



Formations .NET – Audit, Conseil, Développement, UX

Articles et Livres Blancs gratuits à télécharger www.e-naxos.com

Dot.Blog, le blog www.e-naxos.com/blog

© Copyright 2011 Olivier DAHAN

MICROSOFT MVP Silverlight 2011, MVP Client App Dev 2010, MVP C# 2009



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

LE LIVRE BLANC DE JOUNCE MVVM et MEF avec Silverlight 4+

[Applications modulaires suivant MVVM]

Version 1.0 Septembre 2011

Sommaire

Code Source.....	5
Préambule	6
MEF.....	6
MVVM.....	7
Jounce vs MVVM Light	7
Différences et similitudes.....	8
Pourquoi utiliser MEF	10
Quel choix au final ?	10
Que propose Jounce ?	11
La Librairie	12
Présentation	12
Le code	13
Le Tree Map et les espaces de noms.....	13
L'organisation du code	15
Graphe des dépendances.....	17
L'organisation des espaces de noms	19
Les bases.....	20
L'architecture globale d'un projet.....	21
Les références à MEF.....	21
La référence à Jounce	22
App.xaml, Application Service, et messagerie	22
La MainPage. Connexion au ViewModel et Blendabilité.	25
MainPage.xaml	25
Le support des données de conception	26
Le code de MainPage	30
MainPage.Xaml.....	30
MainPage.Xaml.cs	31
L'interface du ViewModel	31
Le ViewModel « réel ».....	32
Le ViewModel de design	32
Point intermédiaire	33
L'essentiel sur	33
L'Application Service	33

Marquage d'une Vue.....	35
Marquage d'un ViewModel.....	36
BaseViewModel, BaseNotify et IViewModel.....	36
ViewModelRouter	40
Créer des routes	44
Point Intermédiaire	44
BaseEntityViewModel	45
L'exemple BaseEntityVM.....	46
Le résultat final.....	46
Le code, généralités.....	47
L'interface du ViewModel	48
Le ViewModel de conception	48
Le ViewModel de runtime	49
La région IMainViewModel	49
La région des validations	51
La région du constructeur	54
La région du code privé	55
La Vue – Code-behind.....	56
La Vue – Xaml	56
Point Intermédiaire	57
Les Commandes.....	58
L'exemple du clic sur un Rectangle	61
L'exemple du Slider	63
ActionCommand.....	66
Le code de l'exemple.....	69
Point intermédiaire	73
La messagerie EventAggregator	73
L'exemple	75
Le visuel	75
Le code	76
L'émetteur	77
Le récepteur	77
Navigation simplifiée : Event Aggregator et ViewRouter.....	79
Naviguer ?.....	79

ViewRouter.....	80
Le mécanisme.....	80
L'exemple	82
Les régions.....	82
L'exemple	83
Chargement de XAP.....	86
Logger personnalisé	88
Créer un Logger personnalisé.....	89
Workflows	94
Qu'est qu'un Workflow Jounce ?	95
L'exemple	96
Le visuel	96
Le code	97
VSM Aggregator et GotoVisualState	99
GotoVisualState.....	101
Exemple	102
Visual State Aggregator (VSA)	104
Conclusion	108

Table des Figures et des Tableaux

Figure 1 - Tree Map des namespaces (en IL)	14
Figure 2 - le tree map (en nombre de classes).....	15
Figure 3 - L'oganisation des sources de Jounce	16
Figure 4 - Graphe des dépendances	18
Figure 5 - Le template Jounce.....	20
Figure 6 - L'arbre du projet template	21
Figure 7 - MainPage par défaut	26
Figure 8 - Le mécanisme des données de conception.....	28
Figure 9 - L'application BaseEntityVM en cours d'utilisation	47
Figure 10 - InvokeCommandAction et le Slider	66
Figure 11 - ActionCommand. Affichage de l'application	67
Figure 12 - ActionCommand. Sélection d'une personne.....	68
Figure 13 - ActionCommand. Le bouton a été cliqué	68
Figure 14 - Event Aggregator - Envoi de message.....	75
Figure 15 - Event Aggregator - Exception non gérée.....	76
Figure 16 - Logger personnalisé	93
Figure 17 - Workflow.....	96
Figure 18 - Mr Smiley est content !	102
Figure 19 - Mr Smiley est triste !	103
Figure 20 - VSA – Panel A (B désactivé).....	104
Figure 21 - VSA - Panel B activé.....	104
 Tableau 1 - Similitudes et Différences entre Jounce et MVVM Light.....	 8

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	<code>ODJounceSamples.sln</code>
Exemple1	<code>BasicJounceApp</code>
Exemple2	<code>BaseEntityVM</code>
Exemple3	<code>SLInvokeCommand</code> (dans une solution à part, n'utilise pas Jounce)
Exemple4	<code>CommandDemo</code>
Exemple5	<code>EventAggregator</code>
Exemple6	<code>SimpleNavigation</code>
Exemple7	<code>SimpleNavitationWithRegion</code>
Exemple8	<code>CustomLogger</code>
Exemple9	<code>Workflow</code>
Exemple10	<code>GotoVisualState</code>
Exemple11	<code>VSMAggregator</code>

Préambule

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 connaît 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 dresser 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 apparaît 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'essaie 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 CRUD¹ 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.

¹ Create, Read, Update, Delete, les quatre opérations de base pour la persistance des données.

