les unes derrière les autres. Limiter MVVM à une telle pratique serait d'ailleurs très réducteur.

Une application de ce type présentera généralement des écrans dont certains au moins seront plus ou moins composites, c'est-à-dire constitués d'informations de natures différentes ou complémentaires qui réclament un pilotage bien précis.

Faire « enfler » un ViewModel pour supporter toutes les subtilités d'une page complexe est vraiment une mauvaise pratique. La modularisation possède un grain très fin, et surtout avec Jounce qui se base sur MEF il serait dommage ne pas se servir des capacités offertes.

Ainsi, une page un peu complexe doit absolument être décomposée en blocs différents gérés de façon unitaire.

Un affichage et un code géré de façon unitaire cela ne vous dit rien ? C'est une Vue et son ViewModel !

Oui mais…

Il est vrai qu'une « Vue » est souvent, et trop vite, associée à une « page affichée », un « écran ». Or une Vue n'est qu'une … vue. Une certaine vision sur des données. Et les Vues peuvent être composées entre elles pour former un « écran » ou une « page ».

Le seul problème qui se pose alors est de savoir comment imbriquer plusieurs Vues dans une autre.

Non pas que le principe soit compliqué, je pense que tout le monde le comprend : Une Vue et son ViewModel peuvent charger d'autres « morceaux », au même titre que la Vue principale et son ViewModel forment le shell qui sait afficher d'autres Vues. Le principe de base est le même.

Ce qui peut être un frein à une telle logique c'est la mise en place d'un mécanisme efficace ne réclamant pas des tonnes de code.

Des toolkits comme MVVM Light ne vont pas jusqu'à offrir de solution pour ce type de problématique. En revanche, bien que light aussi, Jounce s'inspire de toolkits plus gros et plus complets (Prism et Caliburn) qui eux savent gérer la situation.

Et cette solution s'appelle la gestion de *régions*.

Une région est un emplacement dans une Vue qui sera réceptrice d'une autre Vue.

Jounce nous offre la possibilité de définir de telles régions sur la base de trois composants Silverlight selon le résultat qu'on désire obtenir :

- ContentControl, pour charger une instance d'une Vue
- ItemsControl et TabControl, pour charger plusieurs Vues différentes sous la forme de liste ou d'onglets.

Comme je le disais une Vue n'est pas forcément un grande page très complète. On peut fort bien définir une Vue avec une granularité beaucoup plus petite. Le ViewModel est alors restreint lui aussi, et le code, totalement modularisé, devient plus maintenable, plus facile à faire évoluer.

L'exemple

Le projet SimpleNavigationWithRegion reprend le principe et l'essentiel du code de l'exemple précédent. C'est-à-dire trois Vues (cercle, rectangle et texte) dont l'affichage est géré par un menu dynamique (une autre Vue), le tout dans un shell.

Si dans l'exemple précédent seule la navigation simplifiée était utilisée, dans celui-ci cette fonctionnalité est mariée à la gestion de régions.

En réalité les trois Vues sont toujours pilotées de la même façon par le menu dynamique, mais au lieu que le shell décide lui-même de charger chaque Vue à un emplacement précis, par code, c'est la gestion des régions qui va automatiquement faire le travail.

Pour ce faire l'exemple a été modifié à quelques endroits.

Pour gérer des régions il faut :

- Définir dans une Vue le ou les emplacements qui seront des récepteurs de Vues. Ces emplacements sont de fait les fameuses régions. On a observé précédemment que cela pouvait s'opérer sur la base de trois composants de Silverlight.
- Marquer ces emplacements de façon spécifique en leur donnant un nom (par le biais du Behavior ExportAsRegion).
- Exporter les Vues en utilisant un second attribut indiquant le nom de la région.

Le Shell de l'application divise toujours son espace en deux parties, mais cette fois-ci les composants conteneurs sont choisis parmi ceux utilisables avec les régions et ils sont marqués d'un nom de région via le Behavior :

```
<ContentControl Grid.Row="0" Regions:ExportAsRegion.RegionName="NavigationRegion"/>
<ItemsControl Grid.Row="1" Regions:ExportAsRegion.RegionName="ShapeRegion">
```

Le menu sera donc géré comme une région, le conteneur étant un ContentControl puisque le menu est géré par une seule Vue (un UserControl).

La partie réceptrice des Vues navigables est en revanche définie sur la base d'un ItemsControl.

Pourquoi ce choix ?

En réalité, pour simuler le même comportement que l'application précédente un ContentControl serait suffisant (une Vue à la fois). Mais ici la démonstration va utiliser une autre particularité de Jounce : la possibilité de piloter le Visual State Manager des Vues depuis le ViewModel. En utilisant ce procédé un ViewModel peut envoyer un ordre de changement d'état visuel à la Vue qu'il pilote sans briser l'isolation imposée par MVVM.

En utilisant un ItemsControl comme conteneur il sera possible d'afficher plusieurs Vues en même temps. Cela n'est pas utile en soi pour l'application mais cela l'est en revanche pour gérer une transition entre les Vues. Pour cela chaque Vue possède un Groupe d'états à deux possibilités ShowState et HideState. Ces états effectuent un simple changement d'opacité (de transparent à opaque et *vice versa*).

Pour jouer sur cette transition (La nouvelle Vue devenant opaque pendant que l'ancienne devient transparente) il faut bien que les deux Vues soient affichées en même temps, donc que le conteneur puisse le supporter. On trouve donc là la justification de l'utilisation d'un ItemsControl.

L'exemple se base aussi sur un unique ViewModel pour les trois Vues, sachant qu'il n'y a aucune raison d'en créer un pour chacune puisqu'elles ne font pas grand-chose. C'est dans la déclaration des routes (Bindings.cs) que les trois Vues sont attachées au même ViewModel.

Ce dernier override les méthodes ActivateView et DeactivateView pour transmettre à la Vue qui devient inactive le changement d'état visuel vers la transparence et à la Vue qui devient active le changement vers l'opacité, le tout via le pilotage du Visual State Manager de Jounce.

Les Vues navigables sont, elles, exportées de façon classique, mais avec un attribut de plus qui en fait des Vue intégrables dans une région (qui est indiquée). Prenons l'exemple du rectangle rouge, voici son code-behind :

```
namespace SimpleNavigationWithRegion.Views
{
   [ExportAsView("RedSquare",Category="Navigation",MenuName = "Square",
                                ToolTip = "Click to view a red square.")]
   [ExportViewToRegion("RedSquare", "ShapeRegion")]
```

```
  public partial class RedSquare
  {
    public RedSquare()
    {
      InitializeComponent();
    }
  }
}
```

On voit ici que la Vue est exportée une fois de façon standard avec ses métadonnées qui servent notamment à construite le menu et ses boutons, et une seconde fois comme une région qui la relie à « ShapeRegion », le nom de la région réceptrice définie dans le shell.

Comme il est conseillé de le faire, une classe Bindings est créée (Bindings.cs) qui regroupe toutes les routes. On y trouve ainsi les connections entre les Vues et leurs ViewModels comme pour le carré rouge :

```
    [Export]
    public ViewModelRoute Square
    {
      get { return ViewModelRoute.Create("ShapeViewModel", "RedSquare"); }
    }
```

Le shell est exporté avec l'option IsShell =true, ce qui en fera automatiquement la page principale de l'application, Jounce gérant son affichage. Le ViewModel du shell est très simple et se contente d'appeler via la navigation Jounce l'activation du menu :

```
    public void OnImportsSatisfied()
    {
      // publish this so the binding happens
      EventAggregator.Publish(new ViewNavigationArgs("Navigation"));
    }
```

Ce qui déclenche le chargement du menu, l'instanciation de sa Vue et de son ViewModel, leur binding.

La Vue de navigation « Navigation.xaml.cs » est exportée comme une région :

```
  [ExportAsView("Navigation")]
  [ExportViewToRegion("Navigation","NavigationRegion")]
  public partial class Navigation
  { …
```

Ainsi, lors de son chargement elle sera automatiquement placée à l'emplacement indiqué, soit NavigationRegion, définie sur le ContentControl du shell.

Le clic sur les boutons de la Vue du menu déclenche, toujours via les messages de navigation, l'activation de chaque Vue, chacune exportée comme une région et placée automatiquement dans l'ItemsControl du shell, activée ou désactivée visuellement pour créer une transition via la gestion du Visual State Manager de Jounce … Ouf !

Une fois encore les principes sont très simples mais participent à une orchestration assez fine qu'il n'est pas évident de saisir du premier coup. Jounce est puissant, et cela réclame un temps d'adaptation plus long qu'un toolkit plus simple. C'est une Lapalissade.

Comme pour l'exemple précédent je vous conseille vivement de parcourir le code source de l'exemple pour en découvrir toutes les subtilités. Si vous possédez un add-on comme Resharper, n'hésitez pas à l'utiliser pour naviguer dans les classes et traquer les appels qu'elles se font.

Chaque projet exemple possède son propre répertoire Jounce, comme le template de projet le propose, avec sa propre copie de Jounce.dll. Pour l'étude des exemples je vous conseille de supprimer la référence à cette dll et d'ajouter à la solution le code source de Jounce lui-même puis de faire pointer les références vers ce dernier. Il vous sera ainsi très facile, surtout avec Resharper, de naviguer depuis le code des exemples vers les classes de Jounce. C'est un moyen particulièrement efficace à la fois de comprendre les exemples et de se former à Jounce en prenant contact avec son source.

CHARGEMENT DE XAP

Les fichiers XAP ne sont que des ZIP produits par la compilation d'une application Silverlight. Ils contiennent le code de celle-ci et d'éventuelles ressources.

Le temps de chargement d'une application de bureau est crucial, mais il l'est encore plus pour une application Silverlight passant par un réseau, Web, intra ou extranet. Tout ce qui peut diminuer sa taille doit être utilisé pour autoriser un chargement rapide qui ne rebutera pas l'utilisateur.

C'est pour cela que par défaut une application Silverlight est stockée dans un fichier ZIP.

Mais zipper n'est pas suffisant pour une application un peu sérieuse faite de nombreux modules, de nombreuses Vues.

La solution désormais intégrée au Framework .NET 4.0 et à Silverlight de la même version s'appelle MEF, solution que j'ai présentée très en détail dans mon précédent article auquel

je renvoie le lecteur (sa lecture me semble indispensable pour comprendre Jounce si on ne connait pas déjà MEF).

Toutefois, si MEF sait découvrir des modules externes (des DLL en général) pour une application de bureau, la version Silverlight, en raison des limites de ce dernier imposées par une vision ultra sécuritaire chez Microsoft, ne peut offrir le même comportement. En effet, une application Silverlight ne peut en aucun cas aller balayer le répertoire d'un serveur, même si celui-ci l'autorise et même s'il s'agit de « son » serveur (celui dont provient le XAP).

Tout au plus, un XAP chargé depuis un serveur donné peut accéder aux fichiers (XAP ou autres) dont il connait le nom et se trouvant sur ce même serveur.

Ainsi, MEF, en tout cas de base, ne permet pas la découverte automatiquement des plugins sous Silverlight comme il l'autorise pour WPF. Jounce se basant sur MEF ne peut offrir beaucoup plus que ce dernier.

Malgré tout, grâce à un service de déploiement, Jounce permet de définir des routes sur des XAP externes. Lors de l'instanciation de la Vue (ou des Vues) placées dans le XAP externe, Jounce saura charger dynamiquement le fichier depuis le serveur et utiliser son contenu. Utilisant le type Lazy<T> de MEF, Jounce permet ainsi à une application de se charger très rapidement (le XAP principal étant le plus petit possible) et de charger à la demande le code supplémentaire (selon les fonctions appelées par l'utilisateur ou le programme lui-même).

Le code source de Jounce contient de nombreuses démonstrations et plutôt que toutes les recopier ce qui n'aurait aucun sens, je vous les laisse découvrir.

Pour synthétiser :

- La classe ViewXapRoute permet de créer des routes vers des Vues externes en créant des routes précisant le nom du XAP à charger
- Jounce charge le XAP lors du premier appel à la première Vue s'y trouvant (*lazy loading*)
- Il est possible de charger directement un XAP en utilisant Deployment.RequestXap()
- Les XAP externes sont des applications « normales » dont on a supprimé tout le code inutile (comme App.xaml) et dont la référence à Jounce est mise en copie locale = false (ce qui réduit d'autant la taille et peut être utilisé sur d'autres références communes chargées par le XAP principal)
- En implémentant IModuleInitializer les modules externes (comme le principal) peuvent savoir quand ils sont chargés et effectuer certaines initialisations (comme par exemple définir des routes dynamiquement via IFluentViewModelRouter).

La possibilité de charger des XAP à la demande au lieu d'utiliser le lazy loading par défaut doit être étudié attentivement… En effet, lorsqu'un XAP externe est chargé à l'activation d'une Vue l'opération peut prendre du temps et induire un délai qui peut être traduit comme un dysfonctionnement par l'utilisateur. Dans certaines applications il peut être judicieux de lancer le chargement, au moins de certains XAP, dès le lancement de l'application en tâche de fond (ce que fait RequestXap en s'appuyant sur sa gestion de Workflow que nous verrons bientôt).

La déclaration de routes dynamiques est aussi une possibilité ouvrant sur des solutions intéressantes. Comme les routes sont généralement définies par des strings, l'application peut fort bien utiliser ce mécanisme pour lire un fichier XML distant ou local (ou par appel à un service WCF par exemple) dans lequel de nouvelles routes sont définies. Ce qui permet de dynamiser encore plus l'application.

Dans mon article précédent sur MEF je propose une solution de découverte dynamique de XAP externes se basant sur le chargement d'un fichier XML situé sur le serveur. Cette solution permet d'étendre les possibilités d'une application très simplement en simulant une gestion de plugins traditionnelle. Le développeur n'ayant qu'à poser de nouveaux XAP sur le serveur et à modifier le fichier XML. Ce n'est pas aussi automatique que la découverte de DLL par MEF sous WPF mais cela est tout de même assez proche.

On peut parfaitement mixer cette solution, basée uniquement sur MEF, avec Jounce puisqu'il repose sur les mêmes principes. En utilisant un fichier XML posé sur le serveur contenant les noms des modules, une application peut charger dynamiquement des XAP avec RequestXap et définir des routes tout aussi automatiques. Les deux possibilités sont utilisables séparément d'ailleurs. Cela permet d'envisager une modularité maximale qui rendra les plus grands services à toutes les applications composées de nombreux modules. A la fois parce qu'il sera facile d'ajouter des fonctions dans le temps, et parce qu'il sera facile de changer un XAP par un autre plus récent ou limité à certains utilisateurs en modifiant les routes dynamiquement.

Jounce offre ici des briques essentielles qui astucieusement utilisées peuvent voir leur puissance décuplée. Bien entendu, cela se rajoute à tout le reste et fait que Jounce réclame un temps de formation non négligeable pour intégrer toutes ses possibilités et bien définir l'architecture d'une application basée sur ce toolkit. La prise en main de MVVM Light est beaucoup plus rapide, mais certaines limitations ou tout simplement la non prise en charge de certains problèmes que pose MVVM peuvent faire perdre du temps lors du développement. Le choix entre ces deux toolkits « light » doit être bien évalué. Grâce à mes

longs articles et Livres Blancs sur l'un et sur l'autre j'espère pouvoir contribuer efficacement à votre prise de décision !

LOGGER PERSONNALISE

Jounce propose un Logger par défaut qui piste tous les messages et les envoie sur la console de Debug (uniquement ceux de niveau Warning ou plus élevés – modifiables par code).

N'importe où dans le code, et principalement depuis un ViewModel héritant de BaseViewModel où une propriété Logger est exposée, il est possible de modifier le niveau de sévérité des messages qu'on souhaite voir afficher :

```
Logger.SetSeverity(LogSeverity.Verbose);
```

Tous les modules internes de Jounce utilisent le Logger pour laisser une trace de leur activité, qu'il s'agisse d'erreurs ou d'activité normale.

Lorsqu'on utilise Visual Studio pour mettre au point une application, le service par défaut est très pratique et ne nécessite aucune programmation. Il suffit éventuellement dans le shell d'indiquer le niveau de sévérité adapté à ce qu'on souhaite voir (le maximum étant Verbose, le plus restrictif étant Critical).

Toutefois, une application qui peut être facilement déboguée en conception doit aussi l'être en exploitation.

La plupart des applications gère le plus souvent un service de Log laissant des traces soit sur disque (dans l'Isolated Storage généralement pour une application Silverlight, seul endroit où l'application peut écrire sans demande d'autorisation de l'utilisateur), soit en transmettant les erreurs à un serveur via un Web Service ou équivalent. Le mieux étant d'utiliser les deux stratégies car si l'erreur vient du réseau ou de la communication avec celui-ci la dernière technique ne permettra pas de savoir ce qu'il s'est passé.

Créer un Logger personnalisé

Si on désire étoffer le Logger par défaut, et surtout disposer d'un service de Log en exploitation, il est nécessaire d'écrire sa propre classe pour le gérer.

Bien que Jounce importe déjà son propre service par défaut, il accepte qu'une seconde classe implémentant l'interface ILogger soit exportée par l'application. Il remplace alors son propre service par celui proposé par le développeur.

Ce principe ne fonctionne qu'une seule fois : si deux classes s'exportent sur le type ILogger Jounce signalera une erreur.

Si on désire cumuler plusieurs types de Logger il faut utiliser une autre stratégie consistant à implémenter une seule classe de type ILogger recevant les messages et important elle-même une liste de loggers personnalisés (d'un type créé par le développeur) à qui elle transmet les messages qu'elle reçoit. On retrouve ici toute la souplesse désirée en utilisant la modularisation typiquement MEF.

L'interface ILogger est définie comme suit :

```
namespace Jounce.Core.Application
{
   /// <summary>
   ///     Logger interface
   /// </summary>
   public interface ILogger
   {
      /// <summary>
      ///     Sets the severity
      /// </summary>
      /// <param name="minimumLevel">Minimum level</param>
      void SetSeverity(LogSeverity minimumLevel);

      /// <summary>
      ///     Log with a message
      /// </summary>
      /// <param name="severity">The severity</param>
      /// <param name="source">The source</param>
      /// <param name="message">The message</param>
      void Log(LogSeverity severity, string source, string message);

      /// <summary>
      ///     Log with an exception
      /// </summary>
      /// <param name="severity">The severity</param>
      /// <param name="source">The source</param>
      /// <param name="exception">The exception</param>
      void Log(LogSeverity severity, string source, Exception exception);

      /// <summary>
      ///     Log with formatting
      /// </summary>
      /// <param name="severity">The severity</param>
      /// <param name="source">The source</param>
      /// <param name="messageTemplate">The message template</param>
      /// <param name="arguments">The lines to log</param>
      void LogFormat(LogSeverity severity, string source,
                string messageTemplate, params object[] arguments);
```

```
    }
}
```

Un exemple de classe implémentant ILogger se trouve dans l'exemple CustomLogger :

```
[Export(typeof(ILogger))]
public class DebugLoggerViewModel : BaseViewModel, ILogger
{
    private const int CAPACITY = 20;

    private LogSeverity severity = LogSeverity.Verbose;

    /// <summary>
    ///     A queue to hold just the most recent messages
    /// </summary>
    private readonly Queue<string> messages = new Queue<string>(CAPACITY);

    /// <summary>
    ///     Messages
    /// </summary>
    public IEnumerable<string> Messages
    {
        get
        {
            return from m in messages orderby m descending select m;
        }
    }

    /// <summary>
    ///     Sets the severity
    /// </summary>
    /// <param name="minimumLevel">Minimum level</param>
    public void SetSeverity(LogSeverity minimumLevel)
    {
        severity = minimumLevel;
    }

    private void _Enqueue(string message)
    {
        messages.Enqueue(string.Format("{0} {1}",
                DateTime.Now.ToString("yyyy-MM-dd HH:mm:ss.fff"), message));
        if (messages.Count == CAPACITY)
        {
            messages.Dequeue();
        }
        JounceHelper.ExecuteOnUI(() => RaisePropertyChanged(() => Messages));
    }
```

```
    /// <summary>
    ///     Log with a message
    /// </summary>
    /// <param name="severity">The severity</param>
    /// <param name="source">The source</param>
    /// <param name="message">The message</param>
    public void Log(LogSeverity severity, string source, string message)
    {
        if ((int)severity >= (int)this.severity)
        {
            _Enqueue(string.Format("{0} {1} {2}", severity, source, message));
        }
    }

    /// <summary>
    ///     Log with an exception
    /// </summary>
    /// <param name="severity">The severity</param>
    /// <param name="source">The source</param>
    /// <param name="exception">The exception</param>
    public void Log(LogSeverity severity, string source, Exception exception)
    {
        if ((int)severity >= (int)this.severity)
        {
            _Enqueue(string.Format("{0} {1} {2}", severity, source, exception));
        }
    }

    /// <summary>
    ///     Log with formatting
    /// </summary>
    /// <param name="severity">The severity</param>
    /// <param name="source">The source</param>
    /// <param name="messageTemplate">The message template</param>
    /// <param name="arguments">The lines to log</param>
    public void LogFormat(LogSeverity severity, string source,
                    string messageTemplate, params object[] arguments)
    {
        if ((int)severity >= (int)this.severity)
        {
            _Enqueue(string.Format("{0} {1} {2}", severity, source,
                            string.Format(messageTemplate, arguments)));
        }
    }
}
```

Le Logger décrit par le code ci-dessus est destiné à produire des affichages instantanés dans l'application. De fait il ne fait que mémoriser les 20 derniers messages (modifiable par une constante) et proposer une propriété Messages, une liste de chaînes.

La classe est exportée en utilisant un type plutôt qu'un nom de contrat. C'est ainsi que Jounce la reconnaîtra et l'utilisera en place et lieu de son propre service.

L'implémentation étant libre tant qu'on respecte ILogger, tout est bien entendu possible. Le code ci-dessus n'est qu'un exemple très simple.

Pour l'utiliser je suis parti de l'application SimpleNavigation que nous avons déjà étudiée (celle qui utilise les messages de navigation). Elle est assez sophistiquée pour produire suffisamment de messages pour le besoin de cette démonstration (le code du shell émet en plus un message de démonstration après initialisation de la fenêtre de Log, retrouvez-le dans la liste des messages !).

Le shell contient une grille originellement séparée en deux lignes, chacune contenant un ContentControl dont le Content est bindé à une propriété du ShellViewModel. Ce dernier expose les Vues (menu de navigation, Vues chargées suite à la navigation) sous la forme d'objets non typés qui sont soit obtenus dynamiquement (les Vues appelées par la navigation) soit via le Router.

Une fois le code du nouveau Logger intégré à l'application, le plus simple pour s'en servir dans notre exemple est de créer une troisième ligne dans la grille du Shell et d'utiliser un nouveau ContentControl bindé à une nouvelle propriété de type object s'appelant DebugView :

```
<ContentControl Grid.Row="0" Content="{Binding Navigation}"/>
<ContentControl Grid.Row="2" Content="{Binding CurrentView}"/>
<ContentControl Grid.Row="1" Content="{Binding DebugView}" Height="150" />
```

Cette vue n'est qu'un UserControl exporté comme une vue contenant une simple liste avec un DataTemplate simplifié permettant l'affichage de la propriété Messages de notre nouveau Logger.

Nous avons bien au final l'intégration dans le shell d'une Vue proposant les messages, cette Vue est obtenue par un binding sur une propriété du ViewModel du shell.

Mais le « Montage » semble étonnant : cette Vue n'a rien de spécial sauf qu'elle n'exporte aucune route et qu'aucun ViewModel n'a été créé pour elle…

En effet, le Logger étant créé dans tous les cas de figure, il agit comme un Modèle au sens MVVM (c'est une source de données) et MVVM autorise la connexion directe d'un Vue à un Modèle dans les cas simples, ce qui est le cas ici. Le nouveau Logger devrait donc être déclaré dans un répertoire Models par exemple. Mais étant un élément de base de l'application, il pourrait aussi être dans un répertoire Services.

Dans l'exemple j'ai choisi de placer le Logger dans le répertoire des ViewModels, après tout il va être connecté à une Vue et la piloter. La nuance Modèle / ViewModel est ici bien ténue. Le Logger est à la fois un service de l'application, un Modèle puisqu'il fournit des données, et un ViewModel puisqu'il pilote une Vue ! Mais un tel mélange puise sa raison uniquement dans l'extrême simplicité de l'exemple qui, ne faisant que très peu de choses, rend difficile une segmentation bien claire du rôle du Logger. Dans une application réelle, il sera plutôt placé dans un espace de nom Services.

Reste à savoir comment la Vue DebugView va pouvoir être connectée au nouveau Logger comme si ce dernier était un ViewModel standard alors même qu'aucune route n'a été définie et que le Logger n'est pas exporté comme un ViewModel… La Vue elle-même n'est importée nulle part et doit être instanciée.

C'est assez simple et Jounce permet de régler le problème en une seule ligne :

```
DebugView = Router.GetNonSharedView("DebugView", Logger);
```

Cette ligne de code est placée dans la méthode OnImportsSatisfied(), issue de l'interface IPartImportsSatisfiedNotification déjà supportée par l'application. Ainsi, la propriété DebugView est obtenue en créant une Vue *non partagée* à partir de la Vue exportée sous le nom de contrat « DebugView » le tout en la connectant à une instance existante jouant le rôle de ViewModel, ici la propriété Logger du ViewModel du shell, héritée de BaseViewModel et pointant automatiquement sur notre nouveau Logger…

La méthode utilisée n'est pas la seule pour obtenir le résultat mais c'est la plus courte (appel à une seule méthode), en tout cas dans ce contexte précis.

On notera :

- L'utilisation d'un Vue « non partagée », donc une instance unique créée à la volée.
- L'utilisation d'une classe simple non exportée comme un ViewModel sous Jounce pour remplir le rôle de ViewModel.

