Formations .NET – Audit, Conseil, Développement

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

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

MEF et Silverlight 4+

[Applications modulaires]

Version 1.0 Août 2011

Sommaire

Table des figures

Code Source

Cet article 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.

Décompressez le fichier Zip dans un répertoire de votre choix. Attention les projets nécessitent Silverlight 4 et Visual Studio 2010 au minimum. Le dernier exemple utilise le toolkit MVVM Light de Laurent Bugnion.

Projets fournis

Exemple1 « `HelloWord` » , Importation de propriété et de champs

Exemple2 « `MultipleModules` », Importation de plusieurs modules

Exemple3 « `TypedMetaData` », Métadonnées fortement typées

Exemple4 « `CustomAttribute` », Attribut d'exportation personnalisé

Exemple5 « `DynamicXap` », Chargement dynamique et MVVM

Préambule

MEF, **Managed Extensibility Framework**.

> ***MEF est un Framework managé (sous CLR donc) permettant d'améliorer l'extensibilité des applications.***

MEF est une technologie qui fut longtemps un projet séparé, disponible pour WPF et Silverlight sur CodePlex. Arrivé à maturité, MEF a été introduit dans Silverlight 4 (et dans le Framework .NET 4.0). Il devient donc une brique essentielle de l'édifice.

Donc tout le monde connaît l'acronyme, tout le monde l'a déjà lu, entendu, prononcé. Mais combien de lecteurs à cet instant précis et sans se référer à une documentation ou des tutoriaux pourraient construire une application utilisant MEF ?

Certainement très peu car si le mot est connu, et le principe vaguement aussi, peu de développeurs connaissent suffisamment bien la façon d'utiliser MEF pour ne serait-ce qu'envisager de l'intégrer dans leurs applications, tout de suite, sans y réfléchir.

Et c'est dommage. D'autant que l'utilisation de MEF peut soulever d'autres questions comme celle du Lazy loading, ou l'utilisation du cache d'assemblage en conjonction avec MEF, ou encore comment marier MEF et un framework MVVM ou encore plus réaliste, comment offrir des modules MEF différents selon les profils des utilisateurs ? Questions qui réclament un peu d'expérience avec la bête pour pouvoir y répondre.

Il est vrai que la fourniture indépendante de MEF pendant longtemps, avec des variations notables qui ont fait que telle démo ne tournait plus ou que tel tutoriel était devenu caduque n'a pas arrangé les choses. On s'y intéresse un jour, on fait des exemples, et quand on en a besoin, plus rien ne marche avec la dernière version. C'est le côté agaçant de ces produits finis mais en cours d'élaboration, diffusés officiellement mais pas encore intégrés au Framework. Le prix à payer pour savoir à l'avance ce qui sortira un jour, mais un savoir remis en cause à chaque release. Le seul bénéfice est que ce long processus permet aux équipes de MS d'affiner le produit et de l'intégrer au Framework qu'une fois une certaine maturité atteinte. Nous sommes gagnants à l'arrivée, avec un Framework riche et solide. Mais l'entre-deux n'est pas sans poser de problèmes, je le concède volontiers (par exemple l'un de mes billets écrit en 2008 sur MEF n'est plus directement utilisable avec le MEF officiel intégré à SL4).

L'autre aspect négatif de cette fourniture diluée dans le temps est qu'en réalité la version finale intégrée au Framework est une « vieille nouveauté ». J'ai évoqué plus haut mon billet de 2008 sur MEF… 2008 ça fait longtemps ! Qui sera intéressé par un nouvel article sur MEF alors que mon vieux billet, comme la majorité des autres, est sur le Web de longue date et y est bien référencé avec le temps, alors même que plus aucun ne donne de conseils fiables pour la version d'aujourd'hui ? Ceux qui tenteraient maintenant d'utiliser MEF en en lisant ce qu'ils trouveraient facilement sur le Web concluraient rapidement que ce n'est pas « stable » ou que « ça ne marche pas ». Même la documentation sur CodePlex n'est pas vraiment à jour…

Il n'y a pas de choix parfait et Microsoft fait des efforts louables pour faire évoluer la plateforme sans la polluer avec des nouveautés instables et immatures. Pour cela il faut bien sortir des versions

intermédiaires, déconnectées du Framework en se laissant la possibilité de faire des « breaking changes » justement pour atteindre la maturité minimale qui honore tant le Framework. Mais il y a des dommages collatéraux comme ceux que je viens d'évoquer…