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 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 décortilage ! 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'utiliser 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 crée 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 1 - Tree Map des namespaces (en IL)

On voit nettement que les parties **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 :

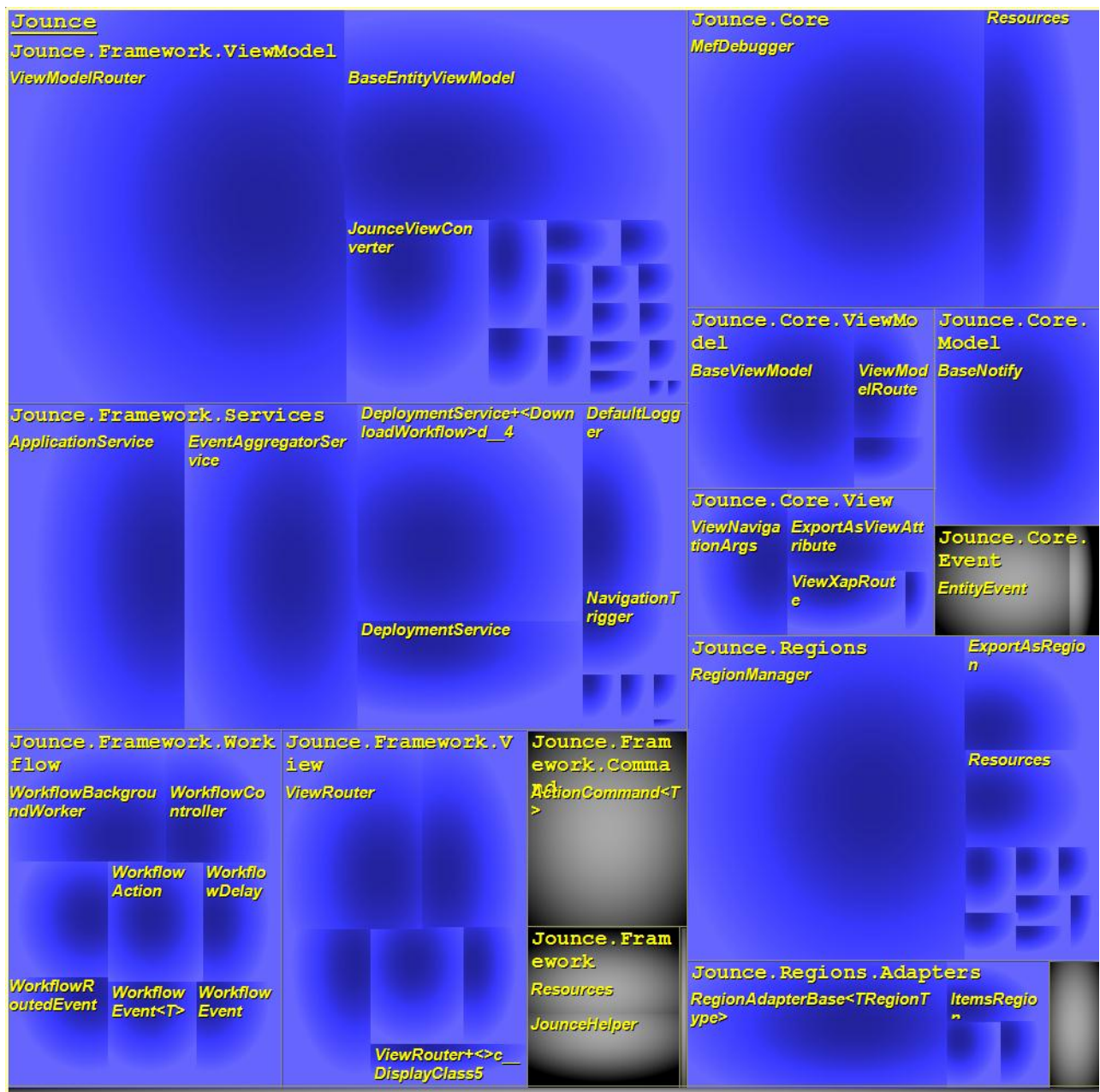


Figure 2 - le tree map (en nombre de classes)

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 :