Le résultat visuel est le suivant :

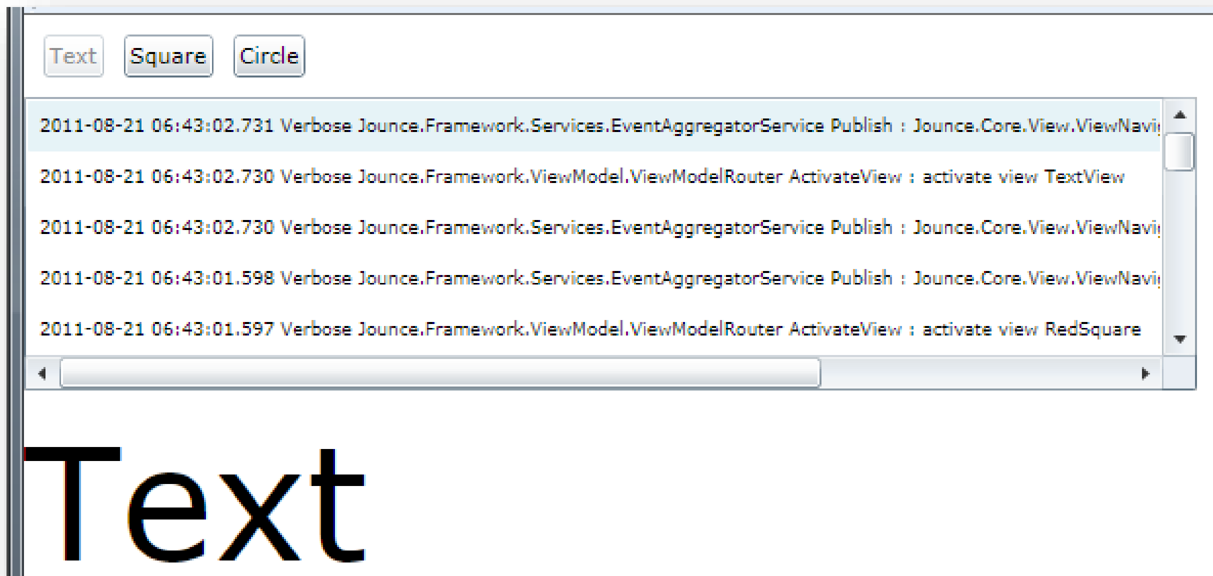


Figure 85 - Logger personnalisé

La figure 16 montre la partie navigation avec ses boutons (en haut), la Vue appelée (en bas, ici la Vue « texte ») et, au centre, la Vue de Log affichant les vingt derniers messages dans l'ordre chronologique inverse (après quelques manipulations et appels de diverses Vues). Le formatage du Log est à la discrétion de la classe qui l'implémente. On pourrait tout aussi bien formater les messages en XML pour les écrire sur un fichier, ou les transmettre à un serveur. De même que la fenêtre de présentation des messages est dockée dans le Shell alors qu'il pourrait s'agir d'une fenêtre secondaire (vers laquelle il faudrait naviguer, ou bien flottante par-dessus le reste de l'application…).

Pour résumer :

Le procédé est simple et efficace. Jounce utilise le Logger pour y placer de nombreux messages de trace qui simplifient le débogue d'une application. Cette dernière peut bien entendu utiliser le Logger pour émettre ses propres messages. Par défaut les messages du Logger Jounce sont émis vers la fenêtre de Debug de Visual Studio. En créant sa propre classe, le développeur peut exploiter traces et messages à sa guise pour les transmettre ou les persister.

> J'ai traité des Vues « non partagées » page 354, je renvoie le lecteur à cette (maintenant lointaine !) section.

WORKFLOWS

Jounce nous aide à résoudre la plupart des problèmes posés par l'implémentation de MVVM. Au fil des pages de ce Livre Blanc, j'ai eu l'occasion de vous présenter les solutions

originales de Jounce. De comparer celles-ci à celles d'autres Frameworks ou toolkits comme Prism, Caliburn ou MVVM Light.

Les « gros » toolkits vont un peu plus loin que Jounce, mais pas tant que ça tellement Jounce a su habilement reprendre l'essentiel en le rendant plus abordable. Ils sont en revanche plus complexes et ce, parfois de façon totalement inutile (comme l'Event Aggregator de Prism).

Jounce n'a donc pas à rougir de ces grands frères. Et de fait il s'éloigne dans l'esprit et la forme assez radicalement de MVVM Light (par endroit trop simple et pas assez complet), tout en restant un toolkit léger, possible à maîtriser en y mettant des moyens raisonnables[25].

Jounce s'offre même le luxe de nous offrir une solution de plus. Elle n'est pas forcément liée directement à la mise en œuvre de MVVM même si les applications suivant ce pattern rencontrent, comme toutes les applications modernes, ce même problème : *l'asynchronisme*.

L'asynchronisme s'insinue partout comme l'eau d'une rivière en crue : envoi et réception de messages, traitement de commandes, tâches de fond, chargement de XAP, WCF, les Ria Services, prise en charge des microprocesseurs multi-cœur, etc.

Orchestrer ce ballet asynchrone, donc aléatoire du point de vue du programme, est un véritable casse-tête dans ce monde de la programmation fondé historiquement sur la nature *déterministe* des ordinateurs et l'aspect *séquentiel* absolu d'un programme (Cf. la machine de Turing et sa longue bande infinie mais séquentielle). Les constructions de type « if then else », les « goto », les « while » ou les boucles « for » n'ont de sens que si le programmeur maîtrise le « trait », le point de code actif et qu'il peut prédire facilement quel autre code sera ensuite exécuté. Ce sont les fondements même de la programmation hors desquels ce n'est plus de l'informatique mais le chaos…

Or, fluidité, réactivité, décentralisation, communications réseau, Internet, et la réalité qui fait que l'augmentation de puissance des machines passe aujourd'hui par la multiplication des cœurs plutôt que par l'élévation de la vitesse de l'horloge de ces derniers, tout cela force à adopter un mode de programmation bien plus étonnant encore que ne fut l'introduction de la programmation évènementielle qui elle-même conduisait à une part d'aléatoire dans le flux d'exécution.

L'asynchronisme est pire que l'évènementiel en ce sens qu'il est de nature parallèle, avec des exécutions réellement simultanées de parties de code différentes. C'est du multitâche,

[25] J'estime malgré tout le temps de formation à Jounce au minimum du double de celui de MVVM Light, si on connaît déjà bien ce dernier et les principes de MEF et de MVVM.

connu depuis de nombreuses années pourriez-vous dire. Oui, mais ici le multitâche est « vrai », ce sont bien des cœurs d'un même processeur agissant de façon autonome et ceux de serveurs de données ou de services qui fonctionnement en même temps. Le parallélisme n'est plus seulement interne mais aussi externe avec ces derniers ! *L'asynchronisme, à gérer, c'est de l'évènementiel mélangé à du multitâche…*

D'ailleurs, si le multitâche est connu de longue date, je peux m'apercevoir lors de formations que, dans la pratique, c'est le flou le plus total qui règne. Rare sont les développeurs aguerris sur ce sujet.

Mais, presque d'un seul coup, ce sont tous les programmes, de tous types, qui doivent gérer l'asynchronisme. Cela n'est plus réservé à certains logiciels précis créés par des équipes possédant « un » spécialiste du multitâche. Tout développeur est confronté à l'synchronisme aujourd'hui.

Or, comme je le disais, il n'y est pas préparé. Comment aborder l'asynchronisme complexe des logiciels contemporains sans maîtriser déjà le multitâche des années passées…

Ni Jounce, ni moi, ne pourront vous transmettre, par magie, la connaissance de ces méthodes de programmation en quelques mots. Même l'étude de librairies comme les RX Extensions conçues pour simplifier le parallélisme réclament du temps. Ne serait-ce que la création basique de classes *thread safe* est en soi un cours de plusieurs jours si on veut être complet et pédagogue.

Je n'entrerai donc pas dans le détail du problème. Juste dans la façon dont il se présente au développeur : l'asynchronisme oblige à prendre charge des évènements arrivant dans un ordre parfaitement inconnu (s'ils arrivent) mais l'application doit le prendre en charge dans un cadre déterministe et séquentiel sinon plus rien n'est programmable.

Ceux qui se sont essayés aux WCF Ria Services et ses interrogations de données asynchrones ont déjà pu comprendre la difficulté d'enchaîner deux ou trois requêtes si elles ont la moindre dépendance entre elles.

Obtenir la liste des factures d'un client à partir de son ID après avoir obtenu ce même ID par une première requête sur la raison sociale du client est pourtant un exemple classique qu'on rencontre, sous cette forme ou d'autres, dans presque tous les programmes de gestion. Hélas, avec les WCF Ria Services (ou tout service distant) il faudra développer un peu de ruse et produire un code peu lisible pour gérer cette « cascade » si on tient à ce que la cause précède toujours la conséquence (obtention de l'ID du client avant de lancer la requête l'utilisant pour retourner ses factures).

La solution, dans l'esprit, consiste à rendre synchrone ce qui ne l'est pas. Vœu pieux ?

Pouvoir écrire un code séquentiel dont chaque ligne est asynchrone, non bloquante, tout en étant sûr que les lignes de la séquence seront exécutées dans l'ordre où elles ont été écrites. Rêve ou réalité ?

Le problème posé n'est pas simple à résoudre. Pourtant Jounce nous propose ici une solution élégante : la gestion de Workflow.

Qu'est qu'un Workflow Jounce ?

D'abord cela n'a rien à voir avec les Workflow de .NET (WWF – Windows Workflow Foundation, introduit dans .NET 3.0). Il s'agit d'un concept et d'une implémentation propre à Jounce.

Synthétiquement, les Workflows Jounce :

- Permettent de déclencher des traitements asynchrones au sein d'une séquence qui elle reste séquentielle
- Les tâches d'un Workflow se définissent en écrivant des classes supportant l'interface IWorkflow
- Jounce propose de base plusieurs implémentations spécialisées comme le WorkflowAction, ou le WorkflowBackgroundWorker, et bien d'autres, qui simplifient grandement la mise en œuvre de Workflows sans avoir à écrire de nouvelles classes.
- Toute l'astuce repose sur la définition (simple) de IWorkflow et de son utilisation à l'intérieur d'un énumérateur C#, le tout complété d'un classe sachant exécuter les Workflows
- On créée un Workflow Jounce en enchaînant une série d'instances supportant IWorkflow au sein d'une méthode énumérable (retournant ces éléments par yield)
- En utilisant WorkflowController.Begin on lance le workflow ainsi défini.

C# propose la notion d'énumérateur. La possibilité de retourner une séquence d'objets en utilisant yield, séquence se présentant « de l'extérieur » comme un IEnumerable<>.

Cela est très puissant bien que peu utilisé. Mais ici Jounce tire parti de ce mécanisme C# en le combinant à une interface pour créer des séquences non bloquantes mais dont l'ordre des étapes reste parfaitement fixé. Le code produit est clair, l'embrouillamini créé par l'asynchronisme disparaît, le développeur peut de nouveau retrouver un moyen séquentiel et fiable d'ordonner les événements qui se produisent *dans et autour* de son application.

Je vous propose de voir concrètement comment cela s'effectue.

L'exemple

Le visuel

Pour mieux comprendre voici à quoi ressemble l'exemple visuellement :

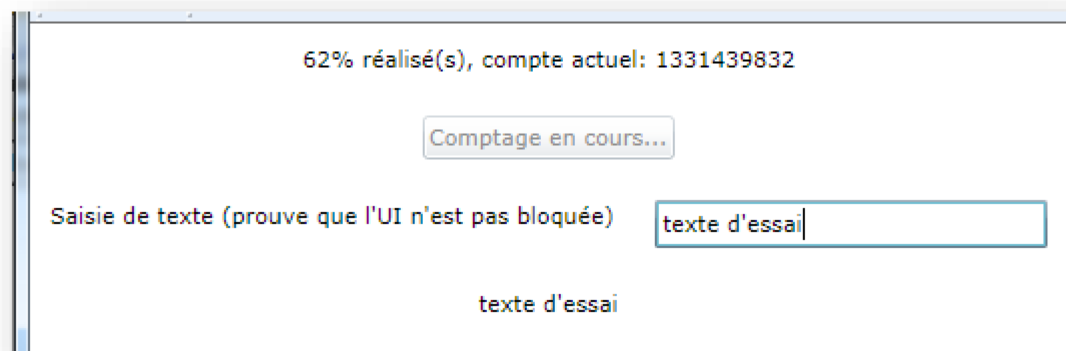


Figure 86 - Workflow

La première ligne de texte retourne des indications en provenance du Workflow qui est exécuté. La figure 17 montre l'une des étapes, un comptage dans une tâche de fond, au moment où 62% de la progression étaient réalisés.

Sous cette ligne se trouve un bouton, actuellement désactivé. Il est manipulé lui aussi par le Workflow et son libellé ainsi que sa fonction varie au cours de l'exécution.

En fait, peu importe ce que fait le Workflow, ce n'est qu'un exemple très simple, il le fait dans un ordre établi et déterminé. Il le fait sans bloquer l'UI, et c'est ce que prouve le reste de la mise en page puisqu'on trouve un TextBox permettant de saisir du texte, texte qui est recopié dans un TextBlock (par le ViewModel et non en utilisant l'Elément Binding, pour prouver que le ViewModel n'est pas bloqué lui non plus).

Le workflow qui est créé par l'exemple est peu vraisemblable, il est conçu pour montrer quelques facettes des classes proposées par Jounce ainsi que la façon de faire fonctionner le mécanisme. Il n'a pas vocation à miner tous les traitements complexes et parallèles d'une véritable application (ce qui réclamerait la mise au point d'une démonstration monstrueuse).

Le code

L'application ne contient qu'une Vue et qu'un ViewModel qui est aussi le Shell. C'est le plus simple qu'on puisse faire.

Lorsque l'application démarre (dans le constructeur du ViewModel du shell) la Workflow est lancé :

```
WorkflowController.Begin(Workflow(),
   ex => JounceHelper.ExecuteOnUI(() => MessageBox.Show(ex.Message)));
```

La classe WorkflowController possède une méthode statique Begin qui prend un ou deux paramètres. Ici elle est utilisée avec deux paramètres. C'est elle qui va « consommer » la séquence créée et la rythmer (bien entendu, dans un thread séparé non bloquant).

Le premier paramètre est le nom de l'itérateur de Workflow, s'appelant tout simplement Workflow() dans l'exemple. Le second est optionnel et définit le code à exécuter si jamais une exception est levée durant l'exécution du Workflow. Dans le cas présent le code utilisé ne fait que récupérer l'exception et demande via un *Helper* de Jounce d'exécuter le Show d'une boîte de dialogue sur le Thread de l'UI.

La recopie du texte saisi dans l'UI pour prouver qu'elle n'est pas bloquée n'a pas d'importance. Il pourrait s'agir de n'importe quel code d'interface réclamant que le thread principal soit libre pour fonctionner. Cela n'a donc pas d'intérêt dans l'exemple (en dehors de prouver le côté non bloquant des Workflows Jounce).

Le code le plus intéressant est bien entendu celui du Workflow lui-même :

```
private IEnumerable<IWorkflow> Workflow()
{
   Button.Visibility = Visibility.Visible;
   Text.Text = "Initialisation...";
   Button.Content = "Pas Encore !";
   Button.IsEnabled = false;

   // attente de deux secondes
   yield return new WorkflowDelay(TimeSpan.FromSeconds(2));

   Button.IsEnabled = true;
   Text.Text = "Premier clic";
   Button.Content = "Cliquez moi !";

   // L'action sur le clic ne fait rien mais cela permet de créer une pause
   // dans le workflow, pause dépendante d'une action utilisateur.
   yield return new WorkflowRoutedEvent(
            () => { }, h => Button.Click += h, h => Button.Click -= h);

   Text.Text = "Maintenant nous allons compter...";
   Button.Content = "Cliquez moi !";
```

```
    yield return new WorkflowRoutedEvent(
            () => { }, h => Button.Click += h, h => Button.Click -= h);

    Button.IsEnabled = false;
    Button.Content = "Comptage en cours...";

    // Comptage dans un thread de background.
    // Le second paramètre peut être omis si on ne désire pas gérer
    // le retour d'information sur la progression.
    // Le thread de background est utilisé pour retourner des infos au
    // thread de l'UI, ce qui est géré, sans avoir à s'en occuper.
    yield return new WorkflowBackgroundWorker(
                    _BackgroundWork, _BackgroundProgress);

    Text.Text = "C'est fini.";
    Button.Visibility = Visibility.Collapsed;
}
```

Le code est celui d'un énumérateur C#. Il ne fait que retourner une séquence de IWorkflow. En soi, la méthode est juste une fonction retournant un IEnumerable<IWorkflow>, c'est aussi simple que cela. On écrit la séquence dans l'ordre souhaité de son déroulement en ne se souciant plus de savoir si les opérations sont asynchrones ou non. C'est déconcertant de simplicité…

Le déroulement du Workflow est cadencé par l'apparition des yield qui retournent chacun une instance de classe supportant IWorkflow. Ce qu'il y a entre chaque yield est exécuté à la suite jusqu'à rencontrer le prochain yield qui, du point de vue de la séquence, est donc un buttoir.

Le yield doit retourner des instances implémentant IWorkflow, cela peut être n'importe quelle classe exécutant n'importe quelle fonction asynchrone (totalement ou en partie). C'est le moteur de Workflow qui va « consommer » la séquence et gérer la suspension et la reprise de son défilement. Et cela grâce à IWorkflow.

Cette interface est définie de la façon suivante :

```
public interface IWorkflow
{
   void Invoke();
   Action Invoked { get; set; }
}
```

Jounce fournit de nombreuses classes implémentant cette interface qui satisfont la grande majorité des utilisations ce qui évite le plus souvent d'avoir à créer des classes supportant IWorkflow. Il reste possible de créer autant de classes personnalisées qu'on le désire. Il suffit juste d'implémenter l'interface. L'exemple en utilise plusieurs comme WorkflowDelay, pour créer une attente, WorfkflowRoutedEvent, pour attendre qu'un évènement routé soit déclenché, WorkflowBackgroundProcess pour exécuter un code en tâche de fond et attendre sa fin tout en pouvant rapporter son activité directement sur le thread de l'UI. Il en existe d'autres.

Pour revenir à IWorkflow, la méthode Invoke() est appelée par le gestionnaire de Workflow. C'est elle qui contient le code de l'action à réaliser, quel qu'il soit (elle peut faire appel à ce qu'elle veut bien entendu). La méthode Inkoded() est une action qui doit être appelée par le code utilisateur une fois la tâche terminée.

On comprend facilement le mécanisme : l'itérateur est utilisé par le gestionnaire de Workflow pour retourner chaque « item » de la séquence. Un item est une instance d'une classe de Workflow implémentant IWorkflow. Grâce à cela le moteur de Workflow sait demander à une tâche de commencer son travail, et il sait aussi, en attendant l'appel à Invoked() quand celle-ci se termine (ce qui lui permet de demander le prochain item de la séquence). Le gestionnaire de Workflow passe donc son temps à attendre l'Invoked() d'un item pour passer au suivant et appeler son Invoke() pour le lancer. Et il attend de façon non bloquante bien entendu.

Le principe est rudimentaire presque. Une idée si simple qu'on en rage de ne pas l'avoir trouvée soi-même ! Facile à comprendre, facile à mettre en œuvre mais qui, par enchantement, transforme un enfer asynchrone en une simple séquence claire, facile à suivre et à maintenir.

Le moteur de Workflow étant indépendant du code de Jounce, il est facile à extraire et à utiliser ailleurs, même dans des applications qui n'utiliseraient pas Jounce. Il s'agit vraiment d'un mécanisme sur lequel je vous invite à réfléchir, à faire des tests, pour le comprendre et l'utiliser partout où vos applications ont à gérer de l'asynchronisme.

VSM AGGREGATOR ET GOTOVISUALSTATE

VSM Aggregator ?

Visual State Manager Aggregator.

Les VSM est cet ajout génial de Silverlight (repris ensuite dans WPF) qui permet de *définir des états visuels* en leur donnant des noms et qui s'occupe de gérer lui-même les *transitions* entre ceux-ci.

Tous les contrôles Silverlight ou presque possèdent des états visuels : aspect du clic, aspect de la souris entrante ou sortante, des validations, etc.

Tout concepteur de contrôle, et même de UserControl peut créer des états pour piloter le visuel de son composant.

Les applications modernes réclament une symbiose parfaite entre les états internes de l'application et les représentations visuelles de l'UI. Le VSM permet de grouper les états par catégories, sait gérer la simultanéité des groupes d'états et se pilote simplement via VisualStateManager.GotoState() et des Behaviors.

Néanmoins, la même modernité impose aussi des patterns comme MVVM, des toolkits comme Jounce. Avec MVVM il n'est pas possible pour un code quelconque, même un ViewModel, de modifier quoi que ce soit qui appartient à l'UI.

> *On dit souvent que le ViewModel pilote la Vue. C'est un abus de langage. C'est l'inverse qui est vrai. La Vue est celle qui transmet les ordres au ViewModel qui lui ne fait qu'exécuter et fournir de nouvelles données à afficher en réponse. C'est la Vue qui est importante et qui pilote le logiciel, non l'inverse, car c'est l'utilisateur, in fine, qui manipule le logiciel au travers de son UI et qui en contrôle le fonctionnement.*

Or, le VSM est un mécanisme purement UI s'il en est.

Placer des appels au VSM dans un ViewModel pour changer l'état visuel d'une Vue serait une hérésie. D'autant qu'avec Jounce il est possible qu'un même ViewModel serve plusieurs Vues simultanément.

Il n'en reste pas moins vrai que la Vue doit refléter les états internes du programme, et *a fortiori* de son ViewModel.

On peut régler le problème ponctuellement, par exemple en faisant en sorte que le ViewModel expose des propriétés simples qui déclencheront des comportements visuels dans la Vue. Une telle approche est celle du BusyIndicator de Silverlight : le ViewModel expose un booléen IsBusy (ou un autre nom) que l'on binde au BusyIndicator, quand le ViewModel est occupé il passe ce booléen à true et automatiquement le BusyIndicator s'affiche pour indiquer à l'utilisateur cet état particulier du programme.

Cela fonctionne. Mais uniquement parce que le BusyIndicator a été conçu totalement pour fonctionner dans cet esprit.

Les changements d'états visuels d'une Vue sont potentiellement infinis et ne dépendent que de la créativité du Designer. Impossible de concevoir toute une série de contrôles adaptés à chaque Vue, à chaque interprétation graphique de chaque Designer !

Il est donc nécessaire de disposer d'un mécanisme respectueux de la séparation MVVM permettant toutefois au ViewModel de signifier à la Vue qu'elle doit changer d'état.

La version basique d'un tel mécanisme n'est pas très compliquée. Avec MVVM Light par exemple, cela se soldera par l'envoi d'un message depuis le ViewModel, message auquel s'abonnera la Vue qui, dans son code-behind, le traduira en une modification visuelle, voire un appel au VSM.

Cette méthode marche, mais elle est lourde, fait une fois encore intervenir la messagerie pour un oui ou un non, fait entrer de l'asynchronisme là où on n'en veut pas, et tend à créer non plus du code spaghetti, mais du « message spaghetti », ce qui, du point de vue de l'horreur à déboguer est peut-être encore pire. Sans parler des délais qui apparaissent puisque les messages ne sont ni bloquants ni immédiats (ils sont dispatchés) et que certains montages intellectuels pour obtenir certains effets visuels tombent à l'eau, noyés dans la complexité de ces petits délais qui ne s'enchaînent pas comme on le pensait (expérience faite !).

Il y a donc un réel besoin de disposer sous MVVM d'un mécanisme permettant de contrôler les états visuels des Vues depuis le ViewModel sans pour autant briser la séparation induite par le pattern et sans tomber dans le « tout messagerie » qui mène à l'enfer.

Jounce propose en réalité deux solutions qui répondent en tous points à cette légitime attente : Le GotoVisualState de BaseViewModel et le VSM Aggregator.

GotoVisualState

C'est la solution la plus simple et la plus directe. BaseViewModel propose une méthode GotoVisualState. Le ViewModel peut ainsi directement piloter sa Vue (ou ses Vues, la méthode peut utiliser un paramètre au nom d'une Vue particulière).

Comment Jeremy s'y est-il pris pour que cela ne viole pas MVVM ?

La question est intéressante parce que la réponse met en lumière une certaine façon de penser la programmation moderne : l'inversion de contrôle.

En réalité cette méthode n'en est pas une. C'est une simple propriété définie comme suit :

```
public Action<string, bool> GoToVisualState { get; set; }
```

C'est à dire que le ViewModel expose une propriété de type Action<string,bool>. Comme ce type est celui d'une action, donc d'une méthode, le code du ViewModel a l'impression de ne faire qu'appeler une méthode, ce qui n'est pas faux en soi.

Mais comme il s'agit d'une propriété, le lien vers le véritable « GotoState » de la Vue n'est en rien connu, s'il n'y avait pas injection de la dépendance à un moment donné, la méthode serait « null » et tout appel se solderait par une exception…

Comment et quand la propriété est-elle initialisée ?

Le chemin n'est pas très direct, mais pour faire simple disons que le ViewModelRouter (BaseViewModel offre une propriété de même nom importée via MEF), dans ses méthodes GetNonSharedView et ActivateView, s'occupe de vérifier si le ViewModel a enregistré les états visuels de la Vue. S'il ne l'a pas déjà fait les états sont enregistrés dans le ViewModel et une action est placée dans la propriété GotoVisualState de ce dernier. L'action est un appel, via un dispatcher sur le thread de l'UI, au GotoState du VSM.

De fait, quand un ViewModel utilise la méthode GotoVisualState héritée de BaseViewModel, il ne fait qu'utiliser une propriété qui a été initialisée par l'activation ou la création de la Vue, cette activation ayant alors placé ce qu'on pourrait appeler un *pointeur* vers une expression Lambda assurant le changement d'état de la Vue.

Comme l'expression Lambda en question est construite de telle sorte à ce que le changement d'état visuel soit effectué via un dispatcher sur le thread de l'UI, le code du ViewModel n'a pas à se soucier de ce problème et peut déclencher GotoVisualState n'importe où, même si ce code est activé par un thread secondaire.

C'est une solution brillante. Simple, bien pensée, et efficace.

En utilisant GotoVisualState, un ViewModel peut synchroniser l'état visuel de sa ou ses Vues directement par des appels synchrones (l'utilisation d'un dispatcher introduit un léger décalage de temps pour atteindre le thread de l'UI mais cela ne pose généralement aucun problème à la différence des délais causés par une messagerie totalement autonome).

Exemple

Prenons un exemple très visuel (mais simple, je ne vais pas redessiner La Joconde☺) : Monsieur Smiley est un gars dont tout le monde sait qu'il peut prendre mille visages selon ses émotions.

Une commande de notre programme permettra d'indiquer si Mr Smiley est triste ou non.

Mr Smiley est un simple agencement de formes. La MainPage, en tant que UserControl, peut définir des états visuels. Elle en aura deux : le premier représentera un Mr Smiley tout sourire, l'autre un Mr Smiley triste.

Un simple Behavior Silverlight « GotoStateAction »connecté au Loaded du dernier objet chargé s'occupera de placer Mr Smiley à son état par défaut : rieur (et sans aucune transition). C'est ainsi qu'on le verra apparaître au chargement de l'application.

Une Checkbox sera reliée à la seule propriété exposée par le ViewModel « IsSad ».

Le setter de « IsSad » dans le ViewModel utilisera GotoVisualState hérité de BaseViewModel pour passer la Vue dans l'état « IsSadState » ou « IsHappyState » selon la valeur. Les transitions seront utilisées. Les noms des états sont ceux indiqués dans la Vue. Ces noms peuvent avoir été définis par des constantes et faire partie des conventions de l'application utilisées par une ou plusieurs Vues (par exemple « Caché » et « Visible », « EnErreur » et « Ok » …).

Pour compléter cette application d'une haute sophistication, un TextBlock sera ajouté, bindé à la propriété IsSad du ViewModel pour refléter l'état interne de ce dernier.

Visuellement cela donne :

Figure 87 - Mr Smiley est content !

Figure 88 - Mr Smiley est triste !

Tout l'intérêt de cette débauche d'effets spéciaux se situe dans le code du ViewModel :

```
private bool isSad;

public bool IsSad
{
   get { return isSad; }
   set
   {
      if (value == isSad) return;
      isSad = value;
      RaisePropertyChanged();
      GoToVisualState(isSad ? "IsSadState" : "IsHappyState", true);
   }
}
```

Bien entendu, le changement d'état visuel est immédiat (abstraction faite de l'éventuelle transition volontaire) et séquentiel : on pourrait placer une autre instruction derrière le changement d'état en étant sûr que le visuel est synchronisé. Toute la chaîne est gérée par des méthodes directes (des Actions, des expressions Lambda, le VSM de Silverlight).