Bref, une application complète est complexe. Elle doit marier plusieurs technologies pour atteindre ses objectifs : MEF, WCF Ria Services, Linq to Entities, Linq to Object, Linq to Xml, MVVM, Entity Framework, Injection de dépendance, Framework de navigation… Chacune de ces techniques ou technologies sont parfois difficiles à maîtriser en elles-mêmes, alors que dire de la difficulté à les marier ensemble dans un tout cohérent sans tomber dans le code spaghetti !

Le présent article n'a pas vocation à répondre directement à cette dernière interrogation, il faudrait plusieurs livres pour en venir à bout certainement. Puis un livre sur ces livres, une prise en main. Puis un tutoriel sur le livre sur les livres… Au final on serait de nouveau noyé. Car tel l'Océan, on ne peut pas décider de maîtriser Silverlight en s'entraînant dans sa baignoire puis dans une piscine, puis un lac, etc… Un vrai marin de haute mer se forme en haute mer. Et seuls ceux qui y survivent peuvent dire qu'ils sont marins.

Des technologies comme Silverlight sont tellement vastes (en intégrant toutes les technologies satellites) qu'à la fois on est bien obligé de l'aborder par petits bouts, tout en ne pouvant considérer y comprendre vraiment quelque chose qu'une fois qu'on a réellement navigué sur cet Océan.

Rien de désespérant ou de paradoxal dans ce constat, marin de haute mer est un métier, une passion, et cela se mérite. La sécurité absolue, la connaissance parfaite n'existent pas, la prise de risque existe toujours. C'est ce qui en fait toute l'exaltation et toute la valeur légendaire de ceux qui rentrent au port sains et saufs…

MEF – Le besoin

Nous avons vécu sans MEF pendant des décennies, qu'est-ce qui pourrait bien nous forcer à l'utiliser aujourd'hui ? Quel ou plutôt quels besoins nous forcent-ils à nous y intéresser ?

Nos logiciels prennent du poids… Ils sont de plus en plus « riches » de fonctionnalités diverses et variées, chacune devenant de plus en plus sophistiquée. Au final l'ensemble devient difficilement maintenable et pèse trop lourd pour une utilisation via l'Internet. Même en mode desktop, avoir un logiciel qui se charge vite et répond tout de suite est un challenge dès qu'il est obligé de charger et d'initialiser des tonnes de code.

Deux problèmes se posent alors : Celui du poids de l'ensemble et de son temps de chargement, et celui de la modularité. La maintenabilité étant un besoin tellement basique qu'il est presque déplacé de le citer encore, sauf d'admettre que cet objectif n'est pas la plus grande réussite des informaticiens, en général…

Qu'un logiciel puisse se charger rapidement est essentiel pour l'UX (l'Expérience Utilisateur). La vivacité, le fait d'être « *responsive* », de répondre immédiatement aux sollicitations de l'utilisateur fait la différence entre un bon logiciel bien conçu et un logiciel lambda.

Mais plus loin, le foisonnement des fonctionnalités impose de les *modulariser*. Pour rendre la maintenance plus simple, certes, mais aussi pour offrir « à la volée » de nouveaux modules sans avoir à modifier, recompiler, redéployer toute l'application ; pour limiter les modules chargés simultanément ; pour offrir à l'utilisateur les modules dont il a besoin (selon son profil, la licence qu'il a achetée, etc…).

Dans l'idéal on part d'une situation de ce type :

Figure 1 - L'application idéale

La gentille Carlita ????, après avoir bien travaillé, release la version 1.0 de sa superbe application. Mais très vite la situation devient la suivante :