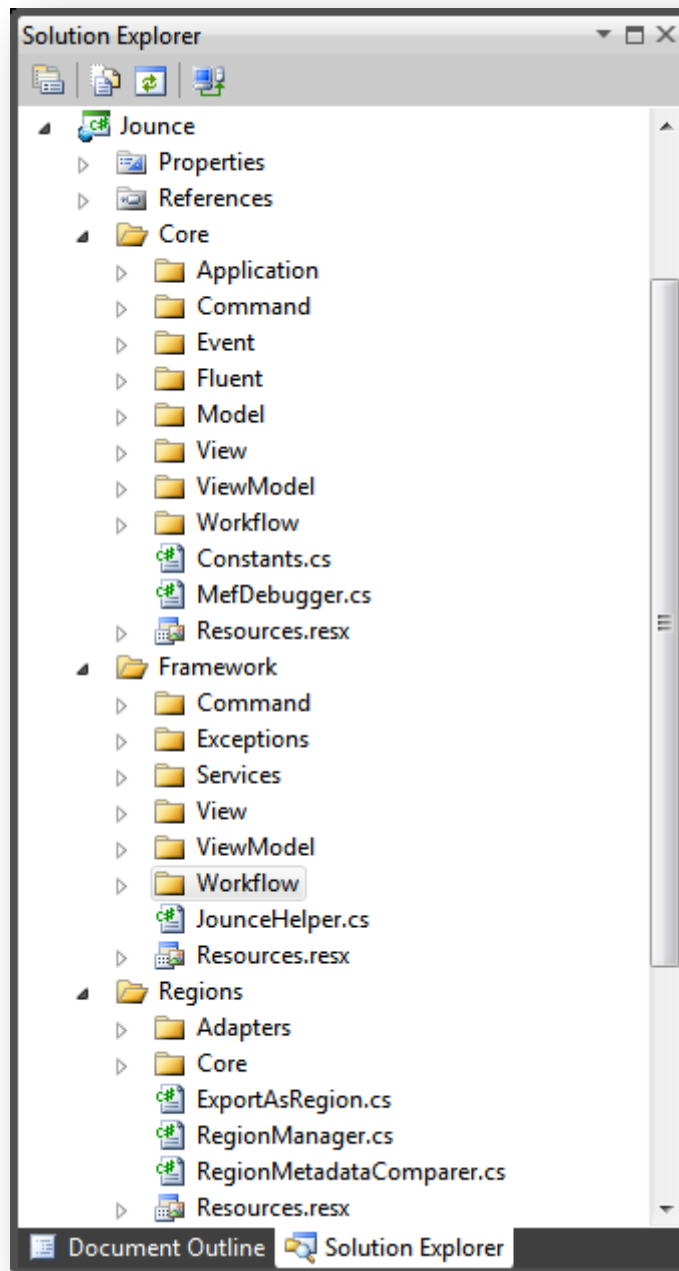


Figure 3 - L'organisation des sources de Jounce

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 **Jounce.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.

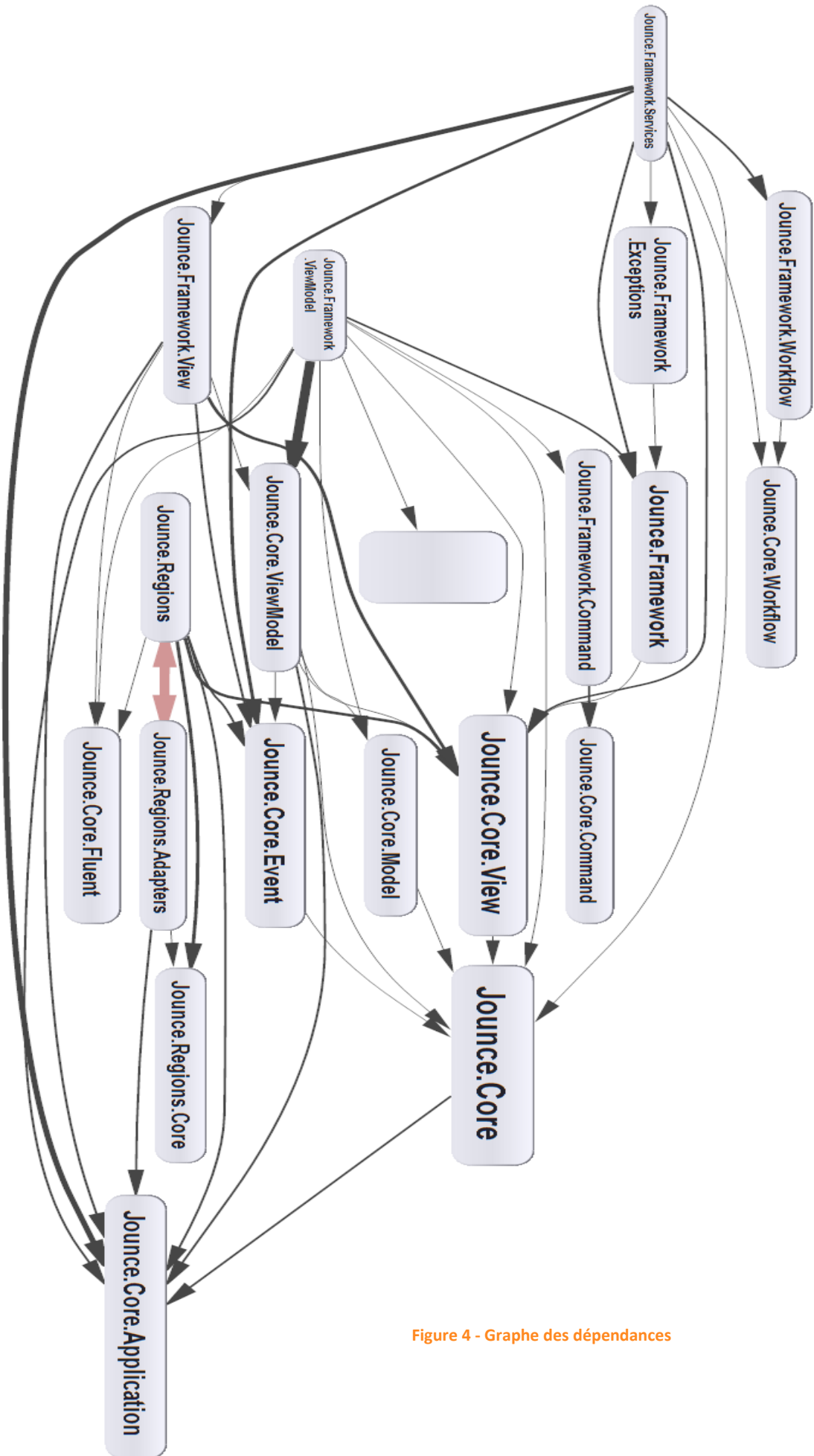


Figure 4 - Graphe des dépendances

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 apparaît sous « **Visual C#** » et non sous « Silverlight » comme le montre la capture ci-dessous (figure 5).