Aucun message, même caché n'est utilisé. C'est comme si on violait MVVM en utilisant directement le VSM dans le ViewModel. Mais sans violer personne.

Comparé à la lourdeur de la solution qu'imposerait une gestion de messages, Jounce nous offre des moyens rapides, directes et synchrones de piloter le VSM d'une Vue dans le respect de MVVM.

Mais ce n'est qu'une des solutions offertes par Jounce. Il y a plus sophistiqué.

> *C'est là qu'on voit très bien que les problèmes posés par l'implémentation de MVVM proviennent du pattern lui-même, qui, pour l'heure, reste une collection de bonnes idées pas assez définies dans le détail. Ce sont les outils et toolkits qui, en proposant des interprétations plus ou moins brillantes de ces idées, transforment peu à peu MVVM en un véritable pattern.*
> *Les toolkits trop simples comme MVVM Light obligent à des contorsions qui font remettre en question la rigueur de MVVM, voire son utilité. Les toolkits trop complexes comme Prism ou Caliburn nous font arriver aux mêmes conclusions !*
> *Les outils intelligents comme Jounce transforment d'un seul coup ce qui était complexe en actions simples et parfaitement compréhensibles. Ici on ne se pose plus la question de savoir si MVVM est à réserver à certains types de projets ou non. C'est utilisable.*
> *Il y a bien un besoin de maturation de MVVM, enrichi par l'expérience de tous, pour arriver à un pattern clairement défini dans ses moindres détails et facilement utilisable. Grâce à des initiatives comme Jounce, MVVM se rapproche un peu plus chaque jour du vrai statut de pattern.*

Visual State Aggregator (VSA)

La solution que nous venons d'étudier semble correspondre au besoin. Pourquoi Jeremy a-t-il créé une seconde solution, plus sophistiquée comme je l'annonçais plus haut ?

Pour comprendre le besoin d'un autre procédé il faut se plonger dans une réflexion simple : En quoi un ViewModel devrait-il connaître les états visuels de sa Vue ?

Supposons une application qui se présenterait schématiquement comme suit :

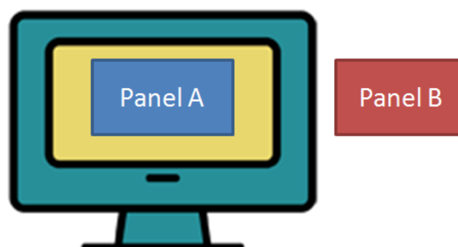


Figure 89 - VSA – Panel A (B désactivé)

Il y a deux panneaux, deux vues, ne partageant même pas le même ViewModel, Panel A et Panel B, ces Vues sont présentées par le shell. Dans la position de départ, Panel B n'est pas visible (représenté hors de l'écran).

On souhaite que lors d'un clic sur le Panel A, il se passe une transition visuelle et que Panel B apparaisse, disons de la façon suivante :

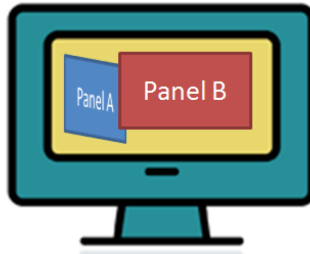


Figure 90 - VSA - Panel B activé

Il s'agit d'un pur effet visuel. Dans la réalité, le clic sur le Panel A aura peut-être pour effet réel de charger le détail d'une facture via un Service Web ou les Ria Services, détail présenté par le Panel B.

En quoi, finalement, la transition visuelle intéresse-t-elle le ViewModel de A ou de B ? En quoi le ViewModel du Panel A devrait-il être informé de l'existence même du Panel B dont il n'est pas même le ViewModel ? Doit-il, même au travers de conventions évoquées précédemment, se « mêler » de ce qui ne « le regarde pas » à savoir le visuel ? Est-ce le rôle du Shell que de gérer les transitions particulières en des Vues qui peuvent apparaitre et disparaître en dehors même de sa volonté ?

Est-ce normal que le code de l'application, tourné vers les données, leur validation, leur chargement, leur traitement soit obligé d'insérer dans des séquences souvent complexes des éléments purement visuels ?

La réponse à toutes ses questions s'impose rapidement : non, un ViewModel ne devrait pas avoir à connaître, même par le biais détourné de « conventions », quoi que ce soit de l'affichage et de ses effets. C'est le rôle de l'interface, du Designer, de définir le visuel. Le ViewModel doit rester concentré sur son job : servir et persister les données de la Vue, exécuter les ordres donnés par l'utilisateur.

C'est le résultat de cette réflexion qui a poussé Jeremy à chercher une solution totalement indépendante du code pour gérer la synchronisation des états visuels.

Et c'est une fois poser les bases de cette séparation absolue, voulue par la logique et le bon sens et non par un pattern qui serait trop tatillon, que Jeremy s'est mis en quête d'une autre voie que celle du GotoVisualState.

Cette autre voie est le VSA.

Le Visual State Aggregator est un service qui est capable de réponse à des évènements en modifiant l'état des Vues. C'est une sorte de messagerie, totalement différente de l'Aggregator que nous avons pu voir jusqu'ici (et qui lui gère une messagerie générique pour toute l'application). Le VSA est une messagerie dédiée au transport de messages particuliers se concluant par une modification des états visuels des Vues.

Le mécanisme se complète d'un Behavior qui permet directement dans le code Xaml à un Control de s'abonner à un message précis qui le fera passer dans un état particulier.

Un autre Behavior de nature Trigger permet à un Control, sur l'un de ses évènements, de transmettre un message VSA.

La communication qui se met ainsi en place se déroule uniquement dans le visuel entre éléments visuels. Exit le ViewModel de ce jeu de scène qui intéresse uniquement les acteurs visuels.

Un Control se dote d'états visuels. Une fois intégré dans un ensemble visuel dont il n'est qu'une partie il s'abonne à certains messages qui le feront changer d'état. D'autres Control déclencheront ces fameux messages. Tout cela en Xaml, sans que le code de l'application ne sache quoi que ce soit, pas même une « convention ».

Le VSA est l'étape de réflexion un niveau au-dessus de GotoVisualState, une élévation vers une séparation encore plus grande entre les intervenants, encore plus profonde entre la Vue et son ViewModel.

Le VSA est un Aggregator. C'est-à-dire à agrégateur. Vous êtes bien avancé ! ☺ Un agrégateur est quelque chose qui agrège… C'est-à-dire qui regroupe des éléments ayant, le plus souvent quelque chose en commun.

Le VSA agrège des VisualStateSubscription.

Un VSS représente un abonnement à un événement visuel. Il contient une référence (de type WeakReference) vers le Control concerné, le nom de l'évènement que ce Control veut écouter et qui déclenchera le changement d'état, le nom de l'état dans lequel basculer le Control lorsque l'évènement en question se produira (et une indication précisant si les transitions doivent ou non être utilisées).

Le VSA repose sur le VSM de Silverlight pour effectuer les changements d'état. Il est donc possible d'atteindre un état avec ou sans transition avec le VSA puisque le VSM le permet.

Une fois le VSS décrit, ne reste plus qu'à définir un agrégateur de VSS, le fameux VSA…

Il est de constitution très simple et ne fait que gérer une liste de VSS dont il sait faire le ménage automatiquement (lorsqu'un message est déclenché, puisque tous les VSS doivent être balayés pour le transmettre – ou non – il est facile de collecter tous les VSS dont la WeakReference vers le Control n'est plus valide).

Le VSA expose ainsi deux méthodes (la liste des VSS étant privée) : AddSubscription() pour ajouter un abonnement, PublishEvent() pour émettre un message.

La publication d'un message par le VSA se limite à balayer la liste des VSS et à appeler le changement d'état décrit si le nom de l'évènement enregistré est celui du message transmis. Vraiment très basique.

C'est avec les Behaviors ajoutés que le mécanisme complet prend forme. Grâce à VisualStateSubscriptionBehavior un Control peut s'abonner à des messages en précisant vers quel état basculer. Voici comment les panneaux de notre exemple (Panneaux A et B) peuvent s'enregistrer :

Dans le Panel A :

```
<UserControl>
  <Grid x:Name="LayoutRoot">
  <i:Interaction.Behaviors>
    <vsm:VisualStateSubscriptionBehavior
        EventName="ActivatePanelB" StateName="Background" UseTransitions="True"/>
    <vsm:VisualStateSubscriptionBehavior
       EventName="DeactivatePanelB" StateName="Foreground" UseTransitions="True"/>
  </i:Interaction.Behaviors>
  </Grid>
</UserControl>
```

On notera que c'est au niveau du LayouRoot que sont posés les Behaviors.

Dans le Panel B :

```
<UserControl>
  <Grid x:Name="LayoutRoot">
  <i:Interaction.Behaviors>
    <vsm:VisualStateSubscriptionBehavior
          EventName="ActivatePanelB" StateName="Onscreen" UseTransitions="True"/>
    <vsm:VisualStateSubscriptionBehavior
          EventName="DeactivatePanelB" StateName="Offscreen" UseTransitions="True"/>
  </i:Interaction.Behaviors>
  </Grid>
</UserControl>
```

Les deux panneaux répondent aux deux mêmes messages « ActivatePanelB » et « DeactivatePanelB ». Sauf que chaque panneau y répond de façon différente en indiquant le nom de ses états visuels propres qui doivent être activés en réponse.

Ainsi, lorsque le message ActivatePanelB sera transmis au VSA, celui-ci visitera les deux panneaux qui s'y sont abonnés, le Panel A ira dans l'état « Background » en utilisation les transitions, pendant que le Panel B se mettra dans l'état « OnScreen » lui aussi en utilisant les transitions.

On comprend facilement le mouvement inverse qui s'opère lorsque le message DeactivatePanelB est transmis : le panneau B se replie, sort de l'écran et le panneau A revient prendre sa place au centre de celui-ci (voir les petites illustrations plus haut).

Mais comment les messages vont-ils s'activer ?

C'est le rôle du second Behavior de permettre ce déclenchement. C'est un Trigger, donc il sait se connecter à tout type d'évènement du Control auquel il est attaché.

Dans le Panel A :

```
<Grid>
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="MouseLeftButtonUp">
      <vsm:VisualStateTrigger EventName="ActivatePanelB"/>
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Grid>
```

Dans le Panel B :

```
<Button Content="Close">
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="Click">
      <vsm:VisualStateTrigger EventName="DeactivatePanelB"/>
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Button>
```

Dans le Panel A on utilise le VisualStateTrigger sur la grille en attrapant son évènement MouseLeftButtonUp et en déclenchant le message ActivatePanelB.

Dans le Panel B c'est sur le bouton Close (fermer) et sur le Click de celui-ci que le Behavior est posé, déclenchant DeactivatePanelB.

Les deux messages activés correspondant à ceux utilisés par les panneaux pour s'abonner au VSA.

La boucle visuelle est bouclée : les panneaux vont s'ouvrir et se fermer selon un ballet visuel bien réglé, le tout uniquement avec du code Xaml, donc d'interface visuelle, sans que jamais les ViewModels ne soient au courant de ce qui se passe.

Les commandes qui seront gérées par les ViewModels (par exemple chargement par le panneau B de la liste des factures du client cliqué dans le panneau A) sont purement fonctionnelles. Les réponses à ces commandes le sont tout autant. Rien de visuel ne vient brouiller le code des ViewModels. On peut décider d'adapter, voire de changer demain totalement le visuel et les effets des Vues, seule la partie visuelle sera concernée, les ViewModels n'auront pas à être modifiés eux aussi.

C'est toute la beauté du VSA. Très simple (fort peu de code), mais donnant corps à une idée puissante.

Vous pourrez analyser le code de l'exemple VSMAggregator fourni avec l'article. Il propose une démonstration très proche de l'exemple développé ci-dessus mais avec de nombreuses subtilités qui méritent d'y consacrer un peu de temps.

CONCLUSION

Créer de vraies applications modernes réclame de mixer plusieurs frameworks et toolkits, plusieurs patterns. Arriver à créer un tout cohérent lorsqu'on associe MVVM, MEF et souvent des services Web ou WCF Ria Services peut devenir un jeu de construction hasardeux. Chaque technologie prise à part n'est pas forcément évidente à aborder, la documentation n'est pas toujours à la hauteur et le temps manque pour expérimenter. Mais lorsqu'on mélange tout cela ensemble, qu'on y ajoute la contrainte récente du Design, l'intervention d'un infographiste, l'espoir d'obtenir un logiciel maintenable et parfaitement fonctionnel peut relever du doux délire si on n'a pris le temps de se former correctement à chacune des problématiques et si on n'a pas su faire le bon choix des outils…

Ce Livre Blanc ne pourra pas vous transmettre mon vécu ni mon expérience, et encore moins vous guider vers la solution la mieux adaptée à vos besoins, ce n'est pas son objectif.

Mon métier m'amène à conseiller des clients aux besoins différents, aux contextes et contraintes parfois opposées. Conseiller le lecteur sans le connaître, lui et ses besoins, serait malhonnête. Pour cela j'offre mes services, en personne. Ce Livre Blanc vise un autre objectif : faciliter et accélérer la découverte de nouvelles approches méthodologiques et pratiques en vous faisant gagner du temps.

C’est un chemin sans fin, notre métier, pire que tous les autres, change et se réinvente en permanence. Mais comme dit le proverbe, un imbécile qui marche ira toujours plus loin qu’un savant qui reste assis sur le bord du chemin… Nous ne sommes pas des imbéciles, et en plus nous marchons ! Sans savoir où va réellement le chemin, il est vrai, mais nous irons certainement plus loin que ceux qui continuent à faire du PHP en utilisant le bloc-notes comme éditeur et qui, par force, ne savent pas ce qu’ils perdent…

Nous faisons parfois les mêmes logiciels, mais nous les faisons de façon plus intelligente.

Notre simple jubilation intellectuelle est en soi l’un des ingrédients les plus excitants de notre métier. Autant que de savoir qu’on bâtit des systèmes totalement virtuels qui sont utilisés par le monde réel et qui, au final, le modifie… D’où l’importance du Design, d’où la nécessité de nouvelles méthodes et patterns comme MVVM, d’où l’inéluctable besoin de disposer d’outils à la hauteur de la tâche qui nous incombe. D’où cette présentation de Jounce…

Ceux qui pratiquent notre métier sans cette indispensable quête de solutions toujours plus élaborées, plus satisfaisantes intellectuellement, sans cette essentielle composante impalpable qu’est la satisfaction de « bien » concevoir, ne font qu’aligner du code comme d’autres tranchent des steaks à longueur de journée dans l’arrière-boutique des boucheries de grandes surfaces. Je les plains.

Soyons heureux de découvrir chaque jour des visions différentes du monde, des façons nouvelles de solutionner les problèmes.

C’est le travail de l’ingénieur.

Celui qui fait le lien entre les théories savantes et l’homme de la rue en trouvant comment appliquer pratiquement les théories du premier pour changer la vie du second...

C’est une mission presque humanitaire. Sans les ingénieurs, de toute corporation, les théories resteraient dans les livres savants, et l’homme vivrait toujours comme au temps des cavernes.

Nous avons la chance de faire ce métier, en contrepartie il nous oblige à sans cesse garder un œil sur les théories et méthodes nouvelles et l’autre sur nos réalisations en cours. Voir le futur dans un design pattern et voir le présent dans un projet à boucler. Les pieds sur terre, la tête dans les nuages. C’est le paradoxe de l’ingénieur qui aime son métier et le fait bien. Comme quoi un paradoxe n’est pas qu’un casse-tête, cela peut aussi être source de cohérence et de joie. Paradoxal ? ! ☺

PS : Ce livre blanc a réclamé des journées entières de travail, je l'ai lu et relu, mon infographiste préférée l'a relu aussi (on saluera l'effort sur un sujet si technique !) mais corriger plus de 100 pages est une tâche délicate. On atteint une taille qui est plus celle de l'édition que de l'entrée de blog ! Or, dans l'édition, que je connais bien pour avoir produit trois livres, un document de cette taille passe aussi entre les mains d'un correcteur professionnel. Plusieurs passes sont souvent nécessaires, pour remanier le texte aussi. Et malgré tout il reste des coquilles. Le présent document, c'est prévisible, est encore imparfait malgré ma bonne volonté et mon attention. Un livre coûte dans les 55 euros, ce livre blanc est gratuit. Le lecteur saura, j'en suis sûr, me pardonner qu'il ne bénéficie pas de toute la chaîne de production d'un couteux livre du commerce…

Prism pour WinRT– Partie 1

Je l'ai déjà évoqué ici plusieurs fois, l'équipe Prism vient de publier les premiers rushs de "Prism pour WinRT" sur Codeplex, une spécification à laquelle j'ai participé en tant que membre du *Developer Guidance Advisory Council*. En toute logique je me devais vous présenter cette nouvelle guidance…

PRISM OR NOT PRISM ?

Autant le dire immédiatement, Prism for the Windows Runtime - Prism pour WinRT - (appelé pendant son développement "Kona RI" pour Référence d'Implémentation "Kona") n'a que très peu de choses à voir avec Prism pour WPF ou Silverlight.
L'équipe n'est certes pas tout à fait la même mais la différence ne vient pas de là. Elle tient à la nature même de WinRT qui a une philosophie radicalement différente de celle de WPF. Le mode fullscreen, l'absence de fenêtres secondaires, le tactile, le mode sandboxé, et les limitations de XAML sous WinRT font partie de ces choses qui interdisent de porter la même guidance de WPF à WinRT.

Il ne faut donc pas chercher dans cette version de Prism une quelconque analogie avec Prism pour WPF/Silverlight, il n'y en a pas, tant au niveau code qu'au niveau de la guidance elle-même. Seul l'esprit est le même : promouvoir les meilleures méthodes pour produire des applications fiables.

WINRT POUR LE LOB ?

C'est "ze question". La nature même des besoins d'une application LOB en entreprise s'adapte en fait assez mal aux contraintes de WinRT et au mode fullscreen. C'est un problème de support et non de technologie. Sur tablette WinRT s'avère au contraire bien meilleur que WPF, d'où mon incompréhension devant le peu de courage à pousser Surface RT et l'empressement de Microsoft à sortir une Surface Pro dont la seule différence réelle

est de rétablir le bureau classique. J'adore Surface RT et elle n'aurait jamais du être "lâchée" aussi vite. Le bureau classique n'a pour moi aucun sens ni intérêt pratique sur Surface car les applications classiques ne sont pas conçues pour de si petits écrans. Mais bon, les choses étant ainsi, faisons avec le monde tel qu'il est et non tel qu'on voudrait qu'il soit… Et tentons de répondre plus en détail à la question.

L'implémentation de référence (la "Kona RI" créée au départ) a fixé un scénario qui se veut orienté "business", comme toute les implémentations de Prism. Il s'agit donc bien de développer des applications LOB ou dans cet "esprit" en tout cas, malgré les limitations déjà évoquées de WinRT. Or, durant les premières phases de l'étude il est très vite apparu que quoi qu'on fasse WinRT ne se pliait guère aux exigences habituelles des applications LOB… L'équipe de Prism n'a pas baissé les bras mais a un peu baissé ses prétentions sans trop l'avouer tout en essayant de trouver des solutions originales. De fait la RI ressemble plus à une application Web de type site marchand qu'à une "vraie" application LOB tel qu'on l'entend en entreprise sur un PC de bureau. Mais le résultat est assez convainquant et prouve que WinRT peut, dans un cadre aux limites qui doivent être comprises, servir à écrire des applications pour le monde de l'entreprise. Vous noterez le bémol dans ma façon de m'exprimer : je ne parle plus d'applications LOB mais d'applications pour le "monde de l'entreprise". C'est une nuance à bien saisir.

On ne verra certainement jamais de WinRT dans les salles de marché par exemple. Fenêtres multiples ouvertes simultanément, écrans multiples où s'étalent des modules de plusieurs applications, tout cela est impossible avec la disparation du bureau classique et le mode fullscreen de WinRT sous Windows 8.

Dès lors soyons francs WinRT sous sa forme actuelle n'est pas une panacée pour les applis LOB car cette nouvelle technologie n'est pas "complète" dans le sens où elle n'est pas en mesure de remplacer totalement et immédiatement l'existant tel que WPF ou même Windows Forms dans les circonstances précises du LOB. Toutefois des pans entiers d'applications peuvent fort bien trouver sur les tablettes un cadre parfait pour s'épanouir. On peut penser à des présentations de catalogue, de la prise de commande, de la saisie de devis autant que des tableaux de bord avec graphiques. Cela fait tout de même beaucoup de choses !

WinRT souffre de problèmes de jeunesse (comme les manques dans XAML tels les Behaviors), mais aussi d'un problème d'UX fondamental, sur PC de bureau le fullscreen et la suppression de la métaphore du bureau ne semble pas une approche pertinente. C'est ainsi plutôt sur tablettes qu'on s'essayera aux applications d'entreprise sous WinRT. Mais, avantage de la plateforme, ces mêmes applications pour tablettes tourneront sur PC aussi. Il faut donc le voir dans ce sens, et l'inconvénient se transforme en avantage !

Pour comprendre le problème d'UX sur PC il faut aller à la genèse de WinRT qui fait partie d'un plan particulier visant à conquérir le marché grand public pour sortir Microsoft de ses déboires sur le marché des unités mobiles. L'idée d'une plateforme unique pour tous les form factors est géniale. Mais WinRT a d'abord été conçu pour le mobile, Windows Phone mais aussi et surtout Surface RT. Et cela se sent. L'UX de Windows 8 sur PC n'est pas à la hauteur de son côté novateur, c'est d'ailleurs pourquoi Microsoft en fait l'aveu en annonçant aussi rapidement de grands changements dans Windows 8.1 sous la forme d'un "je vous ai compris!" et propose d'ailleurs d'offrir cette mise à jour. Une si grande générosité n'a de sens que dans l'aveu d'une "faute" qu'on cherche à effacer.

Ainsi l'idée un peu saugrenue de porter à l'identique le WinRT tablette sur les PC de bureau s'est imposée dans ce schéma d'un OS pour tous les form factors sans que vraiment on s'interroge sur le bienfondé de cette démarche. Un seul OS, une seule plateforme de développement, et un seul look, tout cela semblait tellement parfait. Mais cela implique aussi, hélas, une seule UX. *Or, ce qui est bien conçu pour une tablette ne l'est pas forcément pour un PC de bureau.*
WinRT est une plateforme que j'aime et que je trouve intéressante à plus d'un titre. Ce qui ne va pas, au moins pour l'instant, c'est de l'avoir portée "tel quel" de la tablette au PC et de l'imposer comme "seul et unique futur" de Windows alors même que WinRT n'est tout simplement pas capable d'assumer totalement ce rôle à l'heure actuelle. WPF est négligé, faisant partie d'un passé révolu, alors même qu'il reste le seul et unique moyen de faire des applications modernes sous Windows s'affranchissant des limitations de WinRT (principalement la sandbox et le fullscreen mais aussi les machines équipées de Windows 7 qui représentent un parc énorme).

Membre du DGAC, collaborateur sur le dernier livre de Eyrolles présentant le développement WinRT, auteur de nombreux billets ici sur le sujet, je me considère comme étant un bon spécialiste de la question. Et je dois avouer que sur PC de bureau WinRT n'est pas tout à fait utilisable pour du LOB. Au moins "un certain type d'applications LOB". Comme je le disais plus haut il reste néanmoins de nombreuses applications du "monde de l'entreprise" qui peuvent parfaitement être écrites sous WinRT.
Alors à la question "WinRT est-il utilisable pour des applis LOB ?" je répondrais un peu la normande : oui et non… Non sur PC de bureau pour du LOB traditionnel, mais oui sur tablette pour certaines applications du "monde de l'entreprise".

*Quoi qu'il en soit, au moins sur tablettes, **WinRT offre une bien meilleure UX** que les applications classiques qui ne sont absolument pas conçues pour des petits écrans de 10" voire moins. Et techniquement, WinRT surtout avec C#/XAML est une plateforme de très loin supérieure à celles proposer par Apple ou Google.*

DANS CE CONTEXTE QUELLE EST LA PLACE ET L'INTERET DE PRISM ?

Comme je viens de l'expliquer WinRT n'est peut-être pas une panacée et n'est certainement pas en capacité de pouvoir remplacer totalement des technologies comme WPF en bureau classique. Mais Il ne faut pas voir les choses avec dogmatisme, ni même trop écouter le langage commercial de Microsoft qui, par force, pousse la nouvelle technologie pour d'évidentes raisons ce qui interdit toute nuance dans le discours. Dans la réalité le bureau classique est loin de disparaitre non pas parce que je le "crois" comme une sorte de "conviction" personnelle, tout simplement parce que des milliers de logiciels professionnels ne peuvent absolument pas être portés sous WinRT en raison ne serait-ce que de la sandbox ou du mode fullscreen.

Cette situation est peut-être temporaire, mais elle va perdurer des années encore, peut-être toujours. Le jour où Adobe annoncera une version WinRT de Photoshop et de Illustrator ou Première, le jour où Cubase ou Ableton Live seront proposés en WinRT, ce jour là seulement on pourra dire adieu au bureau classique. Mais là c'est une conviction personnelle, je n'y crois pas. Une bonne raison à cela : si les développeurs indépendants et les petits éditeurs peuvent voir un intérêt d'être distribués sur le market place, les gros éditeurs déjà connus et faisant eux-mêmes leur CA ne sont absolument pas prêts à céder un pourcentage sur leurs ventes à Microsoft, leurs actionnaires ne le permettraient pas. Il y a donc une réalité économique et non technique derrière le maintien pour de longues années du bureau classique…

A l'heure actuelle au moins, il faut donc voir WinRT comme une *nouvelle opportunité de concevoir des logiciels "autrement"* **ne remplaçant rien du tout mais s'ajoutant au contraire à la panoplie déjà disponible** comme WPF.
Il y aura des applications (ou des morceaux d'applications) parfaitement à l'aise sous WinRT qui leur offrira même un écrin leur donnant encore plus de valeur, et il y aura des applications (ou des morceaux d'applications) qui n'auront de sens que sur PC de Bureau en multifenêtrage, donc sous WPF.

C'est plutôt vers une telle **cohabitation intelligente** que nous nous dirigeons. Une fois la langue, un peu de bois, de la phase de lancement très commerciale de Windows 8 / WinRT passée, Microsoft adoptera très certainement un discours plus nuancé.
D'ailleurs, dans le dernier livre paru chez Eyrolles sur le développement WinRT, livre auquel j'ai collaboré et écrit par des employés de Microsoft France, on voit déjà très nettement l'infléchissement du discours "tout WinRT". Les solutions classiques sous WPF ne sont plus présentées comme des vieilleries hasbeen mais comme des alternatives parfois mieux taillées pour certains types de logiciels. Pour qui sait lire entre les lignes ce changement sous la plume de gens très officiellement liés à Microsoft est un signe qui ne trompe pas.

Une fois la place et l'avenir de WinRT et du bureau classique clarifiés on comprend qu'il existe et existera des créneaux tout à fait viables pour certains types d'applications d'entreprise sous WinRT.