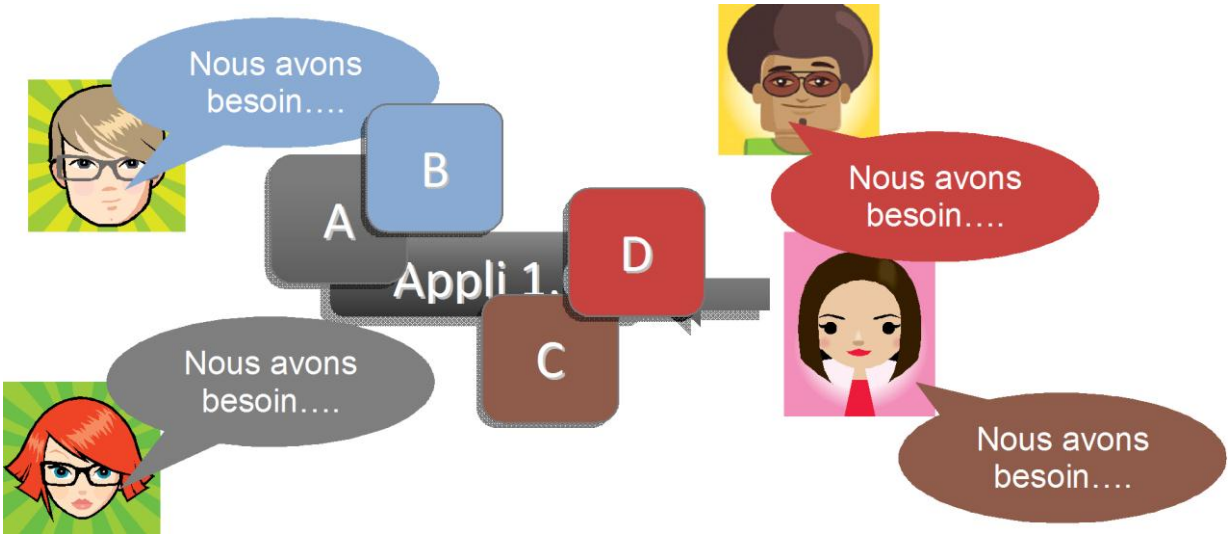


Figure 2 - Dans la vraie vie…

Autant les utilisateurs que les informaticiens s'aperçoivent à l'usage que telle ou telle fonctionnalité manque cruellement, que telle ou telle autre devrait être modifiée, etc… La pauvre Carlita se retrouve face à une situation explosive, les releases se succèdent pour supporter les « verrues » A, B,C, D, … Bientôt l'alphabet ne suffira plus !

MEF est une solution simple qui permet **de gérer la monté en puissance d'une application** et qui **accepte avec tolérance les extensions incrémentales**. C'est un bénéfice immédiat pour :

- Le client
- Le développeur et son équipe

Certes on savait comment charger des fichiers XAP dynamiquement avant MEF. Mais il ne s'agit que d'une des phases d'une gestion dynamique et modulaire : comment connaître la liste des modules disponibles (le *discovery* service), comment la filtrer selon le profil utilisateur, comment gérer les dépendances entre les modules (quel module doit aussi charger tel autre pour pouvoir fonctionner) ? … Bien des problèmes que le simple chargement dynamique d'un XAP ne saurait résoudre à lui seul.

> ***Il faut d'ailleurs noter que si, lorsqu'on évoque MEF, surtout avec Silverlight, on pense au chargement dynamique de XAP, cela n'est qu'une utilisation spéciale. MEF fonctionne de base avec un seul XAP en lui permettant de bénéficier de cette même modularité simplificatrice.***

Pour gérer toutes les facettes de cette modularité il faut en réalité une couche logicielle adaptée. Tout comme nous utilisons désormais des Frameworks MVVM (MVVM Light dont j'ai parlé il y a quelques temps dans de gros articles, ou Caliburn.Micro dont je parlerai bientôt, ou Prism), il convient d'utiliser un Framework pour gérer « l'extensibilité » des applications.

Ce Framework existe, c'est MEF.

Forcément les petites démonstrations de base sont toujours simples et idylliques, se basant sur le principe faussement évident, qu'une fois qu'on a compris les petites démonstrations simples on est capable de faire une vraie application… Aussi vrai que jouer avec un bateau en plastique dans votre baignoire vous prépare à barrer un trois-mâts au cap Horn !

Mais il faut bien prendre le problème par un bout, commencer doucement.

C'est pourquoi nous passerons ici aussi par l'étape de la baignoire avec le petit canard jaune qu'on s'amuse à faire voguer sur la mousse… Mais nous tenterons d'aller un peu plus loin, sans aller jusqu'au cap Horn ! Et nous aborderons les rivages de problèmes plus réalistes, donc plus complexes.