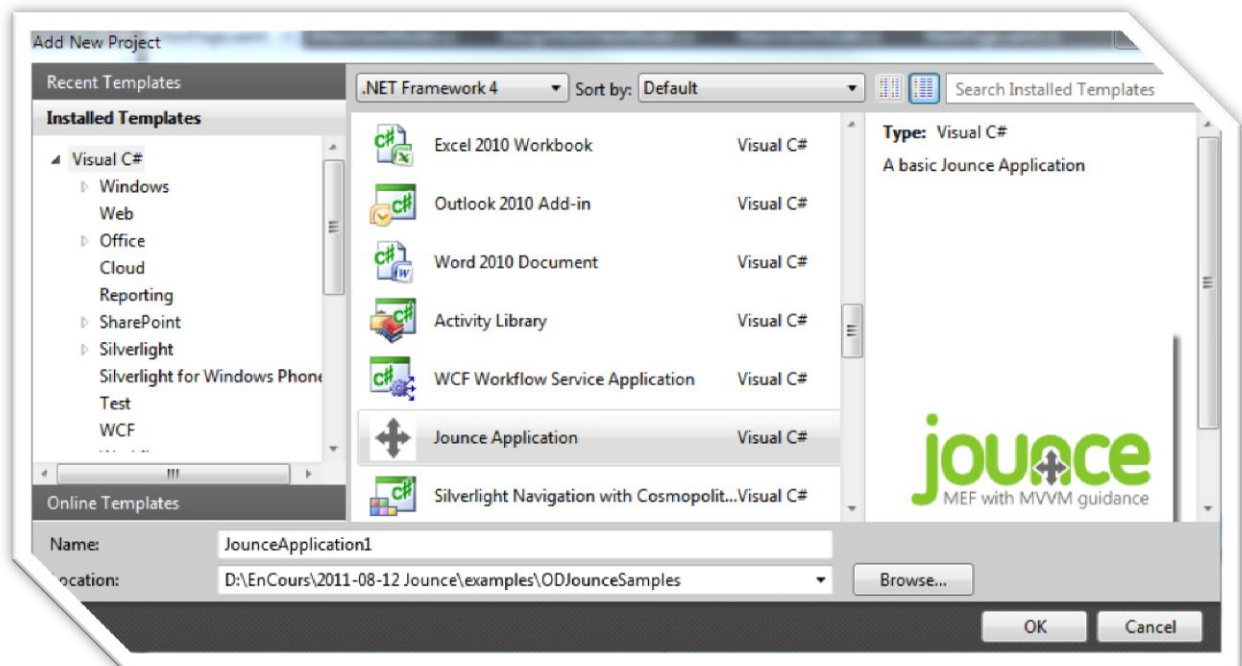


Figure 5 - Le template Jounce

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):

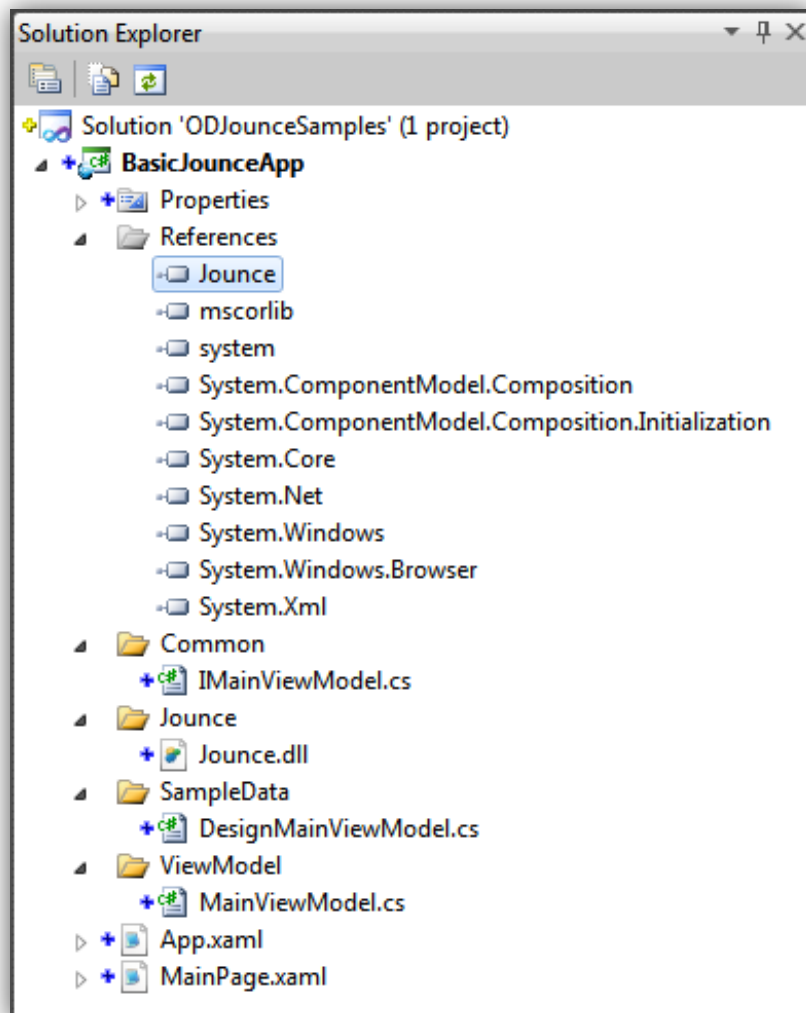


Figure 6 - L'arbre du projet template

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 `UnhandledExceptionEvent`. 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'*EventAggregator* (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) :