Dès lors disposer d'une guidance de qualité comme Prism se justifie totalement et prend tout sens.

PRISM POUR WINRT

Partant d'un scénario orienté business, l'équipe de Prism a débuté une implémentation dite de référence, la fameuse "Kona RI". L'esprit étant de développer une application telle qu'elle pourrait l'être au sein d'une entreprise, avec les besoins d'un tel contexte, sans passer par le market place ni être conçue pour le grand public (type jeu ou réseau social).

En suivant le scénario et en implémentant *AdventureWorks Shopper RI* de nombreux problèmes propres au cadre WinRT ont du être réglés. A chaque étape l'équipe de Prism avec l'aide des membres du DGAC (Developer Guidance Advisory Council) s'est demandée comment généraliser le code, comment le rendre réutilisable. Le code mais aussi les méthodes utilisées qui deviendraient des guidances.
En menant ce travail de bout en bout, l'application de référence a été souvent modifiée, adaptée. Et dans les dernières étapes elle a été refactorée pour en ressortir tout le code réutilisable sous la forme d'une bibliothèque "Microsoft.Practice.Prism.StoreApps" et d'une série de recommandations écrites, le tout formant Prism pour WinRT.

Le lien entre Prism et Prism pour WinRT est donc purement conceptuel. Aucune partie du code de Prism n'a été réutilisée. Prism pour WinRT a été entièrement créé pour répondre aux problématiques WinRT. Certains trouveront dommage que le résultat soit justement si éloigné de Prism pour WPF et qu'il n'existe aucune compatibilité entre ces deux versions de Prism, mais c'est un choix assumé en raison des différences trop importantes entre les deux mondes.

L'esprit de Prism, le concept, est justement de formuler des guidances en adéquation avec la plateforme visée et non pas d'adapter une méthodologie unique à plusieurs plateformes… Prism est un concept, celui des meilleurs moyens pour créer des applications fiables sur une plateforme donnée. Cela implique dans les faits des guidances et des implémentations totalement différentes si les plateformes le sont aussi. Et c'est le cas.

On notera, en écho à mes propos d'introduction, que la RI en question s'appelle AdventureWorks Shopper RI, c'est à dire, l'implémentation de référence du consommateur/client de AdventureWorks. Autrement dit une application native de vente destinée aux clients (fictifs) de la société (fictive) AdventureWorks.

AdventureWorks est l'application RI de Prism pour WPF et Silverlight. Il s'agit d'une société fictive permettant de démontrer plusieurs aspects de la programmation d'applications LOB.

La spécification Prism pour WinRT n'est pas "AdventureWorks" mais la partie "site marchand" de AdventureWorks…

Il ne s'agit donc plus ici de prétendre développer une application LOB classique, mais bien d'utiliser au mieux le support tablette pour développer une partie bien spéciale de l'application. On reste dans le "monde de l'entreprise" mais on n'est plus dans le LOB proprement dit…

On pourrait reprocher d'ailleurs à cette RI d'être une application "externe" destinée aux clients de l'entreprise et de n'être justement pas une application utilisable _dans_ l'entreprise. Mais après tout c'est aussi un besoin de l'entreprise que de séduire ses clients et les fidéliser…

La question finale que soulève cette approche c'est qu'en regardant AdventureWorks Shopper RI on se demande si dans la réalité de l'entreprise on n'aurait pas fait un meilleur choix en optant pour une implémentation sous ASP.NET MVC avec HTML pour une telle application de vente en ligne… Cela aurait été mon conseil si un client m'avait posé la question. Une telle application aurait eu l'avantage de fonctionner aussi depuis une tablette Apple ou Android sans aucune modification. L'avantage du natif ne semblant pas décisif dans ce cas précis qui n'est qu'un site de vente en ligne.

Comme vous le voyez il m'est très difficile de vous donner un conseil aussi clair que d'habitude, je ne cacherais pas mes propres doutes et questionnements sur la place de WinRT en entreprise. Mais finalement mon conseil tient aussi dans ce doute que je partage avec vous…

Après tout, le choix de ce scénario par l'équipe Prism n'était peut-être pas le bon car il prête le flanc aux attaques sur l'intérêt même de WinRT, c'est un problème que j'ai soulevé dès les premières réunions du DGAC, mais ce n'est qu'une RI, un simple exemple d'implémentation qui devait rester simple à comprendre par tous sans imposer un lourd contexte plus réaliste. Nul doute qu'en situation vous aurez milles autres idées de développement plus proches des vrais besoins de votre entreprise.

CONCLUSION

Je vais tenter de conserver un format compatible avec le support "blog" en évitant de faire un papier à rallonge… C'est donc la fin de cette première partie qui présente à la fois Prism pour WinRT et les raisons pour lesquelles je vous présente cette guidance sans masquer les

questions que l'utilisation de WinRT pose en entreprise, ni les opportunités qu'elle ouvre aussi.

J'agis ici en éclaireur, je montre un chemin, à vous de le prendre ou non. Mais pour vous faire votre propre idée le mieux c'est encore de savoir de quoi le chemin sera fait. Et cette étude de Prism est là pour vous aider à faire les meilleurs choix technologiques.

Pas de dogmatisme donc, ni de propagande, une simple présentation d'un travail intelligent pour vous permettre de l'utiliser au mieux ou de savoir pourquoi vous ne l'utiliserez pas. C'est open ! (mais si vous devez faire du LOB en WinRT, ne pas utiliser Prism pour WinRT faute de le connaitre serait vraiment dommage).

Alors pour la suite qui montrera Prism pour WinRT par l'exemple…

Prism pour WinRT– Partie 2 – Premier exemple de code

je vous ai présenté Prism pour WinRT le mois dernier (partie 1), il est temps d'aller un peu plus loin pour comprendre cette nouvelle approche proposée par Patterns & Practices de Microsoft.

PRISM POUR LE WINDOWS RUNTIME

Sans faire de redite et juste pour replacer le sujet dans son contexte je rappellerais ici que PRISM for Windows Runtime (ou Prism WinRT) est une guideline de programmation sous WinRT concoctée par le groupe Patterns & Pratices de Microsoft. L'équipe qui a créé PRISM pour WinRT n'est pas exactement la même que celle qui a conçu PRISM pour WPF/Silverlight mais elle a travaillé dans le même esprit : créer la meilleure guidance pour un environnement précis, ici WinRT.

Il n'y a aucune compatibilité entre ces deux PRISM, ni aucune volonté de les faire se rejoindre. WPF/SL sous .NET réclament une approche propre, WinRT aussi. Malgré ses ressemblances (si on utilise le couple C#/XAML ce qui n'est qu'une option possible) WinRT est fondamentalement différent de .NET. PRISM pour WinRT l'est donc aussi.

A QUOI SERT PRISM ?

Prism est une boîte à outils, un ensemble de guidelines destiné à facilité la création d'applications composites (couplage faible entre les parties, extensibilité…) respectant au mieux les contraintes de la plateforme.

Sous WPF/SL nous disposons de nombreuses bibliothèques ayant un but proche notamment dans la mouvance MVVM : Mvvm Light, Caliburn, Jounce, Prism 4, etc.

Pour WinRT certaines de ces bibliothèques sont aussi disponibles comme Mvvm Light ou d’autres pour aborder le cross-plateforme comme MvvmCross.

Prism WinRT n’est donc qu’une bibliothèque parmi d’autres offrant sa propre logique et s’adaptant mieux à certaines applications qu’à d’autres.

Sous WPF/SL, je n’ai rencontré Prism que chez des clients ayant des équipes de haut niveau et fortement soudées. PRISM 4 est en effet plus difficile semble-t-il à comprendre que d’autres bibliothèques comme Mvvm Light et cela réclame des équipes stables où il ne faut pas réexpliquer régulièrement tout à des membres qui entrent dans l’équipe et en ressortent rapidement.

De plus PRISM WPF/SL a mis du temps à accepter l’idée de MVVM dont le terme n’est apparu que tardivement dans la documentation et plutôt comme une sorte “d’obligation” que comme véritable but à atteindre. Prism 4 propose une approche différente même si elle revient au même (séparation des modules et de l’UI). Toutefois elle s’avère moins pratique et moins efficace de les bibliothèques directement MVVM, notamment lorsque ces dernières comme Mvvm Light assurent la “blendabilité” simplifiant grandement la mise en place du visuel.

Ainsi Prism WPF/SL a toujours été hautement estimé pour sa qualité mais n’a que rarement été utilisé par des petites équipes et encore moins des développeurs seuls.

Prism pour WinRT a été conçu différemment. D’abord l’évidence même de MVVM s’est imposée à tous depuis et l’équipe a pris ce paradigme comme base. Ensuite, l’environnement WinRT, par les contraintes nouvelles qu’il impose, a amené l’équipe de Prism, aidée par les membres du Developer Guidance Advisory Council (dont je fais partie, notamment sur ce projet Prism), à trouver des solutions originales adaptées au contexte.

Les buts visés par Prism WinRT sont les mêmes que ceux de Prism 4, seule la façon d’y arriver est différente :

- La modularité : Consiste à définir et charger dynamiquement des fonctionnalités faiblement couplés dans une instance d'application en cours d'exécution.
- La composition de l’UI : Consiste à brancher des vues dans des conteneurs parents dans un mode à couplage lâche où le parent et l'enfant n'ont pas besoin de se connaître explicitement (pas de références d'objet directes).
- Les communications : Consiste à gérer notamment les commandes dans un mode à couplage lâche et en l’utilisation d’un pattern pub / sub pour les événements entre les composants de l’application.

- La navigation : Consiste à gérer les changements de vue dans un conteneur de lorsque l'utilisateur interagit avec l'application sans jamais que les vues et leur parent n'aient besoin de se connaître.
- La Structure : Consiste à offrir le support du pattern Model-View-ViewModel (MVVM), ainsi qu'un conteneur d'injection de dépendances, et d'autres patterns d'implémentation pour aider à mettre en place la solution avec la meilleure séparation possible entre les tiers la constituant.

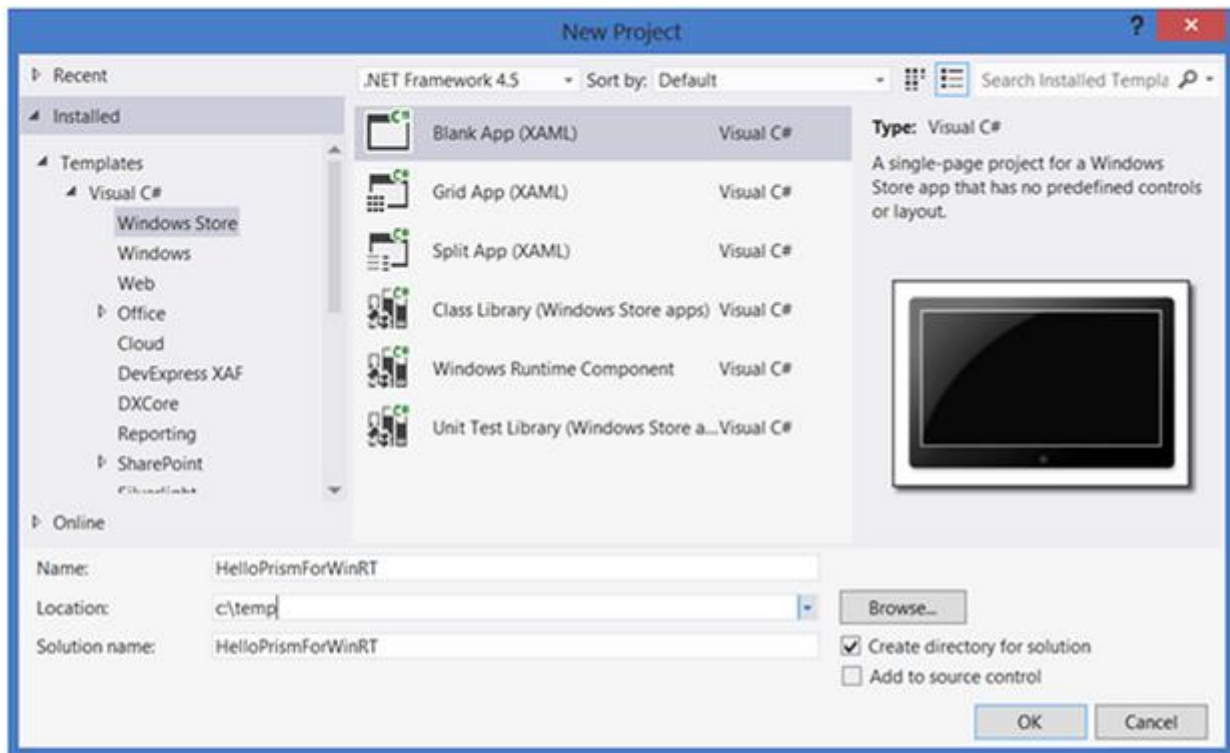
Une chose importante à comprendre au sujet de Prism est que ce n'est pas un cadre “tout-ou-rien”. Vous pouvez utiliser une ou plusieurs de ses caractéristiques et ignorer les autres parties si elles n'ont pas de sens pour votre application ou vos besoins. En plus de ces principales fonctionnalités Prism offre beaucoup de petites petites classes utilitaires qui peuvent être utilisées seules en dehors du contexte Prism. Le code source étant disponible il n'est pas même indispensable d'intégrer les bibliothèques Prism en entier pour juste emprunter telle ou telle partie qui les constituent.

CREER UNE APPLICATION WINRT AVEC PRISM

Le meilleur moyen de comprendre les services offerts par Prism pour WinRT est de bâtir un exemple. Nous nous limiterons dans un premier temps aux bases essentielles. Prism est un “gros morceau” même sous WinRT. Bien entendu il existe des modèles de solution qui permettent d'éviter une partie de la mise en place que nous allons voir maintenant, mais tout l'intérêt est justement de le faire “à la main” pour comprendre le fonctionnement de Prism…

Créer le projet

Pour commencer, ouvrez Visual Studio et créez un nouveau projet Windows Store :



Utilisez le modèle “Blank” qui met en place une structure basique sans fioriture.

Ajouter des Pages et leurs ViewModels

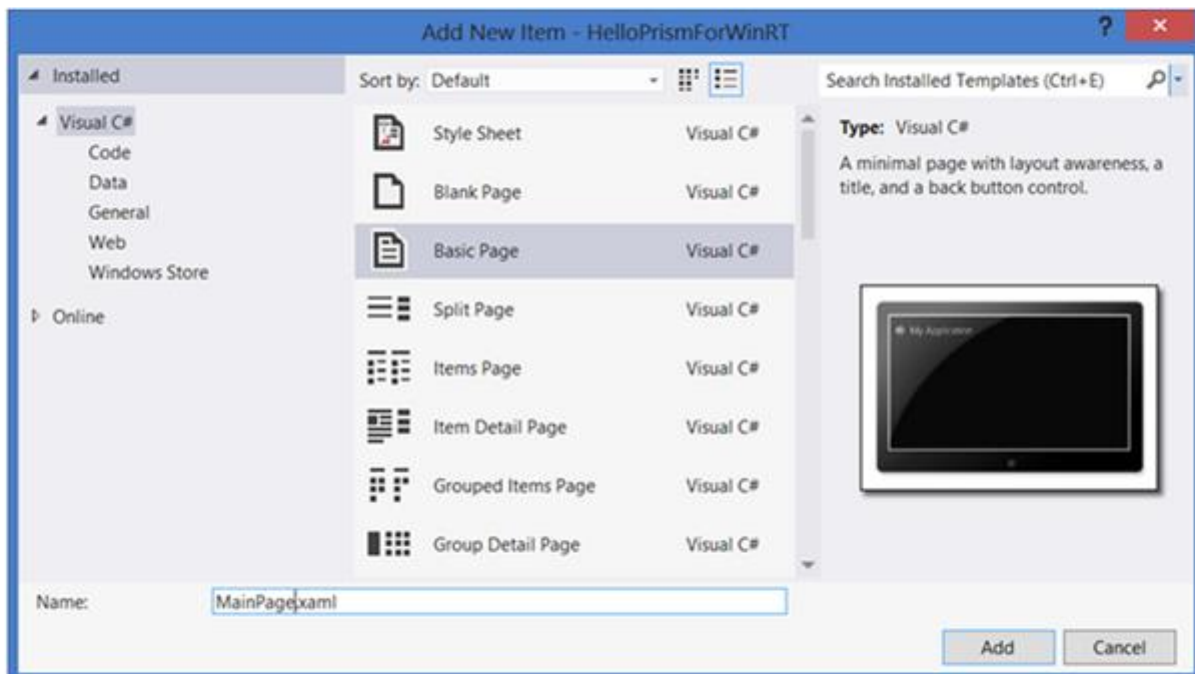
Dans une application Windows Store, vous concevez votre application autour de la notion de Page. L'utilisateur navigue de page en page pour accéder à toutes les fonctionnalités de votre application. Vos pages sont votre niveau de vue le plus élevé suivant une logique MVVM, et vous devrez donc mettre en place les ViewModels correspondants. Vous pourriez avoir des vues enfant de certaines pages - par exemple on peut concevoir l’idée d’un ContentControl dans une page dans lequel vous changeriez les vues enfant. Mais ce genre de mise en page est beaucoup moins fréquente que sous WPF et Silverlight en raison des directives de style de l'interface Modern UI. Avec WinRT vous devez concevoir des écrans moins denses que ce qu’il se pratique en LOB classique sous WPF/SL. Dans le modèle d’UX Modern UI les fonctionnalités de l'application sont réparties sur plusieurs pages parmi lesquelles l'utilisateur navigue au lieu de tout faire dans une grande vue compliquée.

Il s’agit là de la façon “positive” de présenter les restrictions bien réelles de WinRT. Pour dire la vérité et comme nous l’avons vu dans la Partie 1 Modern UI ne s’adapte qu’à un certain type d’application d’entreprise. Celles qui peuvent se satisfaire de pages pauvres en contenu et d’une navigation incessante entre les pages. La plupart des applications LOB ne se plient pas à ces contraintes et ce n’est pas par manque de créativité ou d’imagination mais parce que le fonctionnel l’oblige. L’efficacité veut en général au contraire que l’utilisateur puisse accéder au maximum de choses sans trop avoir à naviguer.

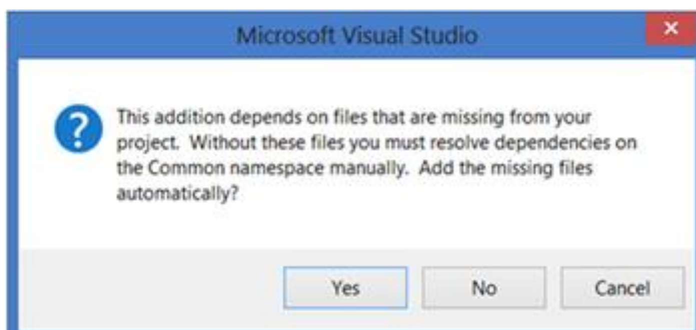
Je ne redirais pas ce que j’ai argumenté dans la Partie 1 mais il faut se rappeler que si Modern UI est parfaitement utilisable en entreprise cela ne peut s’entendre que pour certains types d’applications, vouloir tout programmer en WinRT serait une erreur. Pour simplifier je dirais que tout ce qui était éventuellement réalisé sous la forme de sites Web (Intranet par exemple) peut être facilement porté en WinRT car les contraintes de mise en page et de navigation sont assez semblables au final. D’ailleurs même l’exemple fourni par l’équipe Prism avec les guidances ressemble au final à un gros site marchand, mais pas à une application LOB, et si WinRT peut se programmer en HTML 5 ce n’est pas rien non plus.

En revanche tout ce qui réclame un travail long de l’utilisateur, avec concentration ou des tâches répétitives ne pourra que créer une UX médiocre avec le modèle Modern UI : l’utilisateur ne comprendra pas pourquoi il doit sans cesse naviguer alors qu’il y a plein de place à l’écran non (mal ?) utilisée. N’oubliez pas non plus qu’en entreprise les machines Windows 8 et futures versions ne seront pas majoritairement équipées d’écrans tactiles. C’est toute l’ambigüité de WinRT qui a été jusqu’à ce jour présenté comme le seul et unique remplaçant de toutes les autres technologies Microsoft, notamment de WPF et SL, alors qu’en pratique il faut le voir comme une possibilité supplémentaire pour des parties spécifiques d’applications uniquement et avec le gain d’un exécutable tournant sur PC et Surface.
Les avantages sont donc bien réels et dans ce cadre WinRT est une superbe plateforme qui aurait pu être valorisée mille fois mieux, c’est à vouloir faire la grenouille plus grosse que le bœuf qu’on la fait éclater... Comme quoi le savoir-vendre est parfois plus important que la technologie !
On notera que par souci de simplification Prism WinRT utilise plutôt des conventions que des fichiers de configuration. Il y a une classe ViewModelLocator qui rend l’accès aux VM très simple, un peu comme sous Mvvm Light. La blendabilité s’en trouve améliorée par rapport à Prism WPF/SL. Ce ViewModelLocator utilise d’ailleurs l’une des conventions évoquées plus haut, elle suppose que vous stockez vos pages dans un sous-dossier du projet intitulé Views et vos ViewModels dans un sous-dossier nommé - vous l'aurez deviné - ViewModels. Si cette convention ne vous va pas dans un projet donné il existe des hooks pour placer les fichiers où on veut en le précisant. Mais une grande partie du code réutilisable de Prism est conçu autour d’un modèle commun de guidelines et si vous suivez ces dernières vous aurez moins de travail à fournir.

Donc, pour commencer il faut structurer l’application avec les sous-répertoires Views et ViewModels. Ensuite supprimez le MainPage.xaml qui a été ajouté à la racine du projet par Visual Studio. Puis faites un clic droit sur le dossier Views et sélectionnez Add>New item dans le menu contextuel. Dans la catégorie Windows Store, sélectionnez le modèle de page "Basic" et nommer-le "MainPage".



Lorsque vous cliquez sur Ajouter, vous serez invité à ajouter des fichiers communs aux applications Windows Store à votre projet. Acceptez la proposition de VS, même si ici nous n'utiliserons pas la plupart de ces classes mais les équivalents de Prism qui correspondent mieux avec le modèle MVVM que nous allons utiliser.



Allez dans le code XAML de MainPage et changer "My Application" en "Hello Prism" dans la section des ressources (ou mettez ce qu'il vous plaira, cela n'a aucune incidence).

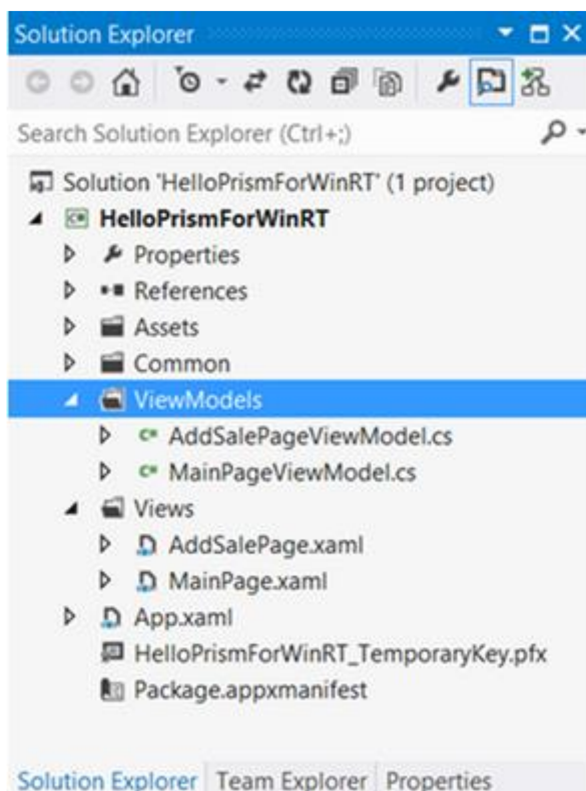
Répétez Add>new Item pour ajouter une deuxième page Basic nommé AddSalePage dans le dossier Views.

Maintenant vous pouvez voir que nous avons des ressources dupliquées dans les différentes pages. Le mieux étant de les supprimer de ces dernières pour les ajouter à App.Xaml et ainsi les centraliser :

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="Common/StandardStyles.xaml" />
        </ResourceDictionary.MergedDictionaries>
        <x:String x:Key="AppName">Hello Prism</x:String>

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

Ensuite, ajoutez deux classes dans le dossier ViewModels: MainPageViewModel et AddSalePageViewModel. À ce stade, votre projet doit ressembler à ceci dans l'Explorateur de solutions :



Ajouter Prism et les références à la DLL

La version finale de Prism pour WinRT peut être téléchargée ici : http://code.msdn.microsoft.com/windowsapps/Prism-for-the-Windows-86b8fb72

Pour ajouter Microsoft.Practices.Prism.StoreApps à votre application il suffit d’ajouter à la solution le projet “Prism.StoreApps” se trouvant dans le zip (que vous aurez décompressé en totalité quelque part sur l’un de vos disques).

Il est fort probable (je n’ai pas vérifié) qu’au moment où j’écris ces lignes le package Nuget soit disponible. Dans ce dernier cas il sera plus simple d’utiliser l’installation automatique par ce biais que d’ajouter les sources du projet Prism à votre solution.

Une fois Prism installé il faut bien entendu ajouter la référence à Microsoft.Practices.Prism.StoreApps évoquée plus haut (sauf si l'installation passe par un package Nuget qui est censé le faire automatiquement).

Modifier la classe de base de l'application

La classe application définit dans App.Xaml.cs dérive de la classe du framework WinRT et contient du code pour gérer le cycle de vie de l'application, la création de la première vue etc… Tout cela est encapsulé dans une classe spécifique de Prism.

Il suffit de substituer la classe mère de App par MvvmAppBase. Cette dernière va fournir des comportements par défaut tout en offrant des hooks dans le cas où vous souhaiteriez reprendre la main sur certains des aspects pris en charge automatiquement.