Larguez les amarres et hissez la Grand-Voile moussaillons, nous levons l'ancre !

Du lego, pas un puzzle !

MEF permet de construire des applications modulaires dont les services rendus peuvent apparaitre ou disparaitre au fur et à mesure de son existence sans remettre en cause la totalité de l'édifice.

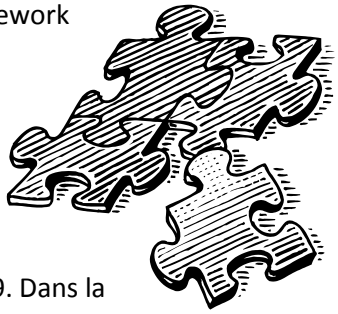
De ce point de vue l'application peut être vue comme une boîte de Lego : une série de briques élémentaires, s'enchaînant les unes ou autres, pouvant dépendre les unes des autres, et créant un tout cohérant mais non figé.

Tout le contraire d'un puzzle ! Et pourtant c'est dans cet état que se retrouvent nombre d'applications dès qu'elles ont évolué u peu… Des tas de pièces spécifiques, s'emboîtant ici et surtout

pas là, avec des trous laissés par les pièces manquantes, et dont la seule évolution possible, à termes, est de finir éparpillées sur le tapis après une bonne crise de nerf...

Figure 3 - Une application est faite de "parties"

L'ajout de fonctionnalités, le changement de certaines est, en réalité, partie prenante de la construction d'un logiciel. Mais si cela n'a pas été prévu d'emblée, on se retrouve face à un puzzle. Morcelé. Tiraillé par des pièces non adaptées rentrées en force comme un enfant face à un puzzle trop complexe. Les pièces ne sont pas interchangeables, elles ont été conçues pour un emplacement, une utilisation unique et spécifique. Il n'y a plus de maintenance possible sans casser l'ensemble, sans remettre en cause l'allure général du puzzle.



En adoptant MEF dès la création d'une application, tout comme un Framework MVVM, on s'évite de se retrouver un jour face à un immense puzzle mal agencé, un peu gondolé par les pièces entrées en force aux mauvais endroits…

MEF ne s'intègre pas « plus tard » ou « si… » mais tout de suite. Même s'il reste possible de « MEFiser » une application existante, cela n'est valable que dans les belles démonstrations pour les PDC ou sur Channel 9. Dans la vraie vie, s'y prendre trop tard… c'est trop tard !

Pensez Lego et non puzzle, utilisez MEF systématiquement…

MEF et MVVM

Ces deux-là n'ont pas été conçus pour fonctionner ensemble. Cela ne signifie pas qu'ils sont incompatibles mais seulement que rien n'a été prévu d'un côté comme de l'autre pour faciliter le mariage.

On peut même sentir une certaine confusion au départ entre ces deux approches pourtant différentes car leurs avantages sont présentés presque de la même manière : meilleure maintenabilité, couplage faible entre les modules, etc… Autant d'arguments qui sont ceux de MVVM comme de MEF.

Ya-t-il compétition entre les deux approches ?

Non. La maintenabilité de MVVM provient de sa séparation entre code et interface. Celle de MEF s'applique entre tous les modules d'une application. Le couplage faible de MVVM s'applique là aussi entre le code et l'UI, entre la Vue et le Modèle de Vue principalement. Le couplage faible de MEF est plus global et concerne tous les modules d'une application. MVVM met en place une

communication par message pour dialoguer entre modules. MEF créée un lien fort mais virtuel par le jeu des importations et des exportations qui peuvent traverser les XAP.

Si les deux techniques tentent d'aboutir à des résultats proches, elles ne se situent pas au même niveau d'implémentation et elles empreintes des chemins si différents qu'au bout du compte elles en deviennent complémentaires et que leurs bénéfices s'ajoutent au lieu de se superposer.

Reste toutefois à bien comprendre comment mixer ces deux approches dans une même application pour en tirer profit. Ce que nous verrons plus loin.