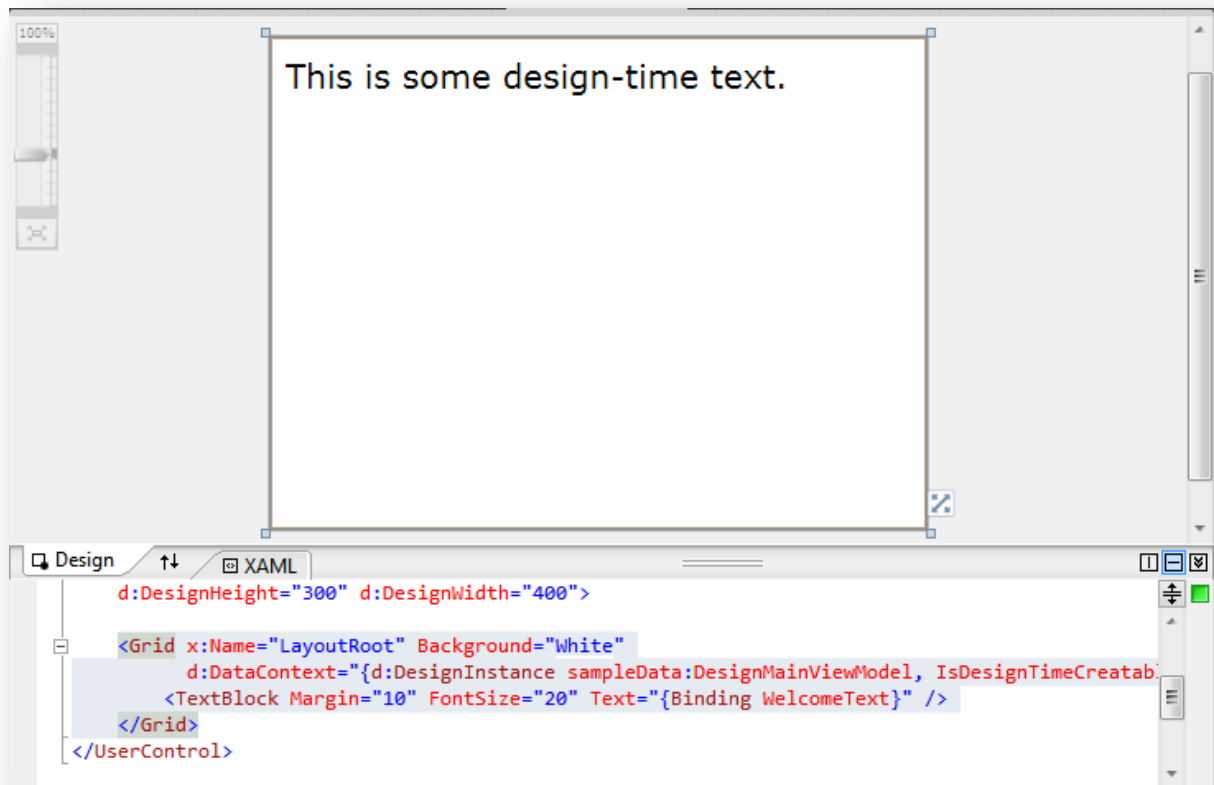


Figure 7 - MainPage par défaut

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.

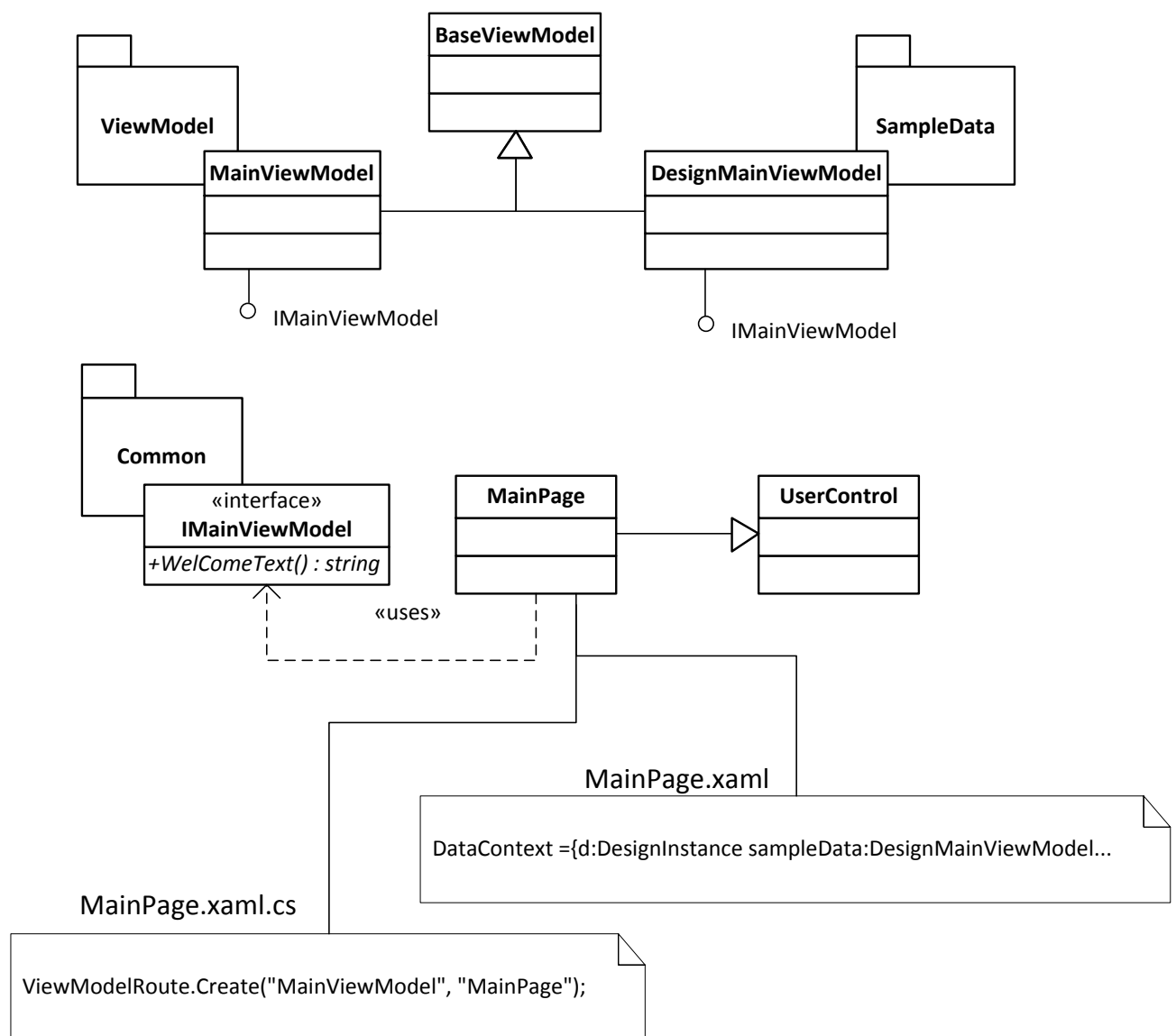


Figure 8 - Le mécanisme des données de conception

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 crée 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 `PropertyChanged`).