On profite de cette substitution pour faire un grand coup de ménage, la quasi totalité du code dans App.Xaml.cs ne sert plus rien. On ne garde au final que le constructeur (avec InitializeComponent) ainsi que la méthode OnLaunchApplication dont on remplace le code présent par un appel au service de navigation de Prism :

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Practices.Prism.StoreApps;
using Windows.ApplicationModel.Activation;

namespace HelloPrismForWinRT
{
    sealed partial class App : MvvmAppBase
    {
        public App()
        {
            this.InitializeComponent();
        }

        protected override void OnLaunchApplication(LaunchActivatedEventArgs args)
        {
            NavigationService.Navigate("Main", null);
        }
    }
}
```

Malgré la grande simplicité du code ci-dessus il y a un certain nombre de choses à prendre en considération car tout ne sera pas forcément si simple tout le temps.

Par exemple la substitution du code de OnLaunchApplication est vraiment minimaliste. Pour un "démarrage à froid" de votre application cela sera parfait. Mais vous devez garder à l'esprit que votre application Windows Store peut être lancée de plein façons différentes y

compris comme une cible pour une opération de charme de recherche, une tuile secondaire sur l'écran de démarrage, ou encore d'autres voies ! Dans ces cas, vous pourriez avoir besoin de naviguer vers une page différente de "Main" sur la base des arguments de lancement qui entrent dans la méthode de lancement surchargée. Bien entendu MvvmAppBase a d'autres méthodes à surcharger pour manipuler recherche ou la gestion des paramètres. Mais pour l'instant dans notre exemple aller à la MainPage est un choix simple dont on se contentera pour ne pas rendre les choses trop compliquées.

Vous aurez certainement noté que nous naviguons sur «Main», et non sur MainPage, le type. Il s'agit d'une décision de conception : les demandes de navigation ne doivent pas être couplées à la nature spécifique de la vue ou de la page. Gardez à l'esprit que si vous voulez concevoir des applications WinRT efficaces et utilisables facilement vous devrez d'abord écrire des storyboards ou des wireframes pour définir avec précision quelles sont les pages, leur rôle et comment l'utilisation doit ou peut naviguer de l'une à l'autre. Vous devrez à ce moment là définir des noms logiques pour les vues même si vous n'avez encore aucune idée des noms définitifs des classes qui seront implémentées. La convention utilisée pour la navigation dans Prism par défaut est que vous accédez à un nom de vue logique, et elle suppose aussi que le nom du type sera nom logique de la vue + "Page".

Dans le cas où vous souhaiteriez adopter une autre convention il existe un hook en surchargeant la méthode GetPageNameToTypeResolver de l'application. Cette méthode retourne un Func<string,Type> de fait vous êtes supposé retourner un délégué qui prend en paramètre une chaîne et renvoie un type. Par exemple si décidez de naviguer sur les vues en partant de leur nom de type et que ces vues résident dans l'espace de noms de base plus ".Views" la méthode sera surchargée de cette façon :

```
public override Func<string, Type> GetPageNameToTypeResolver()
{
    return (viewName) =>
        {
            var rootNamespace = this.GetType().Namespace;
            var viewNamespace = rootNamespace + ".Views";
            return Type.GetType(String.Format("{0}.{1}", viewNamespace, viewName));
        };
}
```

Il est évident que dans l'exemple de code présenté ici nous utiliserons les conventions par défaut.

Il reste donc à faire le changement de classe application aussi dans le fichier App.Xaml (nous l'avons fait dans la partie C# mais pas encore dans la partie XAML) :

```
<prism:MvvmAppBase x:Class="HelloPrismForWinRT.App"
                   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                   xmlns:prism="using:Microsoft.Practices.Prism.StoreApps"
                   xmlns:local="using:HelloPrismForWinRT">

    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Common/StandardStyles.xaml" />
            </ResourceDictionary.MergedDictionaries>
            <x:String x:Key="AppName">Hello Prism</x:String>

        </ResourceDictionary>
    </Application.Resources>
</prism:MvvmAppBase>
```

On notera, outre le changement de nom classe pour la racine, que nous avons ajouté l’espace de noms “prism” qui utilise la DLL de Prism WinRT.

Modifier la classe de base des pages

Prism a une classe spécifique pour les pages qui est semblable à LayoutAwarePage que Visual Studio propose par défaut, mais cette dernière ne prend en charge que l’affichage et non la gestion d'état et de navigation. Dans une application MVVM la navigation et le code de gestion d'état vivent dans le ViewModel, et la classe de base ViewModel de Prism le prend en charge. Nous verrons comment tout cela se connecte à l’étape suivante.

Il ne faut donc pas oublier de remplacer la classe de base des pages par celle fournie par Prism (en ajoutant l’espace de noms).

```
<prism:VisualStateAwarePage x:Name="pageRoot"
                            ...
                            xmlns:prism="using:Microsoft.Practices.Prism.StoreApps">
...
</prism:VisualStateAwarePage>
```

Bien entendu le ménage se poursuit dans la partie code-behind de chaque page où ne doit subsister que le contructeur avec l’appel à InitializeComponent (sans oublier de changer la classe de base comme dans le fichier XAML par VisualStateAwarePage).

Ces modifications très simples mais importantes doivent être effectuées dans les deux pages de l’exemple.

Une dernière chose : le modèle de page proposé par VS contient un bouton appelé “backButton” dont le clic appelle la méthode “GoBack” et le IsEnabled est bindé à des

propriétés de la Frame. Tout cela ne sert plus, il faut conserver le bouton mais tous ses paramètres peuvent être supprimés, seul le nom et le style subsistent :

```
<Button x:Name="backButton"
        Style="{StaticResource BackButtonStyle}" />
```

Ajouter la classe de base aux ViewModels

Les ViewModels descendent bien entendu eux aussi d'une classe de Prism. Je dis "bien entendu" car c'est ce qu'on attend d'une toolbox de ce type : se charger des mécanismes cruciaux à notre place. Et les ViewModels sont d'une telle importance dans une architecture MVVM que c'est tout naturellement qu'ils sont pris en charge par Prism WinRT.

Les ViewModels ont désormais un code qui ressemble à cela :

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Practices.Prism.StoreApps;

namespace HelloPrismForWinRT.ViewModels
{
    public class MainPageViewModel : ViewModel
    {
    }
}
```

Dans cet exemple cela n'a pas beaucoup d'impact, voire aucun. Mais dans une application réelle et dans une prochaine partie nous verrons qu'il est important d'hériter de ViewModel qui prend en charge les subtilités de la navigation et du cycle de vie de l'application.

Connexion du ViewModelLocator aux vues

Prism WinRT fournit un ViewModelLocator comme le fait Mvvm Light mais avec une différence essentielle : par défaut son fonctionnement est totalement automatique et basé sur des conventions ce qui évite d'avoir à en compléter le code en permanence comme on doit le faire sous Mvvm Light.

Grâce à ces conventions, le ViewModelLocator de Prism sait comment connecter une Vue et son ViewModel ce qui évite le code répétitif et un peu stupide il faut le dire du ViewModelLocator de Mvvm Light (où on doit déclarer une propriété qui retourne une instance de la classe du ViewModel soit directement soit au travers d'un conteneur).

Toutefois les automatismes de Prism ne permettent pas de faire l'impasse sur la déclaration du lien au ViewModelLocator dans chaque Page (ce qui s'effectue via une propriété attachée proposée par Prism) :

```
<prism:VisualStateAwarePage ...
                              xmlns:prism="using:Microsoft.Practices.Prism.StoreApps"
                              prism:ViewModelLocator.AutoWireViewModel="True">
```

Les conventions par défaut utilisées par le ViewModelLocator sont que pour un type de vue donné il assumera que la vue est définie dans espace de noms enfant ". Views" et que le ViewModel sera défini dans un espace de noms lui aussi enfant de l'espace de noms racine sous le nom de ".ViewModels" et que la classe portera le même nom que la Vue avec le suffixe "ViewModel".

C'est une convention très simple qui ne fait que reprendre ce qu'on constate chez la plupart des développeurs utilisant une logique MVVM.

Une fois la propriété attachée ajoutée Prism saura fournir directement une instance du bon ViewModel à la Vue et initialisera au passage son DataContext convenablement.

A ce stade si vous exécutez l'application vous ne verrez pas grand chose se passer, mais si vous ajouter un contructeur par défaut au ViewModel de la vue principale et une trace à l'intérieur de ce dernier vous pourrez constater que tout ce passe comme prévu et que la page principale est bien créée et que la trace est bien suivie (qu'il s'agisse d'un log, d'un break point ou autre message de debug).

Bien entendu il serait logique d'ajouter un bouton dans la page principale pour appeler la page secondaire… Mais cela nécessiterait d'entrer dans les détails des commandes et de la navigation qui réclament malgré tout un peu d'explications. Faire entrer ces dernières dans ce billet déjà long ne serait pas une bonne idée. A chaque jour suffit sa peine, n'est-il pas… !

CONCLUSION

Prism pour WinRT simplifie beaucoup la programmation sous cette plateforme qui impose des contraintes particulières. Pour l'instant nous n'avons pas réellement pu voir ses avantages car il nous fallait poser le décor : comment construire une application avec Prism, ses vues, ses ViewModels, comment connecter le ViewModelLocator, modifier les classes de base des principaux intervenants (application, vues, VM), etc.

Ce que nous avons vu est donc peu, mais c'est déjà beaucoup !

Prism pour WinRT a été conçu "sur mesure" sans reprendre ou adapter un code existant (on verra qu'à la marge une ou deux classes de Prism 4 ont été reprises malgré tout). C'est en

partant d'une page blanche et en réfléchissant à la meilleure façon de couvrir les besoins d'une application WinRT que Prism a été construit.

De très nombreuses réunions du *Developer Guidance Advisory Council* ont permis pendant des mois de discuter des meilleures voies à prendre, des choses à affiner, de celles qui pouvaient être ignorées, etc. A chaque fois et dans un véritable esprit d'équipe les membres du groupe de développement de Prism ont su écouter les remarques, ont su aussi convaincre lorsqu'ils pensaient tenir une bonne idée. C'est ce travail en profondeur qui fait la qualité de Prism WinRT.
Je suis assez fier d'avoir participé en tant que membre du Council à cette aventure qu'est la création d'un framework MVVM fonctionnant sur une plateforme aussi sophistiquée que WinRT.

Mes réserves concernant l'utilisation de WinRT pour des applications LOB ne changent pas pour autant dans l'état actuel de cette technologie (ce qui peut évoluer), et il en va de même de mon regard critique sur ce que je considère des erreurs d'approche faites par Microsoft dans la présentation et la diffusion de WinRT qui amène à une situation plus proche du rejet que de l'adoption massive, et c'est un euphémisme. Mais ces critiques concernent les décisions prises par la Direction de Microsoft, décisions concernant la façon de "vendre" WinRT et Windows 8, ce sont des erreurs de management, graves à mes yeux, mais ces critiques ne concernent en rien la plateforme WinRT qui offre de nouvelles possibilités réellement excitantes ni la qualité réelle de Windows 8 en tant qu'OS.

C'est pourquoi, jeu d'équilibrisme difficile je l'accorde, mais parfaitement cohérent, je continuerai à affirmer que l'UX de Windows 8 est un échec sur PC, que présenter WinRT comme seul moyen de créer des applications sous Windows est une hérésie et que toutes ces erreurs se payent aujourd'hui au prix fort par une faible adoption de ces produits, le tout en continuant d'affirmer que ces derniers ne sont pas en cause et qu'autant Windows 8 que WinRT sont des produits exceptionnels et bien ficelés techniquement.

J'aime toujours autant la technologie Microsoft, je soutiens toujours autant la qualité incroyable du travail des équipes Microsoft, mais, vous l'avez compris, je suis pressé que Microsoft change de Direction. Être clair et honnête avec ses lecteurs comme avec ses clients c'est dire ce qu'on croit être juste. C'est une question de sérieux et de crédibilité. Deux choses auxquelles je suis profondément attaché.

C'est pourquoi il y aura une partie 3 sur Prism pour WinRT dans quelques temps !

Stay Tuned !

PS: vous pouvez télécharger les exemples PrismForWindowsRuntimeQuickStarts.zip depuis CodePlex et consulter l'exemple "HelloWorld" dont est inspiré la structure de l'exemple de cet article.

Prism pour WinRT– Partie 3 – Navigation et Commandes

La partie 1 a posé le décor, la partie 2 a montré la mise en œuvre de base, cette partie 3 sur Prism pour Windows Runtime nous fait découvrir comment naviguer avec Prism et ce qu'il propose pour la gestion des commandes.

NAVIGATION ET COMMANDES

Dans la partie 2 de cette série j'ai abordé les bases de Prism pour Windows Runtime. L'exemple d'application que je vous ai aidé à créer n'était pas très impressionnant, après le déroulement de plusieurs étapes nous n'avions qu'une sorte de "Hello World" ni mieux ni moins bien que ce qu'on pourrait faire en une ligne de C# en mode console… Mais tout ce qui ne se voyait pas encore a permis de jeter les bases sur lesquelles vous allez pouvoir construire des applications métier complexes pour Windows 8+ avec Prism, en utilisant le modèle MVVM, l'intégration avec le système de navigation, la manipulation facilitée de la gestion des états suspendu / terminé et bien plus !
Dans cet article je vais continuer à améliorer l'application exemple pour vous montrer comment vous pouvez gérer les dépendances des ViewModels, comment gérer les commandes de l'interface utilisateur dans ces derniers, et la façon de laisser ces mêmes ViewModels participer activement à la navigation et communiquer des données.

NAVIGUER DEPUIS LA MAINPAGE

Il aurait été facile de placer un gros bouton sur la MainPage et d'en gérer le clic pour atteindre la seconde page de notre exemple. Bien entendu cela revient à tout ignorer de Prism et de ce que signifie naviguer dans une véritable application… En général une application propose plusieurs options différentes et le plus souvent sous Modern UI ces options se trouvent dans des GridView, c'est en cliquant sur un item qu'on déclenche une navigation (vue de détail, informations complémentaires etc). Le style même des applications Modern UI est directement issu des "Apps" pour smartphones ou tablettes, un monde dans lequel l'UI se fait la plus légère possible pour que l'utilisateur interagisse directement sur les données et non au travers de menus et sous-menus complexes.

Metro et Windows Phone 7 avaient mis en valeur les polices de caractères et leur rôle important. Les mots devenant cliquables sans pour autant être enchâssés dans des "boutons". Ce design était d'ailleurs inspiré par celui du logiciel Zune fournit avec la machine du même nom (qui n'existe plus). Modern UI est la continuité de ce langage de design. On retrouve d'ailleurs les mêmes évolutions chez Google dans Android où certains boutons disparaissent pour ne laisser que les mots et on a pu voir récemment Apple proposer un iOS

7 dont l'inspiration flat en fait un enfant illégitime de Android et Metro s'éloignant du skeuomorphisme cher à cet OS.

De fait, sous Modern UI on évite les boutons et encore plus les menus à l'ancienne. Les ListBox de Windows Forms à Silverlight en passant par ASP.NET sont devenus des éléments d'UI essentiels. WinRT, tout comme Android et iOS n'échappent pas à cette façon de présenter tout sous forme de listes (quelle qu'en soit la classe et les variantes exactes, ListBox, GridView…).

Les aficionados de XAML se rappelleront d'ailleurs certainement de leur surprise quand ils découvrirent l'exemple WPF "sticky notes" (des sortes de petits post-it à la suite les uns des autres) et qu'ils prirent conscience en voyant le code que tout cela n'était qu'une simple ListBox avec des items designés…

Le design, XAML l'a dans le sang. Les autres OS et autres plateformes le découvrent au fur et à mesure, des années après !

Fidèle à cet esprit que nous appliquons depuis longtemps avec WPF et Silverlight, et en conformité avec l'esprit même de Modern UI aujourd'hui, nous n'allons pas placer un gros bouton sur notre MainPage. Non. Nous allons placer une GridView qui sera bindée à une liste d'Items, chacun déclenchant une commande lorsqu'il est cliqué.

```
<GridView Margin="120,10,0,0"
            Grid.Row="1"
            SelectionMode="None"
            ItemsSource="{Binding NavCommands}">
    <GridView.ItemTemplate>
        <DataTemplate>
            <Grid Height="500"
                    Width="250"
                    Background="#33CC33">
                <StackPanel VerticalAlignment="Center">
                    <TextBlock Text="{Binding}"
                                Style="{StaticResource SubheaderTextStyle}"
                                HorizontalAlignment="Center" />
                </StackPanel>
            </Grid>
        </DataTemplate>
    </GridView.ItemTemplate>
</GridView>
```

Comme le montre le code ci-dessus la GridView voit son ItemsSource bindé à la propriété "NavCommands" du ViewModel et elle contient un ItemTemplate (DataTemplate) récupérant le texte de la commande pour l'afficher. Tout cela est du XAML que nous connaissons depuis longtemps.

UTILISER UN BEHAVIOR ATTACHE POUR GERER L'ITEMCLICK

Le GridView n'est pas forcément l'élément d'UI le plus adapté à la gestion des commandes, il n'expose d'ailleurs aucune propriété ICommand qui permettrait de se brancher directement sur le ViewModel. Mais cela n'est pas très grave, en XAML nous savons que dans une situation de ce genre il nous suffit d'ajouter un Behavior qui routera n'importe quel évènement de la source vers un ICommand du ViewModel.

Mais nous avons un petit problème ici car le XAML de WinRT n'est pas encore au niveau de celui de Silverlight ni de WPF… Et "out of the box" seuls les Behavior attachés sont possibles. C'est moins pratique que les vrais Behaviors de SL ou WPF. Mais cela fait le job tout de même…

La première chose consiste donc à utiliser un tel Behavior attaché. Mais aucun de ce genre n'est fourni non plus ! WinRT est bien jeune, laissons-lui le temps de murir un peu. Oui mais on ne va pas attendre d'avoir des toiles d'araignées non plus… La solution consiste donc à prendre notre clavier et à coder un tel Behavior (une propriété Attachée qui remplira le rôle d'un Behavior pour être précis). Voici le code de ce vrai-faux Behavior :

```
public static class ItemClickToCommandBehavior
{
    #region Command Attached Property
    public static ICommand GetCommand(DependencyObject obj)
    {
        return (ICommand)obj.GetValue(CommandProperty);
    }

    public static void SetCommand(DependencyObject obj, ICommand value)
    {
        obj.SetValue(CommandProperty, value);
    }

    public static readonly DependencyProperty CommandProperty =
        DependencyProperty.RegisterAttached("Command", typeof(ICommand),
            typeof(ItemClickToCommandBehavior),
            new PropertyMetadata(null, OnCommandChanged));
    #endregion

    #region Behavior implementation
    private static void OnCommandChanged(DependencyObject d,
        DependencyPropertyChangedEventArgs e)
    {
        ListViewBase lvb = d as ListViewBase;
        if (lvb == null) return;

        lvb.ItemClick += OnClick;
    }

    private static void OnClick(object sender, ItemClickEventArgs e)
    {
        ListViewBase lvb = sender as ListViewBase;
        ICommand cmd = lvb.GetValue(ItemClickToCommandBehavior.CommandProperty) as ICommand;
        if (cmd != null && cmd.CanExecute(e.ClickedItem))
            cmd.Execute(e.ClickedItem);
    }
    #endregion
```

Rien d’exceptionnel, les habitués de C# et XAML sauront comprendre ce bout de code. En gros la propriété de dépendance qui est définie s’accroche à tout descendant de ListViewBase et détourne son OnClick. Lorsque celui-ci est déclenché la propriété de dépendance récupère la commande associée au ListViewBase (contenue dans le faux Behavior) et si elle existe et qu’elle peut être exécutée elle l’est. Le ICommand pointé est situé dans le ViewModel mais cela n’apparait pas dans ce code (la commande pourrait venir de n’importe où à ce niveau).

L’utilisation de ce “faux Behavior” est d’une simplicité déroutante :

```
<GridView Margin="120,10,0,0"
          Grid.Row="1"
          SelectionMode="None"
          ItemsSource="{Binding NavCommands}"
          beh:ItemClickToCommandBehavior.Command="{Binding NavCommand}"
          IsItemClickEnabled="True">
```

GERER LES COMMANDES DE NAVIGATION

Nous avons mis en place tout le nécessaire dans l'UI XAML, reste à voir comment cela va être pris en compte côté ViewModel.

La première chose est de créer une propriété NavCommands qui contient la liste des commandes exposée par le GridView. Ensuite il faut exposer une commande "NavCommand" qui sera utilisée par le Behavior :

```
public class MainPageViewModel : ViewModel
{
    public MainPageViewModel()
    {
        NavCommands = new ObservableCollection<string>
        {
            "Add Sale"
        };
        NavCommand = new DelegateCommand<string>(OnNavCommand);
    }

    public ObservableCollection<string> NavCommands { get; set; }
    public DelegateCommand<string> NavCommand { get; set; }

    private void OnNavCommand(string navCommand)
    {

    }
}
```

Du grand classique dans la "plomberie" C#/XAML donc. Rien ici n'est spécifique à Prism ou même WinRT. Pour le moment.

Pour que la navigation s'effectue correctement il nous faut utiliser le NavigationService offert par la classe MvvmAppBase de Prism. Cela signifie que nous avons besoin d'injecter cette dépendance dans notre ViewModel.

Il y a deux façons d'arriver à ce résultat avec Prism pour WinRT. Soit en utilisant une Factory pour la construction du ViewModel, soit en utilisant un conteneur d'injection de dépendances.

Voici comment ces alternatives se présentent :

La Factory

Dans ce premier cas c'est le constructeur du ViewModel qui va prendre en paramètre la dépendance, ici une instance de l'interface INavigationService. Une fois le constructeur modifié il faut indiquer au ViewModelLocator de Prism de l'utiliser en place et lieu du constructeur par défaut et en lui passant le bon paramètre. le code va ressembler à cela :

```
public class MainPageViewModel : ViewModel
{
    private readonly INavigationService _NavService;
    public MainPageViewModel(INavigationService navService)
    {
        _NavService = navService;
        ...
    }
    ...
}
```

Ici le constructeur de MainPageViewModel a été modifié pour accepter un paramètre de type INavigationService. Lors de son exécution ce code récupère la valeur du paramètre et la stocke dans une variable privée afin de pouvoir utiliser le service plus tard (dans la prise en compte de la commande liée au Behavior).

Il faut maintenant aller dans le code App.Xaml.cs et surcharger la méthode OnInitialize pour indiquer au ViewModelLocator comment instancier un MainPageViewModel avec le bon paramètre :

```
protected override void OnInitialize(IActivatedEventArgs args)
{
    base.OnInitialize(args);
    ViewModelLocator.Register(typeof(MainPage).ToString(),
        () => new MainPageViewModel(NavigationService));
}
```

Rien de bien compliqué là non plus, le locator possède une méthode Register qui prend le code la Vue en paramètre (ici nous suivons la convention nom de code = nom de classe) ainsi qu'un délégué chargé de créer l'instance du VM correspondant.

Ce mécanisme créé une "route", un lien entre une Vue et son VM de façon simple. Ni la Vue ni le VM ne se connaissent, MVVM est toujours respecté. Les routes sont centralisée dans App.Xaml.cs ce qui simplifie la maintenance de l'application. Si demain nous choisissons de connecter un autre VM à la même Vue, il suffira de changer le Register dans App.Xaml.cs et le reste de notre application n'en saura rien, évitant ainsi les bogues en cloisonnant et en découplant tous les "tiers" au maximum (avec un minimum de code il faut l'admettre).

Cette technique est simple, pratique, facilement modifiable, bref elle répond au besoin de façon très satisfaisante.

Bien entendu cela reste un peu "rustique". Voyons l'autre moyen d'arriver au même résultat…

Le conteneur d'IOC

Si vous êtes à l'aide avec le concept d'injection de dépendances et d'Inversion de Contrôle, peut-être préfèrerez-vous gérer la dépendance du VM au service de navigation en accord avec toutes les autres dépendances que votre application devra prendre en charge ?

Dans ce cas il y a fort à parier que vous choisissiez d'utiliser un DIC (Dependency Injection Container), un conteneur d'injection de dépendances de type Unity.

Pour ce faire il suffit de faire un clic droit sur le projet (dans l'explorateur de solution) et de cliquer sur "Manage Nuget packages". Unity pour WinRT était en preview quand j'ai réalisé le test mais lorsque vous lirez ces lignes cela ne devrait plus être le cas. "Unity for .NET 4.5/WinRT" devrait ainsi apparaitre dans la liste des packages (sinon cochez la case "include prerelease" pour afficher aussi les préversions).

Une fois Unity installé dans le projet il ne suffit plus que de modifier le App.Xaml.cs pour modifier le comportement du ViewModelLocator :

```
sealed partial class App : MvvmAppBase
{
    IUnityContainer _Container = new UnityContainer();
    public App()
    {
        this.InitializeComponent();
    }

    protected override void OnLaunchApplication(LaunchActivatedEventArgs args)
    {
        NavigationService.Navigate("Main", null);
    }

    protected override void OnInitialize(IActivatedEventArgs args)
    {
        base.OnInitialize(args);
        _Container.RegisterInstance<INavigationService>(NavigationService);
        ViewModelLocator.SetDefaultViewModelFactory(
            (viewModelType) => _Container.Resolve(viewModelType));
    }
}
```

Vous risquez de penser “mais quelle différence avec la version précédente, il y a autant de code ?”.

Ce n’est pas faux, mais c’est une impression !

Regardez de plus près : dans OnInitialize nous récupérons le NavigationService de Prism que nous stockons dans un conteneur Unity (créé en haut dans une variable privée) et ensuite nous modifions totalement la factory par défaut du ViewModelLocator pour lui transmettre un délégué qui demande à Unity de résoudre les dépendances du type passé (un VM).

Cela est très différent de la situation précédente. En effet, dans le premier cas nous devons injecter la dépendance de chaque VM “à la main” en modifiant App.Xaml.cs et en indiquant à chaque fois (par un Register) une route spécifique pour chaque couple View/VM. Alors qu’ici nous modifions _une fois pour toute_ le fonctionnement du ViewModelLocator pour laisser le soin à Unity de découvrir les dépendances à injecter dans chaque VM, ces derniers pouvant avoir des constructeurs acceptant une ou plusieurs dépendances non connues pour le moment…

Les deux approches sont fonctionnellement proches, la première est facile à comprendre mais réclame plus d’attention dans sa mise en œuvre dans le temps (maintenance et évolution du code, et même durant sa première écriture), la seconde méthode est

générique, capable de gérer des situations plus complexes automatiquement, elle est plus subtile mais réclame peut-être une meilleure maitrise des notions d'IOC et de DIC.

C'est cette méthode que je vous suggère d'utiliser. Mais à vous de choisir, Prism s'adapte !

Naviguer !

Gréer convenablement son voilier est souvent long. Cela demande de la précision et une attention de chaque détail. Mais une fois le job bouclé, on peut enfin naviguer en toute sécurité !

Avec Prism c'est la même chose…

Maintenant que nous savons comment récupérer proprement le service de navigation de Prism dans chaque VM il est temps de permettre à MainPage de passer à seconde page de notre magnifique application !

Est-ce bien raisonnable tout ça ?

Mais avant une question peut se poser (en tout cas devrait se poser…) :

"Pourquoi tout ce cinéma alors que le service de navigation est fourni par Prism, que l'application est construite pour tourner avec Prism et qu'il suffisait d'appeler ce service dans le VM pour naviguer tranquillement ?" vous dites-vous…

Aie… Question piège.

Mais sacrément importante, et je vous remercie de l'avoir posé cela prouve que vous suivez 😊

Dans l'absolu le service de navigation est offert par la version surchargée de l'objet application fourni par Prism et cet objet est disponible dans toute l'application. L'appeler directement n'est pas "mal" et permet donc de naviguer tout de suite sans mettre en place des détournements savants qui semblent donc compliquer les choses par pur plaisir.