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 **WelcomeText**. 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 pas 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 précisons 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 22). 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 débogage ;
- 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 débogage 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 crée 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 crée 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 préférera 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 *overridees* 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 hérissent. 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 désavantages 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 crée 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 crée 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 crée 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 (puisque'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 « entreprise ». 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 crée 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 *override*) ;
- 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 *override* ;
- 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 Tooltips 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 9 - 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
    {
        ICommand 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 saurais 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 ICommand 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'utilisons 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 crée 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 crée 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 préférera, 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 débogage.

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'apparaît 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 ajoutée, 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 sophistication 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 `OnCommited()`. Elle fait partie des méthodes de la classe mère et on l'utilise par un *override*. `OnCommited()` est appelée automatiquement par la classe mère quand la commande de 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 événements 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 événements fait partie « du décor », de la panoplie de base du développeur.

La classe **Button** expose un événement 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 événements est que cela crée 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'événement, 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'événement **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 événement `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 événements, 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'événements 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 événements 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 événements 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 événement et spécialisée pour le type des arguments de l'événement.

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 événement 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 événements 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 événement, 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 événements qui ne sont pas accessibles 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**
 - o Le constructeur de cette commande qui reçoit en paramètre le ViewModel
 - o Sa méthode **Execute()** qui modifie la propriété **RectangleOpacity** du ViewModel
 - o 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**
 - o Le binding de la commande sur **SliderValueCommand** du ViewModel
 - o 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 10 - 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 :

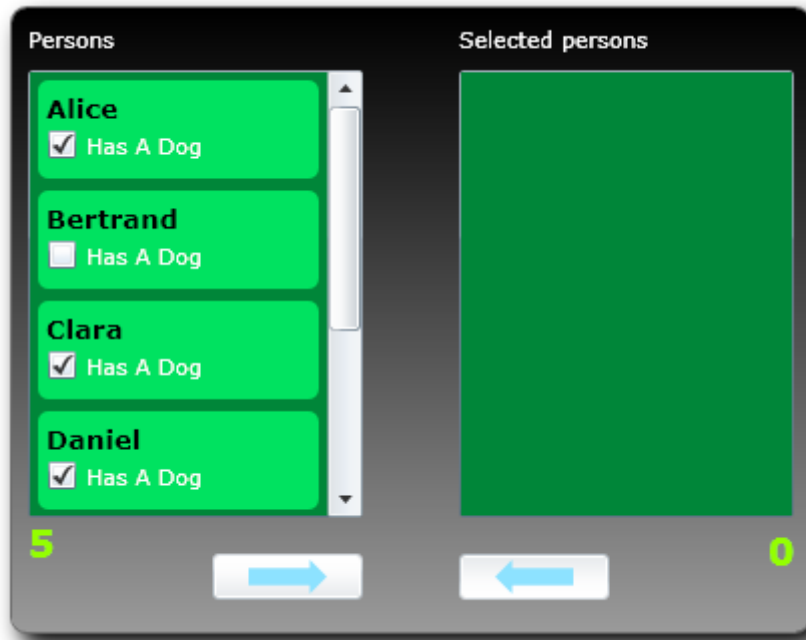


Figure 11 - **ActionCommand**. Affichage de l'application

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).

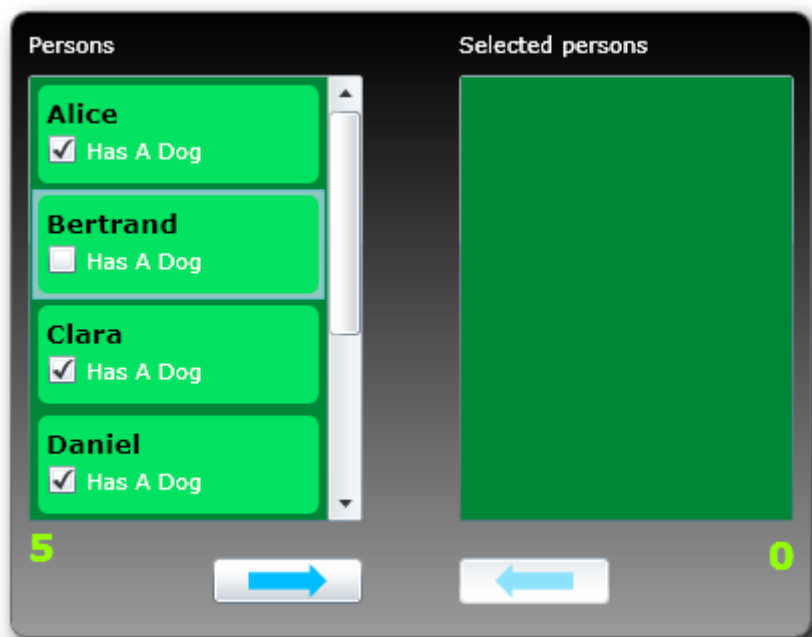


Figure 12 - ActionCommand. Sélection d'une personne

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.