C'est en tout cas ce qu'on peut penser de prime abord. Ecrire une application en appelant directement le service de navigation partout où on en a besoin n'en ferait pas une mauvaise application ni ne conduirait à des bogues délirants. C'est vrai. Mais cela rendrait tout le code écrit entièrement dépendant du service de navigation particulier qu'est celui fourni par l'objet application de Prism. Si demain nous souhaitons modifier ce service pour l'adapter à un besoin particulier de l'application c'est toute cette dernière qu'il faudra vérifier et les bogues sournois apparaitront alors… En effet, le service de Prism existera toujours (à moins de supprimer Prism totalement de l'application), donc tous les codes y faisant référence

seront toujours valables d'un point de vue technique, donc aucun moyen simple de repérer là où les changements n'auront pas été faits pour appeler le nouveau service de navigation… Et c'est ainsi que les ennuis commencent et que les spaghettis du code commencent à s'emmêler…

L'intérêt d'un découplage fort se trouve là. Prism tout entier, MVVM aussi, toute ces méthodes ne servent quasiment qu'un seul but, le découplage fort. Certains penseront que c'est beaucoup s'embêter pour rien, je l'ai déjà entendu en tout cas, coder à l'arrache c'est tellement plus fun…
Quand on travaille dans une SSII parisienne et qu'on sera parti pour une autre avant la fin du projet ou sa maintenance c'est certainement une philosophie jouable, malhonnête mais jouable. Quand on a le souci de bien faire son job et plus encore si on doit le terminer puis le faire évoluer, tout de suite ce genre de position n'est plus tenable, on doit faire dès le départ de la qualité pour ne pas avoir à le regretter après !
Respecter le code qu'on écrit, peu importe si on considère être bien ou mal payé pour le faire, c'est se respecter soi-même avant tout, et cela … n'a pas de prix !

Levons l'ancre !

Il était essentiel de s'octroyer le temps d'un aparté pour expliquer. Les méthodologies ne s'appliquent pas pour le plaisir, parce c'est la mode ou je ne sais quelle autre raison farfelue, on les applique parce qu'on les comprend et qu'on saisit les avantages qu'on en tirera. Si une méthode ne vous convainc pas, si on se refuse à vous en expliquer le pourquoi du comment, si on laisse sous silence des interrogations cruciales, ne l'appliquez pas, et changez d'interlocuteur.

Mais il est temps de naviguer, levons l'ancre, attention tambours, voici le code que nous pouvons maintenant écrire dans le MainPageViewModel :

```
private void OnNavCommand(string navCommand)
{
    _NavService.Navigate("AddSale", null);
}
```

C'est sûr ça fait peu pour toutes ces explications !

Pour rappel le premier argument est une chaîne qui indique le nom logique de la Vue (dans cet exemple “AddSale”).

Arrivez ici vous pouvez compiler l'application et la lancer, et bien entendu cliquer sur l'appel à la seconde page. Elle sera vide à ce niveau de l'exercice (placez-ci un rectangle de couleur ou autre pour la différencier de la page principale).

I'll be back !

Oui, certainement… mais pour le moment vous êtes bloqué sur la page deux et le bouton "back" ne sert à rien…

Cherchez un peu … tic tac tic tac… Et oui, le ViewModel de la page 2 ne contient rien et donc ne gère pas la navigation. Il va falloir modifier ce VM pour y ajouter la prise en charge de la navigation.

Pour ce faire lui aussi aura besoin d'accéder au NavigationService. Il le fera exactement de la même façon (méthode manuelle ou par conteneur d'injection).

Vous en arriverez de toute façon à la nécessité d'un code semblable à ce celui-ci :

```
public class AddSalePageViewModel : ViewModel
{
    private readonly INavigationService _NavService;
    public AddSalePageViewModel(INavigationService navService)
    {
        _NavService = navService;
        GoBackCommand = new DelegateCommand(
            () => _NavService.GoBack(),
            () => _NavService.CanGoBack());
    }

    public DelegateCommand GoBackCommand { get; set; }
}
```

L'accès au NavigationService ouvre la porte de ces méthodes dont "GoBack" qui exécute le retour en arrière et "CanGoBack" qui indique si un tel retour est possible.

Il ne faudra pas oublier de modifier le code XAML du bouton "backButton" se trouvant sur la page 2 :

```
<Button x:Name="backButton"
        Command="{Binding GoBackCommand}"
        Style="{StaticResource BackButtonStyle}" />
```

Et voilà !

Une fois le principe compris et les bases en place, gérer même cinquante pages devient un jeu d'enfant…

NAVIGUER DEPUIS UNE APPBAR

La navigation par la page Main de notre application et son GridView servant de menu, la navigation arrière par le bouton “back”, tout cela n’est pas suffisant dans une véritable application. Si vous voulez que l’utilisateur puisse naviguer plus directement et plonger dans l’application à différents niveau sans suivre un parcours dicté par les pages le plus logique sera d’ajouter un menu dans la AppBar WinRT.

Par exemple on pourrait ajouter à la MainPage un telle AppBar de cette façon :

```
<Page.Resources>
    <Style x:Key="AddSaleAppBarButtonStyle"
           TargetType="ButtonBase"
           BasedOn="{StaticResource AppBarButtonStyle}">
        <Setter Property="AutomationProperties.AutomationId"
                Value="AddAppBarButton" />
        <Setter Property="AutomationProperties.Name"
                Value="Add Sale" />
        <Setter Property="Content"
                Value="&#xE109;" />
    </Style>
</Page.Resources>
<Page.TopAppBar>
    <AppBar>
        <StackPanel Orientation="Horizontal">
            <Button Command="{Binding AddSaleCommand}"
                    Style="{StaticResource AddSaleAppBarButtonStyle}" />
        </StackPanel>
    </AppBar>
</Page.TopAppBar>
```

Le ViewModel de la MainPage sera complété d’une commande :

```
public DelegateCommand AddSaleCommand { get; set; }
```

Commande qui sera initialisée de la sorte :

```
AddSaleCommand = new DelegateCommand(() => _NavService.Navigate("AddSale", null));
```

TRANSMETTRE DES INFORMATIONS ENTRE VIEWMODELS

Très souvent il arrive qu’on ait besoin de transmettre des informations depuis le VM départ vers le VM destination d’une navigation. Le cas le plus classique est celui du maître/détail : une page présente une liste (maître), l’utilisateur clique sur un item et navigue alors sur la page de détail. Cette dernière a besoin de connaître soit l’item (l’objet lui-même) soit au minimum son ID. Des informations simples mais qu’il faut bien transmettre.

Et comment transmettre d'un VM à l'autre lorsque le découplage est très fort entre les VM au point qu'il leur est impossible de dialoguer directement les uns avec les autres ?

D'où je viens et où vais-je ?

Il n'y a pas que les philosophes qui s'interrogent ainsi… Les VM aussi peuvent avoir besoin de savoir quel est leur parcours, qui les a précédés et qui les suivra, et si quelque chose leur a été transmis ou s'ils doivent eux-mêmes transmettre à leur suiveur quelques données en héritage… La vie des VM est finalement pleine de poésie et de questions existentielles. C'est bien pour cela qu'on parle de leur cycle de vie !

Un VM peut ainsi avoir à récupérer des données du VM qui vient de l'appeler. Prism gère cela de façon très simple. Si vous avez fait attention, la méthode Navigate() du NavigationService prend deux paramètres, le premier étant le nom logique de la vue, le second pour l'instant a été utilisé à null. En réalité il s'agit d'un contexte que tout VM qui en appelle un autre peut utiliser pour transmettre des données.

Passer les données entre VM est donc très simple. Mais pour les recevoir ? Le VM de base, celui que Prism offre pour créer vos propres VM, contient une méthode intéressante qu'il est possible de surcharger : OnNavigatedTo.

Cette méthode est appelée après la construction du VM et c'est ici que ce dernier peut récupérer le contexte passer par Navigate() du NavigationService :

```
public override void OnNavigatedTo(object navigationParameter,
    NavigationMode navigationMode, Dictionary<string, object> viewState)
{
    base.OnNavigatedTo(navigationParameter, navigationMode, viewState);
}
```

Trois paramètres sont passé par Prism : un "navigationParameter", le fameux context envoyé par Navigate(), un "navigationMode" qui est déterminé par WinRT et qui prend les valeurs "New", "Forward", "Back" ou "Refresh", et un troisième paramètre, "viewState".

Le mode de navigation permet de savoir comment un VM a été activé vis à vis de la pile de navigation. Une navigation par Navigate créera un "New", une nouvelle branche de navigation. Un retour en arrière indiquera un "Back". Le Forward et le Refresh sont pris en charge par la Frame WinRT mais ne sont pas implémentés directement dans Prism car il s'agit de mode navigation rarement explicites dans une application. Il est toutefois possible de les gérer en adaptant le code de FrameNavigationService.

Quant au "viewState", un dictionnaire de paires clé/objet, il s'agit d'un réservoir de données dans lequel on peut lire et écrire et qui participe au mécanisme de suspension et

terminaison des applications sous WinRT. Il peut être ignoré pour l'instant car c'est une autre histoire que je vous raconterais la prochaine fois !

CONCLUSION

Naviguer peut se faire comme un marin d'eau douce ou comme un vieux loup de mer… On peut voguer sur une coquille de noix ou un destroyer de dernière génération.

Il y a peu de différence, sauf que dans certains de ces cas arriver à bon port sera un miracle alors qu'en optant pour de meilleurs choix c'est de ne pas arriver sain et sauf qui sera rarissime.

Prism vous offre le destroyer, l'annexe pour embarquer et débarquer facilement, les bouées de secours et l'assurance d'une mer calme. Pourquoi choisir d'autres voies plus incertaines ?

Prism pour WinRT – Partie 4 – Gestion des états de l'application

La partie 1 a posé le décor, la partie 2 a montré la mise en œuvre de base, la partie 3 de cette étude de Prism pour Windows Runtime nous a fait découvrir comment naviguer avec Prism et ce qu'il propose pour la gestion des commandes. La partie 4 aborde la gestion délicate des états de l'application.

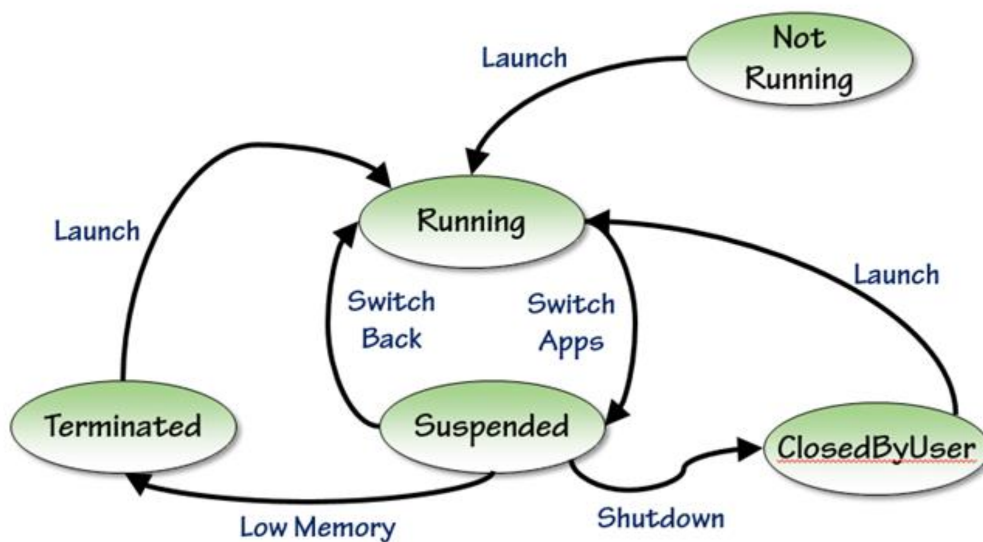
LE CYCLE DE VIE DES APPLICATIONS SOUS WINRT

Avec Windows et le bureau dit classique les applications ont un cycle de vie simple : elles sont exécutées par l'utilisateur puis à un moment donné terminées, généralement par le même utilisateur.
Avec la nécessité de gérer de nouvelles plates-formes matérielles comme les tablettes et les smartphones le cycle de vie des applications a été bouleversé. Des changements ont été fait en profondeur afin de minimiser l'usage des ressources du système et maximiser la durée de la batterie.
De fait ce qui était rudimentaire sous Linux ou Windows devient sophistiqué pour une application moderne sous iOS, Android, Windows Phone et même WinRT sur PC qui adopte un fonctionnement identique à sa version tablette. Cela peut être vu comme une contrainte inutile pour WinRT PC, et d'un certain point de vue ce n'est pas faux, mais en retour cela permet à une même application, un même code, de fonctionner sur PC et sur Surface… Une différence importante entre ces deux environnements aurait de toute façon déclenché le même type de reproche, mais dans l'autre sens ! ("comment ? MS produit un OS cross-plateforme et il y a de si grandes nuances de programmation ?" – vous voyez ce n'est pas si simple de plaire à tout le monde !).

Comment donner l'impression de fluidité, comment offrir la puissance maximale à l'application en cours tout en laissant à l'utilisateur la possibilité de revenir sur l'application précédente dans l'état où il l'a quittée et sans qu'il ne se doute qu'elle a été arrêtée, voire vidée de la mémoire ?

La réponse se trouve dans la gestion du cycle de vie des applications. Les nuances sont minces entre les OS mobiles et tous adoptent un même mécanisme de "suspension", d'arrêt temporaire, etc.

Pour WinRT le schéma du cycle de vie est le suivant :



une vision plus complète des changements d'état

Cycle de vie d'une application sous WinRT (illustration empruntée au livre "Développement Windows 8" chez Eyrolles auquel j'ai collaboré)

On voit que trois états dominent : L'état non démarrée (l'application n'est pas lancée), l'état "en cours d'exécution" (l'application est en avant-plan et fonctionne normalement) et l'état suspendue (l'application est dans un état intermédiaire).

On démarre une application, elle passe dans l'état "en cours d'exécution", le message envoyé est "Activating" (activation). Mais durant le fonctionnement de l'application il n'y a plus la possibilité de l'arrêter ! L'utilisateur ne fait que passer d'une application à l'autre sans se soucier de les arrêter. Sous WinRT il y a une gestuelle (et le raccourci Alt-F4) qui permettent de fermer une application mais cela est anecdotique et ne fait pas partie du cycle normal de vie.

En revanche, le passage d'une application à une autre va placer celle qui est quittée dans ce fameux état intermédiaire qu'est l'état "suspendue" (transition "Suspending"). Dans cet état c'est l'OS qui va décider selon les besoins en RAM et en puissance ce qu'il va advenir de l'application en mémoire. Si elle devient gênante elle sera tout simplement fermée (transition "terminating"). Mais si l'utilisateur revient dessus avant cette opération de vidange de la mémoire elle sera reprise tel quel de la mémoire pour prendre l'avant-plan. Auquel cas la transition sera "resuming" (reprise).

De la cuisine interne allez-vous dire… Hélas non.

Car cette "cuisine" si elle bien gérée par l'OS sans aucun contrôle de l'utilisateur ni même du développeur n'est pas sans conséquences pour ce dernier !

Si votre application passe de l'état actif à l'état suspendu puis est immédiatement reprise (resuming) en effet cela n'a aucune incidence. Mais dans la réalité elle passera souvent par l'état d'arrêt. Que deviennent les données en cours de saisie ? Si l'utilisateur revient sur l'application verra-t-il tout son travail perdu ou bien retrouvera-t-il, comme il s'y attend, l'application dans l'état où il l'a quittée (même en plein milieu d'une saisie non validée) ?

La réponse est très simple : rien. Il n'arrive rien. Par défaut tout est perdu.

C'est grâce aux transitions évoquées, qui sont autant de messages ou d'appels de méthodes, que l'OS prévient votre application. Il lui faudra réagir vite pour sauvegarder tout ce qui doit l'être ! WinRT n'offre que très peu de temps avant de détruire l'application. De même, dans l'autre sens, à la reprise de l'application, en fonction des éléments sauvegardés et du message d'activation (Activating, Resuming) il faudra décider quelle page doit être affichée

mais aussi réhydrater tous les objets pour que l'utilisateur se retrouve dans l'exacte situation qu'il a laissée…

Et c'est là que ça se complique un peu.

Pour résumer, le cycle de vie des applications en environnement mobile ajoute un niveau de complexité totalement nouveau qui, bien que signalé par des évènements de l'OS, reste totalement à la charge du développeur…

PRISM DANS TOUT ÇA ?

Ce qu'on attend d'une bibliothèque de ce genre, totalement adaptée à l'OS, c'est bien entendu qu'elle simplifie ce qui est complexe. Ce type d'aide est tellement indispensable qu'on en viendrait à se dire que les OS sont "mal finis" car ils devraient apporter cette couche de simplification d'emblée. Mais comme si la cohérence et la simplicité était un luxe, un bonus, tous les OS mobiles s'arrêtent quelques lignes de code avant cette étape et laisse ainsi le champ libre à des bibliothèques de combler ce qu'ils auraient du proposer dès le départ. Etrange façon de concevoir un logiciel (un OS n'est qu'un logiciel comme un autre).

Quoi qu'il en soit Prism pour WinRT met en place des mécanismes qui aident le développeur à gérer le cycle de vie des applications. C'est l'une des raisons qui dans les parties précédentes nous a obligé à faire descendre les Page d'un type fourni par Prism car en dehors de la mise en page il prend en charge les aspects de navigation et de cycle de vie. Il en va de même pour les ViewModels et même pour l'objet application.

L'APPLICATION EXEMPLE

Pour illustrer ce billet je vais utiliser une application exemple écrite par un membre de l'équipe Prism. Vous pouvez la télécharge directement ici :

Solution HellowPrismWinRT partie4 example.zip

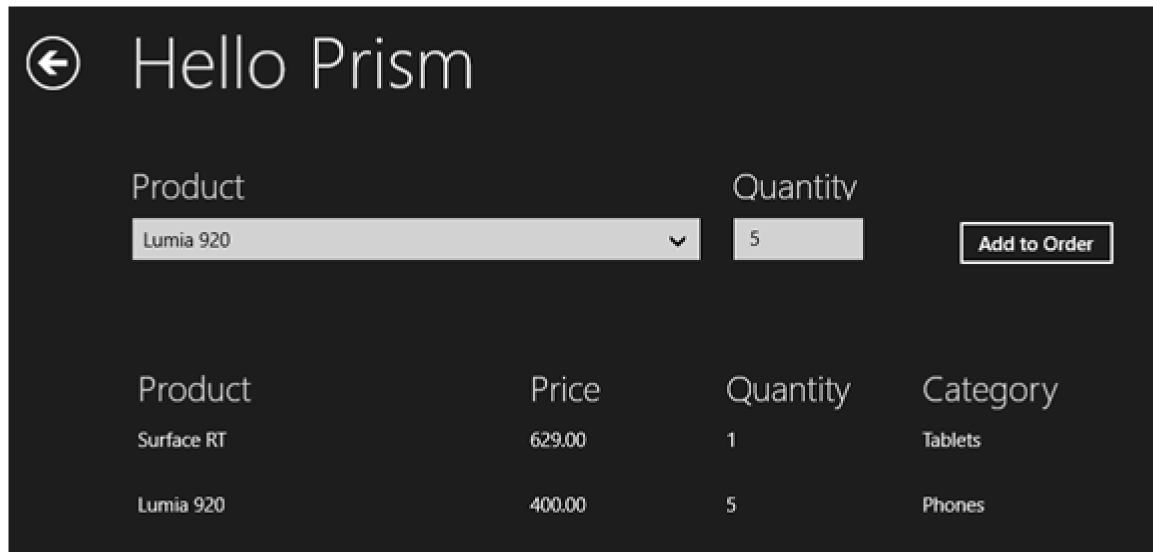
AJOUTER DES DONNEES

L'exemple prolonge le code étudié dans les parties précédentes, une application WinRT avec une page principale et une seconde page. La page Main contient un "menu" qui permet d'accéder à la page de saisie, cette dernière permettant d'ajouter des articles à un charriot comme sur un site marchand.

Dans la partie 3 cette seconde page était restée vide.

Nous allons lui ajouter la saisie d'articles, comme une commande sur un site Web, afin de disposer de données dans deux états différents : "en cours de saisie" et "validées".

Les données en cours de saisie seront celles qui permettent d'ajouter un article et sa quantité au charriot, les données validées seront représentées en mémoire par la liste d'objets déjà ajoutés.



Voici (ci-dessus) la capture de la page secondaire de l'application exemple depuis le simulateur.

Le code source étant fourni (téléchargeable dans le paragraphe précédent), je vous passe les détails de mise en page et d'une partie du code.

Les zones "product" et "quantity" en haut de l'écran permettent de choisir l'article et sa quantité, le bouton "add to order", ajoute tout cela à la commande en cours, et la liste en dessous récapitule ce qui est déjà dans cette dernière.

La liste des produits disponibles est gérée ici par une simple liste en mémoire qui joue le rôle de la base de données :

```
public class ProductsRepository : IProductsRepository
{
    public IEnumerable<Product> AllProducts()
    {
        return new List<Product>
        {
            new Product { ProductId = 1, ProductName = "Surface Pro",
                Category="Tablets", UnitPrice = 999.00m, UnitsInStock=42},
            new Product { ProductId = 2, ProductName = "Surface RT",
                Category="Tablets", UnitPrice = 629.00m, UnitsInStock=33},
            new Product { ProductId = 3, ProductName = "Lumia 920",
                Category="Phones", UnitPrice = 400.00m, UnitsInStock=12},
            new Product { ProductId = 4, ProductName = "iPhone 5",
                Category="Phones", UnitPrice = 629.00m, UnitsInStock=54},
            new Product { ProductId = 5, ProductName = "Razr",
                Category="Phones", UnitPrice = 59.99m, UnitsInStock=987},
        };
    }
}
```

L'application gère un repository pour la commande en cours :

```
public class OrderRepository : IOrderRepository
{
    readonly ObservableCollection<OrderItem> _OrderItems =
        new ObservableCollection<OrderItem>();

    public void AddToOrder(Product product, int quantity)
    {
        var orderitem = (from oi in _OrderItems where oi.Product.ProductId
                         == product.ProductId select oi).FirstOrDefault();
        if (orderitem == null)
            _OrderItems.Add(new OrderItem { Product = product, Quantity = quantity });
        else orderitem.Quantity += quantity;
    }

    public ObservableCollection<OrderItem> CurrentOrderItems
    {
        get { return _OrderItems; }
    }
}
```

Du C# bien classique sur lequel je ne m'étendrais pas.

EXPOSER LES DONNEES POUR LA VUE

La vue capturée plus haut s'attend bien entendu à trouver certaines propriétés dans le ViewModel comme la liste de produits (Products dans le code XAML), l'ID du produit sélectionné, la quantité choisie et une commande pour le bouton d'ajout. Il faut aussi une liste CurrentOrderItems pour gérer l'affichage des objets faisant déjà partie de la commande.

On retrouve tout cela dans le ViewModel “AddSalepageViewModel” :

```
public class AddSalePageViewModel : ViewModel
{
    private readonly INavigationService _NavService;
    private readonly IProductsRepository _ProductsRespository;
    private readonly IOrderRepository _OrderRepository;
    private int _SelectedProductId;
    private int _Quantity = 1;

    public AddSalePageViewModel(INavigationService navService,
        IProductsRepository productsRespository,
        IOrderRepository orderRepository)
    {
        _OrderRepository = orderRepository;
        _ProductsRespository = productsRespository;
        _NavService = navService;
        GoBackCommand = new DelegateCommand(
            () => _NavService.GoBack(),
            () => _NavService.CanGoBack());
        AddToOrderCommand = new DelegateCommand(OnAddToOrder, CanAddToOrder);
    }

    public DelegateCommand GoBackCommand { get; private set; }
    public DelegateCommand AddToOrderCommand { get; private set; }

    public IEnumerable<Product> Products
    {
        get { return _ProductsRespository.AllProducts(); }
    }

    public ObservableCollection<OrderItem> CurrentOrderItems
    {
        get { return _OrderRepository.CurrentOrderItems; }
    }

    public int SelectedProductId
    {
        get { return _SelectedProductId; }
        set
        {
            SetProperty(ref _SelectedProductId, value);
            AddToOrderCommand.RaiseCanExecuteChanged();
        }
    }

    public int Quantity
    {
        get { return _Quantity; }
        set { SetProperty(ref _Quantity, value); }
    }

    private bool CanAddToOrder()
    {
        var prod = (from p in _ProductsRespository.AllProducts() where
            p.ProductId == _SelectedProductId select p).FirstOrDefault();
        return prod != null;
    }
```

Une fois encore, rien que du classique, tant côté C# qu’en terme de support du pattern MVVM.

Jusqu’ici ce code pourrait être écrit pour WPF avec Mvvm Light, on ne pourrait pas faire la différence. Beauté de C# et XAML et intelligence de Microsoft ne n’avoir pas totalement tué ces merveilles pour la démagogie totale qu’est la programmation en HTML de WinRT ou le retour de la ringardise pour ce qui est du retour de C++ …

QUELQUE CHOSE QUI CLOCHE…

En l’état du code écrit il y a quelque chose qui ne va pas. Essayez et vous verrez…

Si vous vous contentez de lancer l’application, d’aller en page 2 et de saisir quelques articles vous allez penser que tout va bien.

Maintenant prenons un scénario plus réaliste : lancez l’application, allez en page 2, choisissez un article et une quantité, deux données transitoires, puis arrêtez vous là, et dans VS dans la barre “debug location” (qu’il faut ajouter si elle n’est pas présente) vous trouverez un bouton “Suspend”, cliquez dessus, cela ouvre une liste de choix (Suspend, Resume et Suspend and Shutdown), cliquez sur “Suspend and Shutdown”. Cela va suspendre l’application (comme si l’utilisateur était passé à une autre application) et l’arrêter dans la foulée (comme si durant sa suspensation l’OS avait eu besoin de plus de mémoire).

Relancez l’application (F5 ou le bouton de débogue). Par magie l’application va se lancer sur la bonne page, la page de saisie des commandes, c’est déjà pas mal (merci Prism !) mais l’article sélectionné et la quantité saisie auront disparu…

C’est une violation claire des guidlines qui impliquent que la suspension suivie ou non d’un arrêt puis d’une reprise doit être totalement transparent pour l’utilisateur.

PROPRIETES AUTO-SAUVEGARDEES

Prism va venir encore à notre secours grâce à un Attribut qui permet de décorer les propriétés à sauvegarder automatiquement “RestorableState” :

```
[RestorableState]
public int SelectedProductId
{
    get { return _SelectedProductId; }
    set
    {
        SetProperty(ref _SelectedProductId, value);
        AddToOrderCommand.RaiseCanExecuteChanged();
    }
}

[RestorableState]
public int Quantity
{
    get { return _Quantity; }
    set { SetProperty(ref _Quantity, value); }
}
```

Les propriétés du VM ainsi marquées seront sauvegardées et réhydratées automatiquement par Prism.

Rejouez le scénario précédent.

L'application reprend exactement là où elle a été arrêtée, les données transitoires (en cours de saisie) sont intactes, les guidelines Modern UI sont enfin respectées, tout est transparent pour l'utilisateur !

Au passage vous noterez que pour le développeur, et grâce à Prism, cela n'a pas été trop compliqué non plus…

Que se passe-t-il si vous avez besoin de sauvegarder des types complexes ou des objets de la couche Modèle ? Il y a une petite étape à ajouter… En interne le FrameNavigationService de Prism se charge de gérer les états et de sauvegarder / réhydrater les ViewModels, mais il le fait en utilisant le DataContractSerializer de WCF qui a besoin de connaître les types qu'il va manipuler. Pour les types de base c'est automatique (entier, chaînes…), mais pour les types custom comme peut l'être la classe Product dans l'application exemple, il est nécessaire d'indiquer au serializer que ce type est "connu". Cela passe par une ligne de code (une par type complexe à sérialiser) :

```
protected override void OnRegisterKnownTypesForSerialization()
{
    base.OnRegisterKnownTypesForSerialization();
    SessionStateService.RegisterKnownType(typeof(Product));
    SessionStateService.RegisterKnownType(typeof(OrderItem));
    SessionStateService.RegisterKnownType(typeof(OrderItem[]));
}
```

Il faut ainsi surcharger la méthode OnRegisterKnowTypesForSerialization de la classe MvvmAppBase dont hérite l'objet application avec Prism. Le code est très simple puisqu'il suffit d'enregistrer un à un tous les types qui seront gérés par le mécanisme évoqué plus haut.

Maintenant retournez dans l'application, ajoutez quelques articles à votre charriot et recommencez l'exercice de suspension/arrêt.

Relancez.

Parlesangbleu ! Fichtre ! Diantre !

Hmmm, oui. Vous avez raison, vous avez encore quelques problèmes : la liste des objets déjà commandés ne réapparait pas. Il va falloir travailler encore un peu…

ISESSIONSTATESERVICE

Le problème c'est qu'il n'y a pas que les ViewModels qui doivent maintenir leur état dans une application. Les états visuels doivent l'être aussi mais Prism s'en charge déjà (position d'une scrollbar par exemple). Mais je veux parler ici d'informations qui ne sont ni dans l'UI ni dans les VM. C'est le premier M de MVVM… Le Modèle (ou les modèles d'ailleurs).

Dans notre application exemple nous avons une liste de produits qui simule une base de données. Mais nous avons aussi un Repository, une couche modèle, qui gère la commande en cours et son contenu. Si la couche SGBD (simulée ou non) n'est pas concernée par nos soucis de réhydratation, la couche Modèle en contre-partie l'est…

Le singleton qui stocke la commande en cours et autorise l'ajout d'items ne sait pas se sauvegarder tout seul. Et n'appartement pas à la logique d'un VM, Prism ne peut rien pour lui directement.

Heureusement Prism ne nous laisse pas tomber. Il propose un service de maintien de l'état d'une session, SessionStateService, accessible via une Interface. Il suffit d'injecter une dépendance à ce service dans tous les objets du Modèle qui doivent être sauvegardés / restaurés.

Ici nous allons modifier le Repository pour injecter dans son constructeur la dépendance au SessionStateService, exactement comme nous l'avons déjà fait pour d'autres services jusqu'ici.

```
private readonly ISessionStateService _StateService;
private const string CurrentOrderKey = "CurrentOrderKey";
public OrderRepository(ISessionStateService stateService)
{
    _StateService = stateService;
}
```

La méthode d'ajout d'article à la commande en cours va être elle aussi modifiée pour systématiquement sauvegarder la liste interne dans le SessionState :

```
public void AddToOrder(Product product, int quantity)
{
    var orderitem = (from oi in _OrderItems where oi.Product.ProductId
        == product.ProductId select oi).FirstOrDefault();
    if (orderitem == null)
        _OrderItems.Add(new OrderItem { Product = product, Quantity = quantity });
    else orderitem.Quantity += quantity;
    // Write the collection out to the state service
    _StateService.SessionState[CurrentOrderKey] = _OrderItems.ToArray();
}
```

C'est la dernière ligne de code qui change (qui a été ajoutée). Elle prend l'objet liste interne et le transforme en tableau qu'elle ajoute au SessionState à l'aide d'une clé fixe (voir le code juste avant pour sa définition – le SessionState est un dictionnaire clé/valeur).

Toutefois il manque encore un dernier effort : sauvegarder la liste c'est bien, la récupérer c'est mieux… Dans le constructeur du Repository il est maintenant nécessaire d'aller récupérer le tableau des commandes s'il existe :

```
public OrderRepository(ISessionStateService stateService)
{
    _StateService = stateService;
    if (_StateService.SessionState.ContainsKey(CurrentOrderKey))
    {
        IEnumerable<OrderItem> sessionItems =
            _StateService.SessionState[CurrentOrderKey] as IEnumerable<OrderItem>;
        if (sessionItems != null)
        {
            foreach (OrderItem item in sessionItems)
            {
                _OrderItems.Add(item);
            }
        }
    }

}
```

Si le tableau est présent, on le balaye en appelant la méthode d'ajout d'item. Tout ce passera donc comme si l'utilisateur avait lui-même ajouter les articles (très vite!), ce qui garantit que si des calculs sont effectués (dans une véritable application cela serait certainement le cas : prix total commande par exemple) tout sera fait comme si les articles avaient été re-saisis manuellement.

Fini ?

Allez, encore un dernier effort, si c'est vrai cette fois-ci, juré !

Comme nous utilisons un service par injection de dépendance il va falloir gérer cette injection. Comme nous l'avons vu dans les parties précédentes on peut choisir un mode "manuel" ou bien utiliser un conteneur de type Unity. Dans un cas comme dans l'autre il faudra ajouter au moins une fois du code pour gérer la situation.

Dans la classe application, au même endroit où les autres instances de service sont déjà enregistrées (dans le cas de l'utilisation de Unity, ce que je vous conseille par rapport à la version "manuelle") nous devons juste enregistrer le SessionStateService :

```
protected override void OnInitialize(IActivatedEventArgs args)
{
    base.OnInitialize(args);
    _Container.RegisterInstance<ISessionStateService>(SessionStateService);
    ...
}
```

Voilà, c'est vraiment terminé…

Relancez l'application, terminez-là, relancez-là, tout est en ordre. Bravo ! Vous avez fait une application WinRT qui respecte les guidances Modern UI sans trop vous fatiguer grâce à Prism

CONCLUSION

La gestion du cycle de vie des applications avec les OS mobiles est pénible il faut dire la vérité. Comme je le disais quelque part plus haut on a un peu l'impression que tous les OS se sont arrêtés juste avant l'étape "terminé". Tous proposent un mécanisme de message ou de méthodes prévenant l'application des changements d'état. C'est bien gentil mais un peu naïf : ces messages servent justement à persister l'état de l'application alors n'aurait-il pas été plus intelligent que d'aller jusqu'au bout de la démarche en proposant un mécanisme unique dans l'OS pour le faire ?

Pourquoi s'arrêter si près du but ? Mystère. Et tous sont à la même enseigne, d'Android à iOS en passant par Windows 8. A croire qu'un seul a bossé et que deux autres ont servilement copié sans se poser de questions…

Je n'aime pas cette sensation d'œuvre inachevée qu'est la gestion du cycle de vie sous tous les OS mobiles, il y a manquement, travail bâclé.

Heureusement, sous WinRT Microsoft a l'intelligence d'avoir le groupe Patterns & Practices qui comble beaucoup des errements de ce type…

Merci à l'équipe Prism de finir le boulot que les gars de l'OS auraient du faire !

Mais Prism ce n'est pas que ça, c'est encore plein de choses, alors …

Stay Tuned !

Nota: cette série d'article sur WinRT est assez longue et chaque article est lui-même d'une taille dépassant un simple billet de blog. Il doit rester des coquilles, mais à force de relire j'ai les yeux qui n'en peuvent plus, j'espère en l'indulgence des lecteurs ! Ceci n'est pas un livre : c'est Dot.Blog, c'est gratuit et c'est plus complet

Prism pour WinRT – Partie 5 – Communiquer sans se connaitre (L'EventAggregator)

Dans cette 5ème partie sur Prism pour WinRT je vous propose d'aborder le problème des communications à l'intérieur des applications. Elles se doivent de respecter un découplage fort et sont essentielles au bon fonctionnement de l'ensemble, même si elles ne sont pas toujours bien comprises…

DANS LES EPISODES PRECEDENTS…

La partie 1 a posé le décor, la partie 2 a montré la mise en œuvre d'un projet de base utilisant Prism WinRT, la partie 3 a aborder la navigation sous WinRT avec Prism, la partie 4 a traité du délicat mécanisme de gestion du cycle de vie des applications.
Cette approche de Prism pour WinRT se poursuit aujourd'hui avec les communications intra-projet.

L'OMBRE DE PRISM 4

Quand une solution est bonne, pourquoi en changer ?

C'est exactement ce que l'équipe de Prism et les membres du Council se sont dit à propos des communications entre les différentes parties d'une application.

Ainsi, l'EventAggregator de Prism 4 est repris de façon quasi identique, si vous savez utiliser ce dernier alors vous savez utiliser celui de Prism pour WinRT.

Quelques différences existent dans l'implémentation toutefois. Par exemple pour éviter les dépendances avec WPF et SL la partie PubSubEvents a été codée comme une bibliothèque portable, ce qui fait qu'elle est utilisable seule dans tout logiciel utilisant .NET 4.0 et au-dessus.

MESSAGERIE OU AGREGATEUR D'EVENEMENTS ?

En réalité ce dont nous parlons est une messagerie, un service dédié de l'application qui autorise l'envoi et la réception de messages entre parties du programme qui ne se connaissent pas (les types ne se références pas les uns les autres).

Une telle messagerie existe dans la quasi-totalité des framework MVVM. Sous Mvvm Light c'est le Messenger auquel on s'inscrit pour recevoir les messages et qu'on utilise pour les transmettre. Personnellement j'aime cette implémentation simple et très peu contraignante (même si l'implémentation de Mvvm Light n'est pas exempte de défauts). J'avais d'ailleurs proposé lors d'une session du Council qu'on réutilise ce principe qui m'apparait plus compréhensible que l'EventAggregator dont même le nom porte à confusion. Mais l'ombre de Prism 4 rodait tellement au-dessus de nos têtes que la tentation de conserver ce code original (l'un des rares) dans Prism WinRT a été trop fort !

Va pour l'EventAggregator donc !

Pourquoi un nom si bizarre ? "Agrégateur d'évènements".

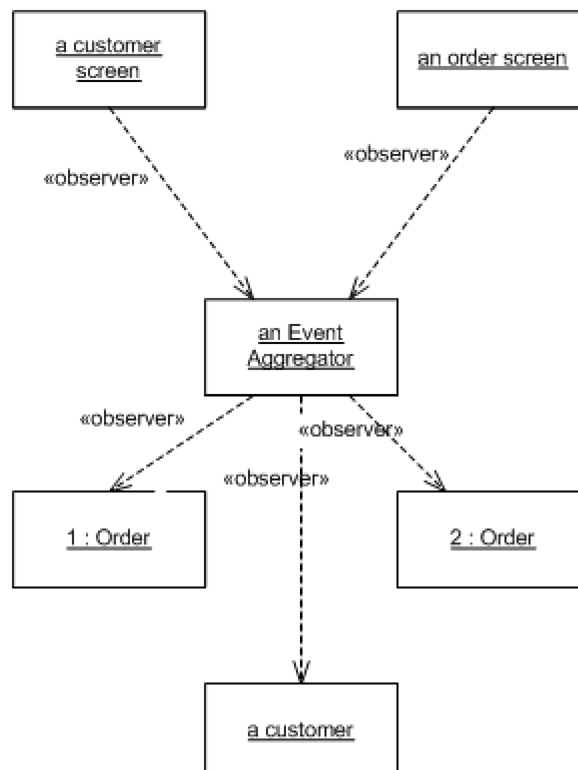
Si on y réfléchit c'est bien ce que fait une messagerie : elle agrège les communications venant de toute part et les distribue à ceux qui en font la demande. Le concept est donc juste, mais "messagerie" aurait été un nom plus clair.

Et d'où vient ce nom ? Pourquoi le terme "messagerie" n'est pas venu en tête directement chez les concepteurs de Prism WPF/SL ? Tout simplement parce que leurs pensées ont suivi une autre voie… Au lieu de partir du concept de messagerie ils sont partis d'un design pattern bien connu qui s'appelle… l'Event Aggregator, décrit par Martin Fowler notamment, l'un des spécialistes des patterns le plus connu avec le Gang Of Four.

L'EVENT AGGREGATOR

J'adore les design patterns, ces petits morceaux de savoir réutilisables à l'infini en connaissant à l'avance les avantages et les inconvénients. Résoudre des problèmes récurrents avec des méthodes éprouvées sans pour cela forcer l'utilisation d'un langage ou d'un OS particulier. C'est génial.

A noter : on parle bien de "*design* pattern", c'est à dire des "patrons de *conception*". Cela n'a rien à voir avec le "design" au sens de conception visuelle plus ou moins artistique.
L'Event Aggregator se définit comme servant à "*Canaliser les évènements en provenance de différents objets dans un objet unique pour simplifier l'enregistrement des clients s'intéressant à ces évènements*".



Dans un système complexe avec de nombreux objets, gérer la communication entre ces derniers peut devenir un enfer, des entrelacs de connexion format un vrai plat de spaghetti ! Maintenir un tel système est une gageüre qui se termine généralement mal.

L'esprit de l'EA est de mettre en place un objet unique (un singleton, autre pattern) vers qui tous les objets sources d'évènement vont envoyer leurs messages. L'EA les centralise, les objets sources ne connaissent que l'EA. Quant aux objets désirant recevoir des notifications il leur suffit de s'abonner à l'EA qui distribuera le courrier en quelque sorte. Ni les sources ni les récepteurs se connaissent et n'ont besoin de le faire. L'EA lui-même est mis en œuvre de telle façon qu'il n'a pas besoin de cette connaissance sur les sources et les récepteurs, il est totalement générique.

Le découplage est donc très fort entre les parties, et c'est exactement ce qu'on cherche à obtenir !

(Pour plus d'informations détaillées sur le pattern Event Aggregator lisez sa fiche sur le site de Fowler)
Si le principe est simple, la mise en œuvre au sein de vos applications peut apparaitre un peu lourde comme nous le verrons plus bas. C'est pourquoi j'avais personnellement milité au sein du Council pour que nous reprenions le principe du Messenger de Mvvm Light au lieu de l'Event Aggregator de Prism. Cette idée n'a pas été retenue et je reste critique sur la complexité de mise en œuvre là où tout aurait pu être bien plus simple. L'E.A. étant une pièce rapportée non obligatoire vous pouvez décider d'utiliser un autre système de messagerie, celui de Mvvm Light par exemple ou écrire le vôtre, pour les besoins les plus courant ce n'est pas si compliqué que cela.

L'EVENTAGGREGATOR EN ACTION

La première étape dans ce type d'exercice est de mettre en place un scénario minimaliste permettant de jouer ensuite avec le code.

L'EA comme toute messagerie sert à échanger des informations entre des parties de programme qui ne doivent pas se connaitre. Souvent ces communications ont lieu entre deux ViewModels ou entre un ViewModel et un Model.

Il est important de bien penser cette communication car pour pratique qu'elle soit elle peut rapidement rendre une application difficile à maintenir. Il convient de la limiter aux exemples cités ci-dessus. Les communications Model/Model ou Infrastructure/ViewModel/Model ou ViewModel/View (via le code-béhind), etc, sont des combinaisons à éviter. Souvent de tels besoins sont les stigmates d'une architecture globale mal pensée… Et si on en arrive à un tel point, mieux vaut réfléchir un instant à comment

corriger l'architecture plutôt que de résoudre le problème à coup de messages dans tous les sens… Conseil d'ami.

Cela étant dit, mais c'était important de le préciser, notre scénario de test sera très simple : nous allons concevoir deux Vues et leurs ViewModels associés que nous allons placer à l'intérieur de la MainPage, côte à côte. Ces deux Vues vont s'échanger des informations sans se connaitre, en passant par l'E.A.

L'exemple est tiré des QuickStarts de Prism WinRT (projet EventAggragatorQuickStart), avec quelques modifications.

Visuellement cela donne :

A gauche le Publisher, l'émetteur, et à droite le Subscriber, le souscripteur.

On notera que tout code peut être à la fois émetteur et récepteur, l'un n'exclut pas l'autre. De même on acceptera l'équivalence "Publisher / Subscriber" pour "émetteur / récepteur". La nuance sémantique existe malgré tout, le fait de publier se rapproche de celui d'émettre mais la notion de "souscripteur" est légèrement différente de celle de "récepteur". Le dernier est plutôt passif, comme un récepteur de radio qui ne fait que tendre une antenne et écouter ce que cette dernière "attrape", alors que le souscripteur, comme un abonné à un journal, a effectué une action volontaire pour écouter, sans cette déclaration de volonté, il n'entendrait rien.

L'EA est donc plutôt basé sur un mécanisme Pub/Sub qu'émetteur / récepteur. Ces précisions n'ont l'air de rien mais elles évitent de mal classer l'information dans votre mémoire 😊

Intégrer Prism

Depuis la mi-mai 2013 Prism pour WinRT est disponible en package Nuget, il est donc très facile d'ajouter ce paquet à un projet pour bénéficier de ses services sans installation complexe. De même le package "Prism.PubSubEvents" est séparé et doit être installé de la même façon, il n'est pas intégré au package de base.

Les communications avec l'EA

Dans un mécanisme Pub/Sub il est nécessaire, comme nous l'avons vu en introduction, d'avoir un "homme du milieu", un tiers qui découple les échanges entre source et récepteur. L'EventAggregator est cet homme du milieu (middleman en anglais). Aucun des tiers n'a besoin de dépendre des autres.

Vu de très haut l'implémentation de l'EA est en fait un simple dictionnaire. Losqu'on s'abonne pour recevoir des messages on ne fait qu'ajouter l'adresse du délégué qui le gèrera dans le dictionnaire de l'EA. Quand on publie un message, l'EA balaye son dictionnaire et appelle les délégués enregistrés pour le type de message en question.

Le code source étant disponible vous pourrez l'étudier pour voir que dans la réalité les choses sont un peu plus complexes, mais le principe est celui d'un dictionnaire.

Dans l'exemple de code que nous allons voir les dépendances seront injectées manuellement (par code) au lieu de passer un conteneur de type Unity, cela simplifie un peu la présentation.

La MainPage et l'injection de dépendance

Comme déjà évoqué elle sera très simple : un conteneur visuel qui contient les deux Vues secondaires (source et récepteur). Le ViewModel se contente ainsi de créer les instances et d'exposer deux propriétés qui sont rendus, côté XAML, par un DataTemplate.

```
public class MainPageViewModel : ViewModel
{
    public MainPageViewModel()
    {
        SubscriberViewModel = new SubscriberViewModel();
        PublisherViewModel = new PublisherViewModel();
    }

    public SubscriberViewModel SubscriberViewModel { get; set; }
    public PublisherViewModel PublisherViewModel { get; set; }
}
```

```
<Page.Resources>
    <DataTemplate x:Key="SubscriberTemplate">
        <views:SubscriberView />
    </DataTemplate>
    <DataTemplate x:Key="PublisherTemplate">
        <views:PublisherView />
    </DataTemplate>
</Page.Resources>
...
<ContentControl Content="{Binding PublisherViewModel}"
                ContentTemplate="{StaticResource PublisherTemplate}"
                HorizontalContentAlignment="Center"
                VerticalContentAlignment="Center" />
<ContentControl Grid.Column="1"
                Content="{Binding SubscriberViewModel}"
                ContentTemplate="{StaticResource SubscriberTemplate}"
                HorizontalContentAlignment="Center"
                VerticalContentAlignment="Center" />
```

De cette façon il devient très simple d'instancier l'EA dans l'objet application et de modifier le routage Vue/VM pour utiliser le constructeur modifié de MainPageViewModel afin d'injecter la dépendance à l'EA :

```
sealed partial class App : MvvmAppBase
{
    IEventAggregator _EventAggregator;

    protected override void OnLaunchApplication(LaunchActivatedEventArgs args)
    {
        NavigationService.Navigate("Main", null);
    }

    protected override void OnInitialize(IActivatedEventArgs args)
    {
        base.OnInitialize(args);
        _EventAggregator = new EventAggregator();
        ViewModelLocator.Register(typeof(MainPage).ToString(),
            () => new MainPageViewModel(_EventAggregator));
    }
}
```

On voit ici la lourdeur que j'évoquais dans le choix de mise en œuvre de l'injection de dépendance avec Prism : si un VM doit utiliser l'EA il doit exposer un constructeur spécial qui devra être appelé de façon spécifique (soit manuellement comme ici, soit par Unity par exemple). Tout cela est fort lourd, une classe de service globale à toute l'application publiant une propriété de type EventAggregator est bien plus simple à gérer et évite de modifier les constructeurs à chaque fois qu'on a besoin d'utiliser un service. Car ici c'est l'EA, mais dans un VM classique il faudra aussi prévoir un constructeur recevant à la fois l'EA et le service de navigation, et si vous créez d'autres services… vos constructeurs vont devenir monstrueux et il ne faudra rien oublier. De plus chaque classe utilisant un service est obligée de déclarer une variable privée pour maintenir le lien vers le service. Autant de services, autant de variables.

Personnellement je trouve cette approche sans véritable intérêt et je lui préfère une classe partagée qui expose en propriété tous les services disponibles. Les constructeurs ne doivent plus être bricolé à chaque ajout ou suppression d'un service utilisé, plus besoin de variables privées, etc. Techniquement cela revient strictement au même sans aucun inconvénient. On créée bien entendu une dépendance à la classe de service, mais dans une application réelle cette dépendance peut être découplée par l'exposition d'une interface de service ce qui règle la question.

Mais continuons (pour voir que cette méthode d'injection est très lourde …).

Tout cela n'est pas suffisant car maintenant il faut aussi modifier le code des VM source et récepteur pour gérer la dépendance au service :

```
public class MainPageViewModel : ViewModel
{
    public MainPageViewModel(IEventAggregator eventAggregator)
    {
        SubscriberViewModel = new SubscriberViewModel(eventAggregator);
        PublisherViewModel = new PublisherViewModel(eventAggregator);
    }

    public SubscriberViewModel SubscriberViewModel { get; set; }
    public PublisherViewModel PublisherViewModel { get; set; }

    ...
}
```

```
public class PublisherViewModel : ViewModel
{
    private readonly IEventAggregator _EventAggregator;

    public PublisherViewModel(IEventAggregator eventAggregator)
    {
        _EventAggregator = eventAggregator;
        ...
    }
    ...
}
public class SubscriberViewModel : ViewModel
{
    private readonly IEventAggregator _EventAggregator;

    public SubscriberViewModel(IEventAggregator eventAggregator)
    {
        _EventAggregator = eventAggregator;
        ...
    }
    ...
}
```

Comme on le voit toutes les classes ayant besoin du service doivent être modifiées pour gérer l'injection de dépendance. La publication d'une classe centrale exposant l'EA serait une solution plus simple. Je vous présente la méthode "officielle" mais je vous incite, selon vos projets et après avoir pesé le pour et le contre à adopter le principe d'une classe centralisatrice des services.

Définir le message à échanger

La communication via l'EA s'effectue par l'échange de messages. Ces messages ne sont rien d'autre que des instances d'une classe transportant les informations.

Prism offre une classe de base générique permettant de définir rapidement de tels messages. Cette classe prend en charge certains aspects techniques de la communication qui sont ainsi cachés au développeur pour plus de simplicité.

Dans l'exemple utilisé les deux VM qui dialoguent s'échangent un message de prise de commande d'un article qui est ajouté dans le charriot de l'utilisateur. La classe est définie comme suit :

```
public class ShoppingCartChangedEvent : PubSubEvent<ShoppingCart> { }
```

L'EA de Prism ne nécessite pas de définir un contenu dans le message, c'est au moment de son envoi qu'on pourra passer l'information à transmettre. Et si le message est une simple notification, ce paramètre pourra donc être passé à null sans problème. L'implémentation

même de l'EA est très bien faite (ma critique exprimée ne porte que sur l'utilisation de l'injection de dépendance qui complique les choses inutilement).

Publier un message

Après toute cette mise en place il est temps d'échanger enfin un message ! Commençons par l'émetteur (ou Publisher).

De la façon la plus simple, il suffit d'écrire une ligne de ce type :

```
_EventAggregator.GetEvent<ShoppingCartChangedEvent>().Publish(_cart);
```

La vairable _EventAggregator est celle récupérée par l'injection de dépendance dans le constructeur de la classe considérée. On Appelle la méthode GetEvent<TMessage>() de l'EA pour obtenir une référence vers un singleton de la classe message (ce travail est effectué par l'EA), puis à la suite on appelle la méthode Publish du message en passant en paramètre les éventuelles données supplémentaires nécessaire à son traitement par le récepteur.

Une fois encore, on peut trouver à redire sur le mécanisme et le côté un peu ésotérique de "GetEvent()". L'EA comme je le disais n'est qu'une option de Prism, on peut utiliser d'autres messageries ou écrire la sienne si on trouve celle de Prism trop biscornue.

L'appel à la ligne d'émission du message se trouve dans le VM de l'émetteur :

```
private void PublishOnUIThread()
{
    AddItemToCart();
    _EventAggregator.GetEvent<ShoppingCartChangedEvent>().Publish(_cart);
}
```

Une fois un nouvel item ajouté au charriot, le message est émis en transférant une référence vers le charriot. Le ou les récepteurs obtiendront deux informations essentielles : 1) qu'un item vient d'être ajouté au charriot, 2) le contenu de la commande incluant le nouvel item.

Ce n'est qu'un exemple et on pourrait penser que le charriot est géré par un Repository central dans l'application (comme nous l'avons vu dans les parties précédentes), auquel cas le message pourrait se contenter de passer "null" en paramètre puisque chaque partie de code de l'application peut à sa guise consulter le Repository. De même le message pourrait être une instance d'une autre classe spécialement conçue pour l'occasion qui transporterait un indicateur d'action (ajout, suppression d'item, vidage de la commande, etc) et l'ID de l'item concerné. En se référant au Repository les récepteurs pourraient alors prendre des décisions plus fines que dans l'exemple où la totalité du charriot est passé en paramètre.

S’abonner à un message

Envoyer des bouteilles à la mer est une chose, les récupérer en est une autre… Arrivés ici nous avons juste conçu une bouteille, l’avons rempli avec un message, puis l’avons jeté dans les vagues…

Pour attraper le message il faut qu’un code s’abonne à l’EA pour ce type de message.

```
public SubscriberViewModel(IEventAggregator eventAggregator)
{
    _EventAggregator = eventAggregator;
    ...
    _EventAggregator.GetEvent<ShoppingCartChangedEvent>().Subscribe(HandleShoppingCartUpdate);
}
```

C’est généralement dans le constructeur que cette déclaration d’abonnement est effectuée comme le montre le code ci-dessus. On peut répondre au message en codant directement une expression lambda ou un délégué, on peut aussi, et c’est souvent plus “propre”, déclarer une méthode privée qui sera chargée de traiter le message. C’est le cas ici.

Bien entendu dans des conditions plus complexes un récepteur peut être à l’écoute qu’à certains moments et pas forcément durant tout son cycle de vie. Il est possible de s’abonner à un message ailleurs que dans le constructeur de même qu’il est possible de désabonner.

```
private void HandleShoppingCartUpdate(ShoppingCart cart)
{
    ItemsInCart = cart.Count;
}
```

La méthode du récepteur qui gère le message est ici fort simple (ci-dessus) puisqu’elle ne fait que récupérer le nombre d’items dans le charriot, propriété publiée par le VM et affichée par la Vue du récepteur.

S’ABONNER EN TACHE DE FOND

Il n’y a pas que les VM qui peuvent être source ou récepteur de messages, tout code peut participer à la farandole des messages ! Mais faites attention de ne pas en abuser !