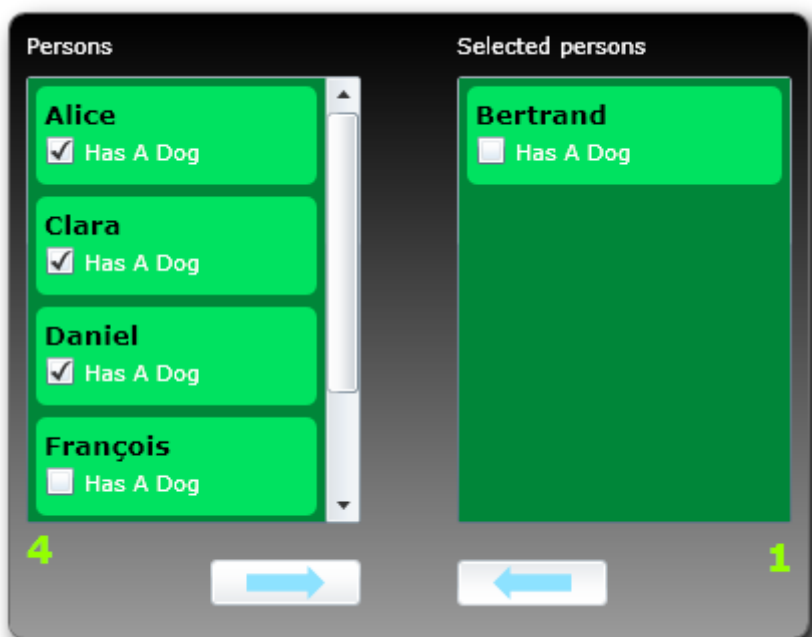


Figure 13 - ActionCommand. Le bouton a été cliqué

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
 - o La classe **Person** est définie. Elle hérite de **BaseNotify** de Jounce ce qui simplifie le support de **PropertyChanged**.
 - o 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.
 - o Elle prévoit aussi deux propriétés retournant le nombre d'éléments de chaque liste.
 - o 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 `Override` 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-la.

- 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 rafraîchir 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*² 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

² « Vanité des vanités, et tout est vanité ! », l'Ecclésiaste est un livre de la Bible hébraïque.

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.

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 complexité³.

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 »

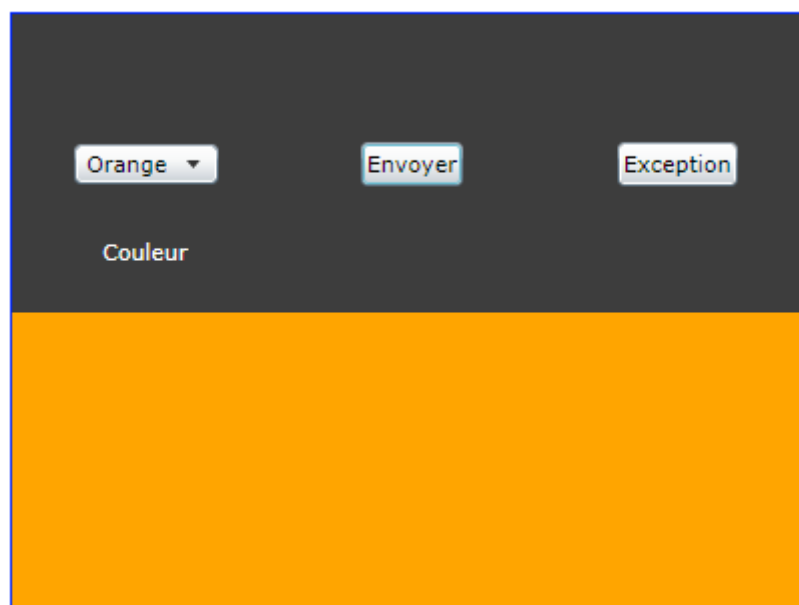


Figure 14 - Event Aggregator - Envoi de message

³ On trouve d'ailleurs de nombreux messages ou billets sur le Web qui abordent ce sujet épineux.

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** apparaît alors (figure 15 ci-dessous).

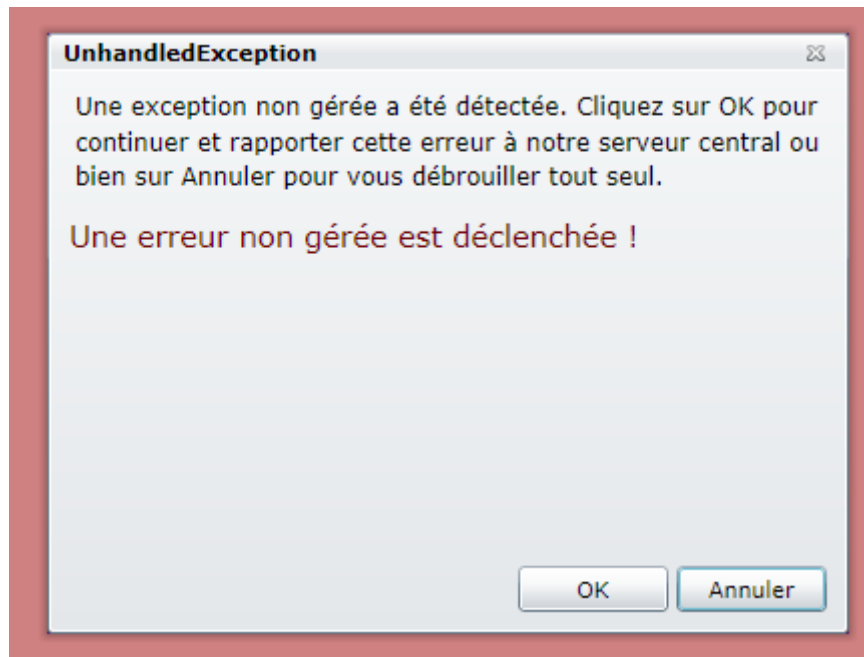


Figure 15 - Event Aggregator - Exception non gérée

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.
 - o 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'`EventAggregator`. Lorsqu'on crée 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 simplifiée : 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 override `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 régions