L’exemple que nous regardons aujourd’hui est tiré de ceux fournis avec le code de Prism. Il est complété par une démonstration d’un souscripteur en tâche de fond (backgroundSubscriber.cs). Ce code ne montre rien de spécial concernant l’EA que nous n’ayons déjà vu mais il insiste sur certains points spécifiques à la situation décrite, notamment la nécessité de conserver quelque part une référence sur l’instance du récepteur pour éviter qu’il soit collecté par le GC, Prism et l’EA utilisant des WeakReference.

Je vous invite à prendre connaissance de ce code un peu plus complexe mais très important si vous devez gérer la messagerie ailleurs que dans des VM.

THREADS ET EA

l'EA de Prism prend en charge un aspect important des communications : le thread utilisé pour échanger les messages.

Lorsqu'on publie un message de façon simple il n'est pas assuré du tout qu'il soit véhiculé sur le thread d'UI, de fait le récepteur traitera la réponse dans ce thread et s'il doit mettre à jour l'UI cela se soldera par une exception… On en peut que passer par le thread d'UI pour toucher à l'UI.

Si vous déclenchez le message de la façon suivante :

```
private void PublishOnBackgroundThread()
{
    AddItemToCart();
    Task.Factory.StartNew(() =>
        {
            Debug.WriteLine(String.Format("Publishing from thread: {0}",
                Environment.CurrentManagedThreadId));
            _EventAggregator.GetEvent<ShoppingCartChangedEvent>().Publish(_cart);
        });
}
```

… Vous obtiendrez une exception dans le récepteur au moment de la mise à jour de l'UI.

Ici cela parait assez simple puisque c'est volontairement que nous utilisez un thread secondaire pour publier le message. Mais dans une application réelle cette situation pourra se produire de façon moins "grossière" ce qui débouchera sur des bogues aléatoires parfois difficiles à découvrir.

Heureusement la méthode de réception d'un message prend en charge cet aspect des choses. Si votre message ne concerne que des classes qui n'ont rien à voir avec l'UI, il n'y a pas de problème, mais si le message doit mettre à jour l'UI alors il est important de le préciser au moment de l'abonnement. Bien entendu ce n'est pas l'émetteur qui est concerné, il ne peut pas deviner comment le message qu'il envoie sera utilisé. C'est donc bien celui qui reçoit et traite le message qui doit faire attention au thread de traitement :

```
_EventAggregator.GetEvent<ShoppingCartChangedEvent>().Subscribe(
    HandleShoppingCartUpdate, ThreadOption.UIThread);
```

La méthode Subscribe accèpte un paramètre supplémentaire permettant de préciser, comme ici par exemple, qu'on désire que le message soit traité sur le thread d'UI.

Dès lors le code précédent (qui envoie le message depuis un thread secondaire) ne pose plus de problème…

Mais il faut faire faire attention à un dernier détail : si on désire utiliser cette option, il faut s'assurer que l'instance du service de l'EA a elle-même bien été créée dans le thread de l'UI !

On retrouve ces contraintes dans Mvvm Light et dans tous les frameworks MVVM. Mvvm Light propose une classe dispatcher qui permet de s'assurer qu'on agit bien sur le thread d'UI. C'est peut être plus simple.

FILTRAGE

Il arrive souvent qu'un récepteur doivent filtrer les messages, le type du message n'est pas toujours suffisant pour différencier certains cas qui réclament, ou non, de traiter l'information.

On peut se base sur des sous-classes et créer des arborescences de messages ultra spécialisées. Cela devient vite du code spaghetti.

Trouver le juste milieu implique qu'il y a des cas où un filtrage sera nécessaire. La méthode d'abonnement de l'EA autorise le passage d'une expression Func<T,bool> qui assurera le filtrage avant traitement. Cela est très pratique :

```
_EventAggregator.GetEvent<ShoppingCartChangedEvent>().Subscribe(
      HandleShoppingCartUpdateFiltered,
      ThreadOption.UIThread, false, (cart) => cart.Count > 10);
```

Le code ci-dessus montre l'utilisation d'une expression filtrant les messages en ne laissant passer que ceux dont la commande possède plus de 10 articles.

CONCLUSION

L'Event Aggregator de Prism est un mécanisme bien rodé qui permet d'assurer la fluidité des communications entre les différentes parties d'une application en garantissant un découplage fort entre ces dernières.

L'implémentation n'est pas exempte de défauts ou de lourdeurs, certaines peuvent être contournées en évitant de faire de l'injection de dépendance pour le plaisir et sans véritable motivation technique, d'autres sont intrinsèques à Prism. Dans le cas où on souhaiterait s'affranchir de ces lourdeurs il peut être intéressant de comparer les services rendus par d'autres frameworks.

Prism pour WinRT introduit en revanche des aides précieuses spécifiques à cet environnement, notamment la conservation des états et la navigation.

Le framework parfait pour WinRT reste à écrire, si vous le sentez, c'est les vacances, un bon moyen de devenir célèbre en ne bronzant pas idiot !

Prism pour WinRT–Adresses utiles

Avant de passer à la partie 2 de la présentation de Prism pour WinRT je tenais à vous confier de nouvelles adresses de téléchargement pour le source et le documentation.

PRISM POUR WINRT – SOURCE ET DOCUMENTATION A JOUR

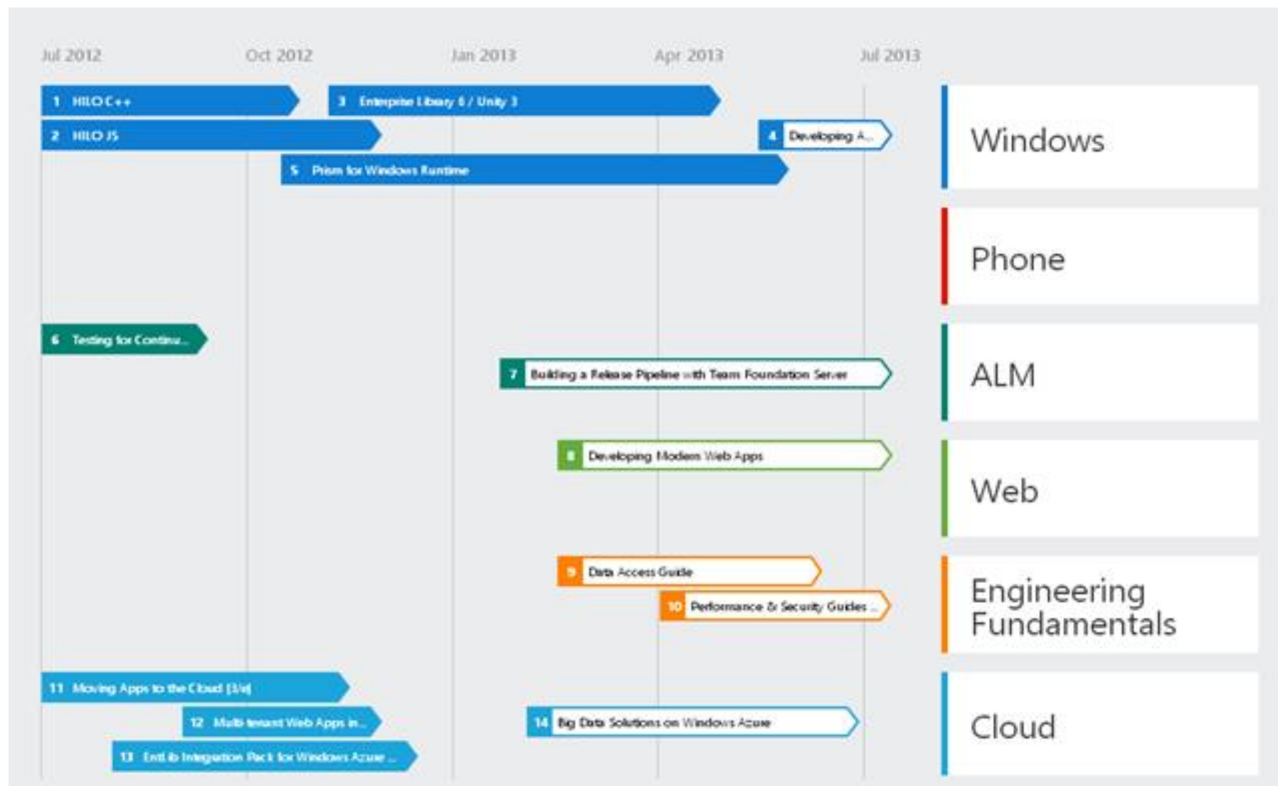
S'il existe bien une version diffusée sur CodePlex dont je vous donnais l'adresse dans la Partie 1, l'équipe de Prism a aussi mis à disposition le source et la documentation sur le Windows Dev Center. Des package Nuget sont prêts (ou devraient l'être quand j'écris ces lignes) et peuvent être téléchargés depuis Visual Studio.
Si vous souhaitez disposer des dernières sources suivez ce lien :

http://code.msdn.microsoft.com/windowsapps/Prism-for-the-Windows-86b8fb72
Si vous souhaitez accéder à la documentation de la guidance en PDF c'est ici (+de 210 pages pour la version de mai 2013) :

http://www.microsoft.com/en-us/download/details.aspx?id=39042
Vous pouvez aussi accéder à la documentation en ligne ici :

http://msdn.microsoft.com/en-us/library/windows/apps/xx130643.aspx
Un endroit où jeter un oeil n'est jamais du temps perdu, le blog de Blaine Wastell, membre principal de l'équipe de Prism pour WinRT :

http://blogs.msdn.com/b/blaine/
A bientôt pour la partie 2 de notre cycle de découverte de celle nouvelle guidance du groupe Patterns & Practices de Microsoft qui publie des tas de choses merveilleuses dont voici un bref rappel en image :



Prism pour WinRT : une application exemple

Je vous ai présenté longuement “Kona” qui s’appelle désormais Prism pour Windows Store apps, et pour compléter cette série voici aujourd’hui une application exemple assez représentative dont le source est disponible sur GitHub pour l’étudier en détail.

ITINERARY HUNTER, LE CHASSEUR D’ITINERAIRE

Itenerary Hunter est un site web (http://itineraryhunter.com/) destiné aux personnes souhaitant se créer un itinéraire touristique par eux-mêmes. Le “slogan” étant d’ailleurs “*ready made self travel*”.
Vous allez me dire un site web ? WinRT ce n’est pas le web !

Si vous avez bien suivi toutes mes présentations de WinRT et surtout de Kona/Prism pour WinRT, vous devez vous rappeler que l’application exemple fournie avec les guidances ressemble à une application Web avec un caddy. Et donc vous vous rappelez certainement ce que j’en avais dit puisqu’ayant participé à l’élaboration de Prism pour WinRT j’avais tenté, infructueusement, de faire changer cet exemple pour quelque chose de plus “entreprise” et moins “web”…

Techniquement Prism pour WinRT n’est pas destiné à faire des apps qui ressemblent à sites Web, mais dans la pratique c’est plutôt WinRT lui-même qui par sa structure, son mode fullscreen, et ses autres caractéristiques fait que les applications qui collent bien sont plutôt celles qui ressemblent à des sites Web… Peut-être cela n’est-il lié qu’à un manque

d'imagination des développeurs en entreprise, mais en tout cas cela explique certainement la faible adoption de WinRT par les entreprises. Trop grand public, même dans ces exemples très techniques et pas assez utilisateur "pro".

Avec le temps et les Surface 2 et Windows 8.1 est-ce que le "rouleau compresseur" Microsoft arrivera à imposer petit à petit WinRT en entreprise ou bien cela en restera-t-il au niveau actuel, c'est un mystère dont la réponse se trouve chez les DSI…

Quoi qu'il en soit, ce nouvel exemple, très bien développé, est lui aussi basé sur un site Web existant. A chacun de voir si le résultat obtenu justifie le développement d'une application spécifique pour un système relativement peu utilisé ou bien si la version Web n'est pas suffisante et d'emblée accessible à tout le monde. Cela peut aussi être une stratégie, la version WinRT pouvant s'avérer mieux conçue dans une logique d'UX pour tablette Surface… En créant son propre écosystème WinRT peut se justifier lui-même.

Bref, l'exemple est programmé en faisant appel aux dernières technologies Microsoft, donc WinRT avec son look Modern UI et en respectant les guidances du groupe Patterns & Practices qui se concrétisent dans Prism pour Windows Store apps. Cela en fait ainsi un excellent support pour étudier de plus près la mise en œuvre de Prism sous WinRT et complète à merveille ma série sur ce sujet !

L'application a été créée par Steven Hollidge dont la page G+ est la suivante : https://plus.google.com/101698469376330546143

UNE BONNE UTILISATION DES GUIDANCES

L'application montre différents aspects de la programmation sous WinRT et notamment le respect des guidances de Prism pour WinRT.

On y trouve aussi bien une présentation soignée et le respect d'une UI de type Modern UI (avec le zoom sémantique notamment) que l'utilisation des ViewModels, des convertisseurs de valeurs, des services et même des tests unitaires.

LE CODE SOURCE

Steven a eu la bonne idée de publier le code source de son application sur GitHub et c'est ce qui mérite la présentation ici de cette dernière. En téléchargeant ce code vous pourrez à loisir étudier un exemple supplémentaire d'utilisation de Prism pour WinRT.

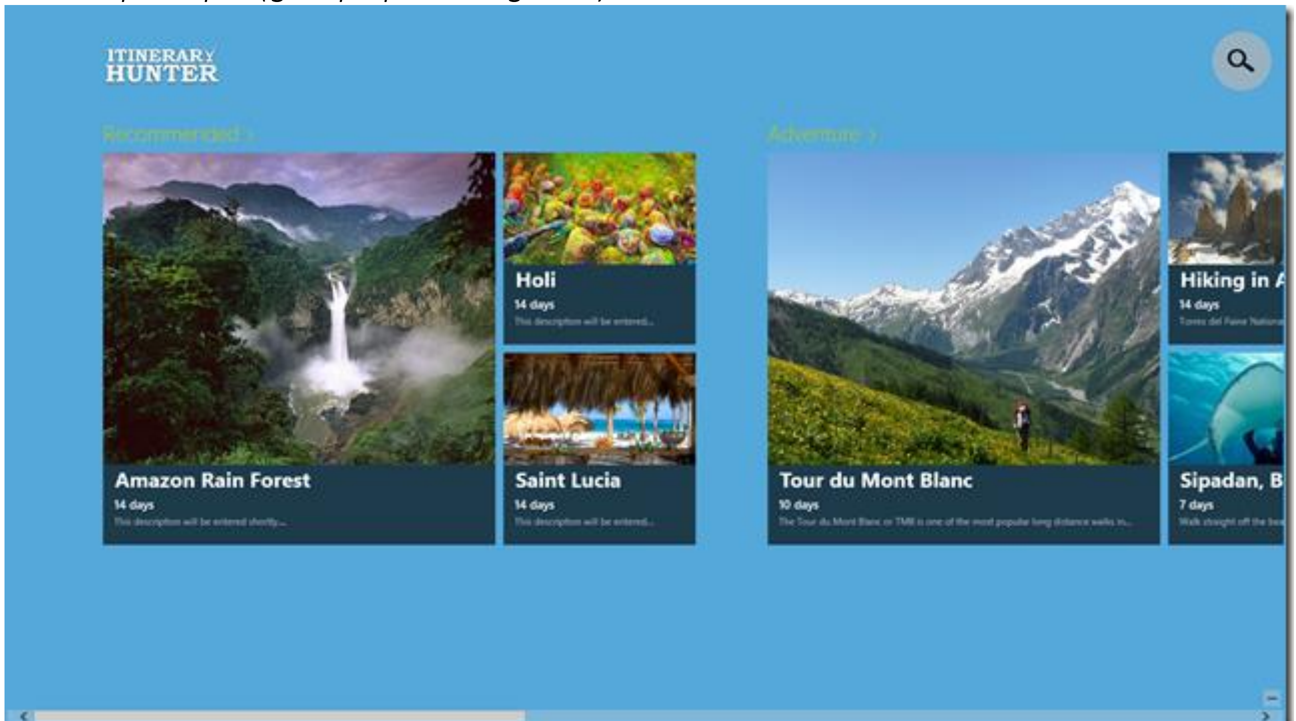
Le code se trouve à cet endroit : https://github.com/stevenh77/ItineraryHunter-Win8

QUELQUES COPIES D'ECRAN

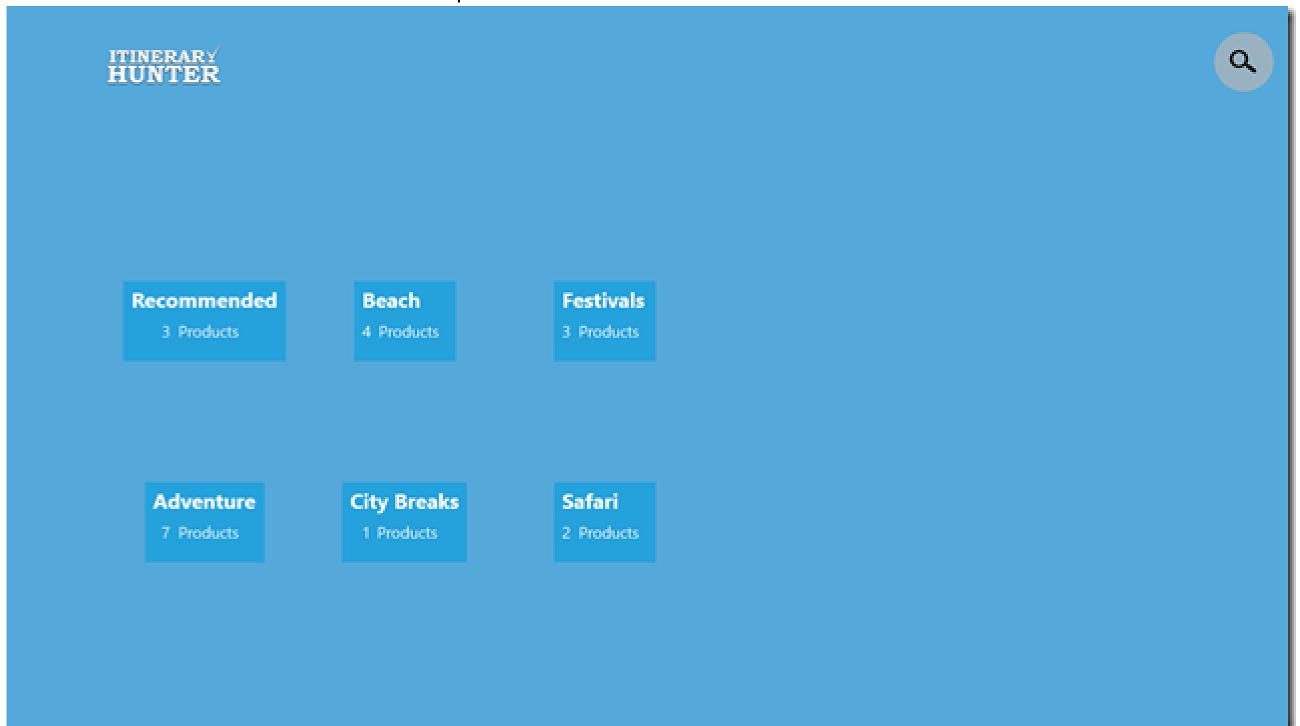
Pour se faire une meilleure idée du style de l'application et de ses fonctions, voici quelques captures d'écran.

Le splash

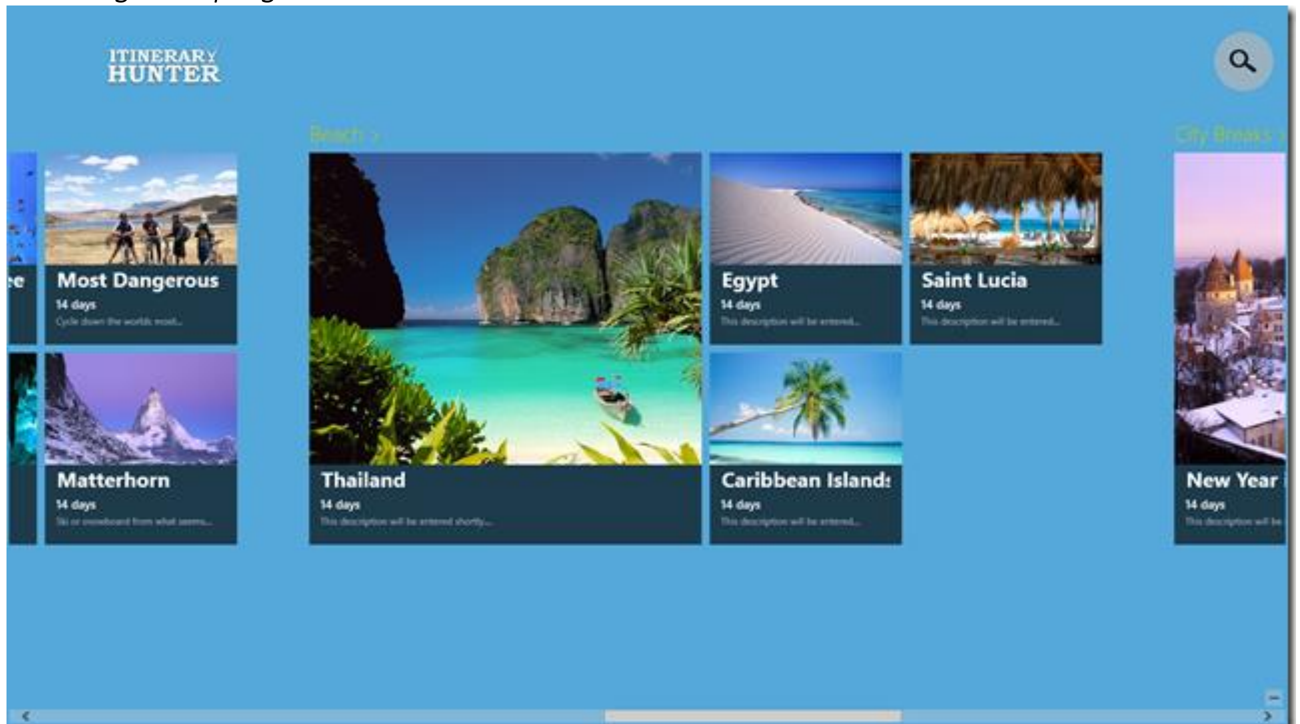
L'écran principal (groupé par catégories)



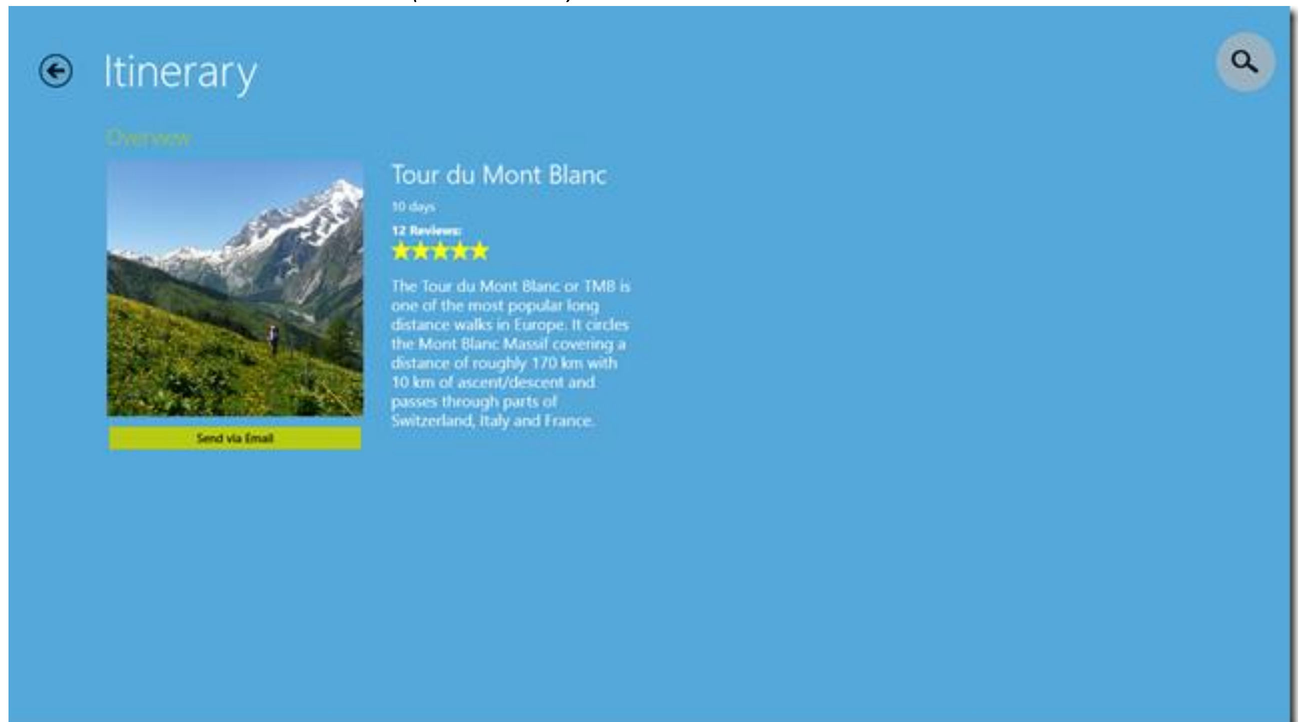
L'utilisation du Zoom sémantique



La catégorie "plage"



La sélection d’un itinéraire (vue détail)



CONCLUSION

Cette application n’est pas parfaite, c’est juste un exemple, mais elle fonctionne sur une base réelle et utilise de nombreuses techniques spécifiques à WinRT, dont le support de Prism.

Justement, pas trop complexe non plus, elle permet une étude de son code source sans se perdre dans une énorme application.

Le fait qu’il s’agisse encore d’un exemple copiant un comportement Web ne plaide pas pour WinRT en entreprise c’est sûr. Mais je reste convaincu malgré tout que de nombreuses applications dans ce contexte peuvent tirer un énorme avantage d’une mise en œuvre sous WinRT : bornes informatives, catalogue pour les clients, outil de prise de commande pour les représentants en visite, etc. Si WinRT n’est pas une panacée et ne remplacera peut-être jamais le bureau classique pour tout un tas d’applications, il serait étonnant que la force et l’énergie que Microsoft déploie pour imposer cette plateforme reste absolument sans effet.

Et puis WinRT se programme surtout en Xaml et C#, et quand on voit l’indigence des autres plateformes mobiles en terme de programmation et de création d’UI, on se dit que ça serait pas mal quand même que cela fasse un succès. Honnêtement je préfère largement faire du WinRT que développer des interfaces en XML Android ou pour le Web où chaque personnalisation se solde par la création de 4 images bitmap, des ninepatches, dans plusieurs résolutions pour différentes densités, etc. Vive la création d’UI en mode vectoriel ce qui est unique au monde et seulement chez Microsoft avec XAML, et si nous n’avons plus

beaucoup d’espoirs pour Silverlight, au moins que vive WinRT (et WPF), tout le reste est préhistorique !

Avertissements

L'ensemble des textes proposés ici sont issus du blog « Dot.Blog » écrit par Olivier Dahan et produit par la société E-Naxos.

Les billets ont été collectés fin septembre 2013 pour les regrouper par thème et les transformer en document PDF cela pour en rendre ainsi l'accès plus facile.

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.

Ce reccueil 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 C# existera…

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 à 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, notammant 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 !