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 une 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 construire 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 connaît 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 connaît 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 personnalisé

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 tenue. 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 :

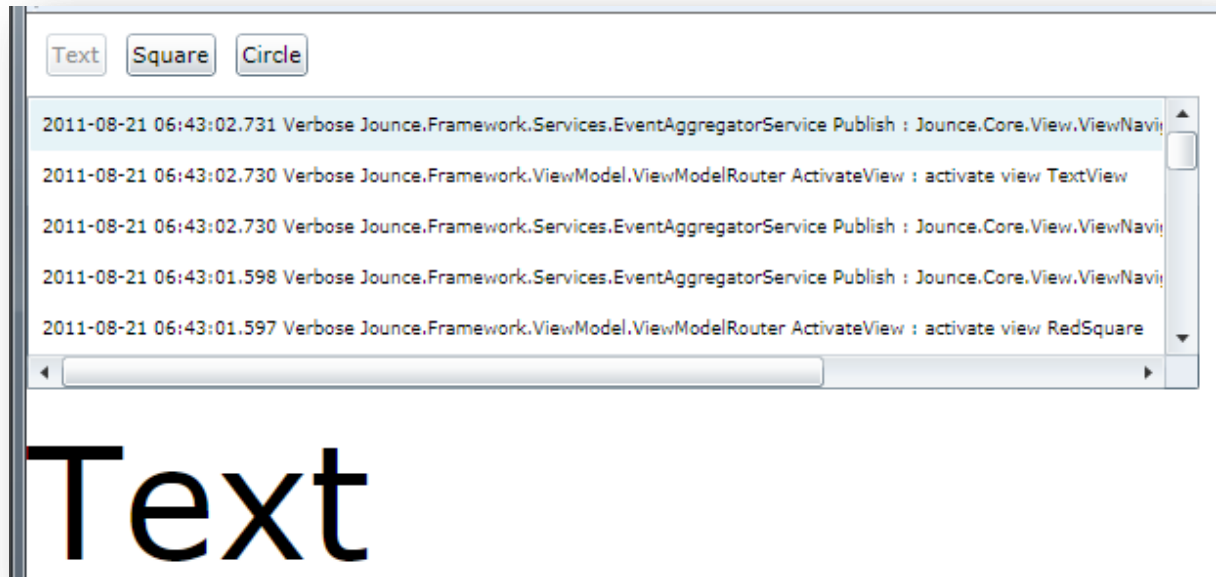


Figure 16 - Logger personnalisé

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 42, 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 raisonnables⁴.

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 événementielle 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, 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 fonctionnent 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...*

⁴ 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.

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'asynchronisme 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 bibliothèques 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 événements 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'une classe sachant exécuter les Workflows
- On crée 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 :

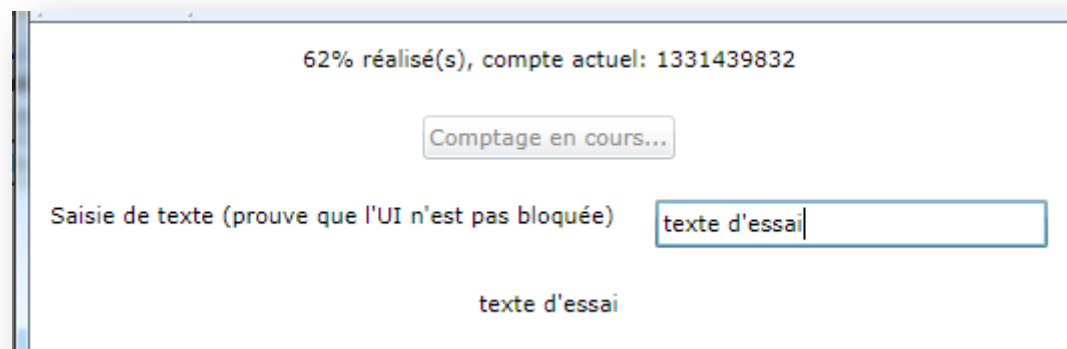


Figure 17 - Workflow

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'Élément 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 bouton.

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, **WorkflowRoutedEvent**, pour attendre qu'un événement 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 18 - Mr Smiley est content !



Figure 19 - 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 :

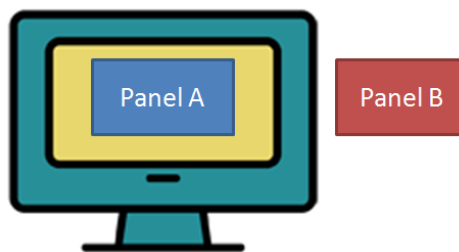


Figure 20 - VSA – Panel A (B désactivé)

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 :

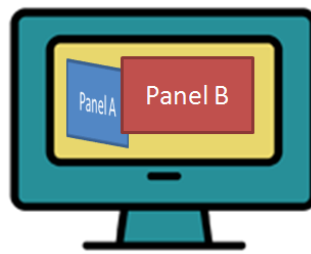


Figure 21 - VSA - Panel B activé

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 apparaître 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 posé 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 répondre à des événements 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 événements, 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 **LayoutRoot** 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 utilisant 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 ? ! ☺

Bon Développement !

Olivier Dahan

MVP Silverlight



Pour tout conseil, formation, audit ou développement , contact : odahan@e-naxos.com

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...