Principes de base : Exportation, Importation, Composition

Les principes de MEF sont très simples :

- *Exporter* consiste à marquer un code par un attribut spécifique. Dès lors ce code pourra être vu comme l'une des briques de l'ensemble.
- *Importer* consiste, par un marquage de même type, à indiquer quels variables sont en demande d'informations en provenance des briques exportées.
- *Composer* est une opération qui demande à MEF, une fois le catalogue des briques et des demandeurs connu, de faire la jonction entre eux, de satisfaire les demandeurs en leur fournissant les données des briques exportées.

Tout cela peut avoir lieu au sein d'un même XAP, entre diverses DLL le composant, ou bien, entre le XAP « maître » et des XAP satellites.

MEF fonctionne aussi avec WPF et Windows Phone 7.

Le gain le plus grand pour une application Silverlight est bien entendu la séparation en plusieurs XAP qui ne seront téléchargés qu'en fonction des besoins de l'utilisateur.

L'exportation

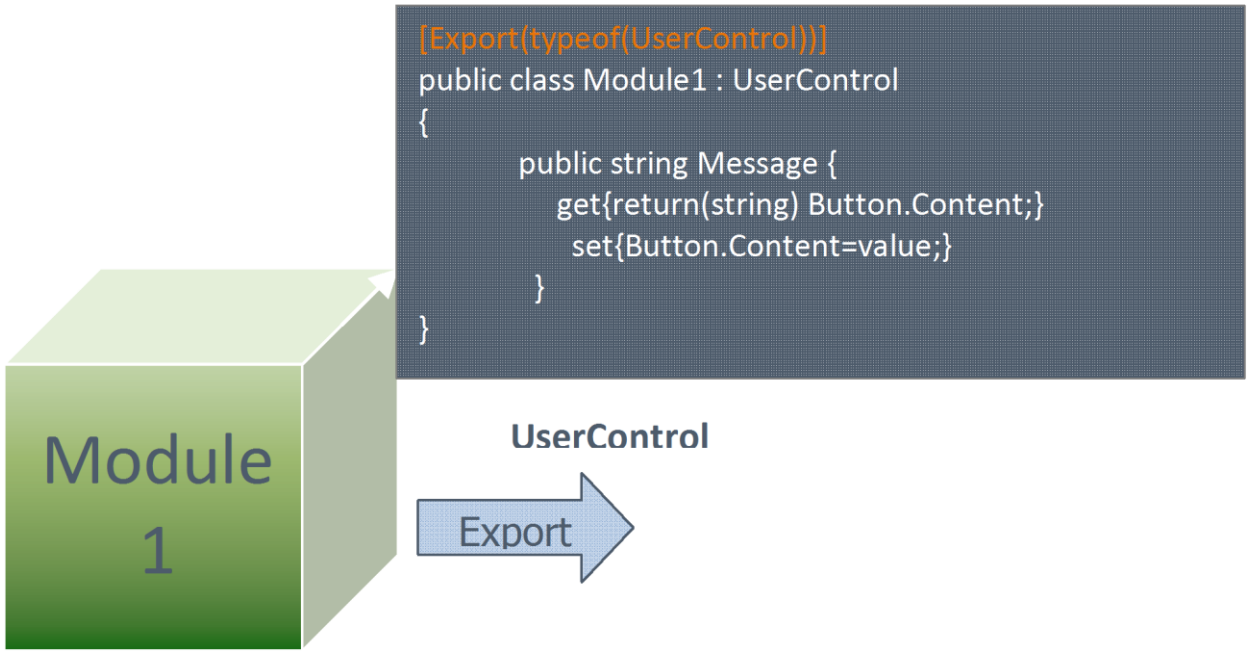


Figure 4 - Le principe d'exportation

Le module (ou partie) qu'on souhaite « publier » pour le rendre disponible à l'application est un code « normal », ici un `UserControl`. L'exportation s'effectue très facilement en ajoutant un attribut `[Export]` directement sur la classe. Il existe des variantes de cet attribut permettant de fixer des conditions plus subtiles que le seul nom de la classe (le type) ce qui est le comportement par défaut.

On remarque que le module de l'exemple ci-dessus possède une propriété publique « `Message` » tout à fait standard. En réalité, en dehors l'attribut d'exportation, le code du module ne change pas, on écrit les choses comme s'il n'y avait pas MEF. C'est l'un de ses grands avantages, MEF est très peu intrusif.

Mais cela n'interdit pas à notre module d'être lui-même consommateur d'autres parties…

L'importation

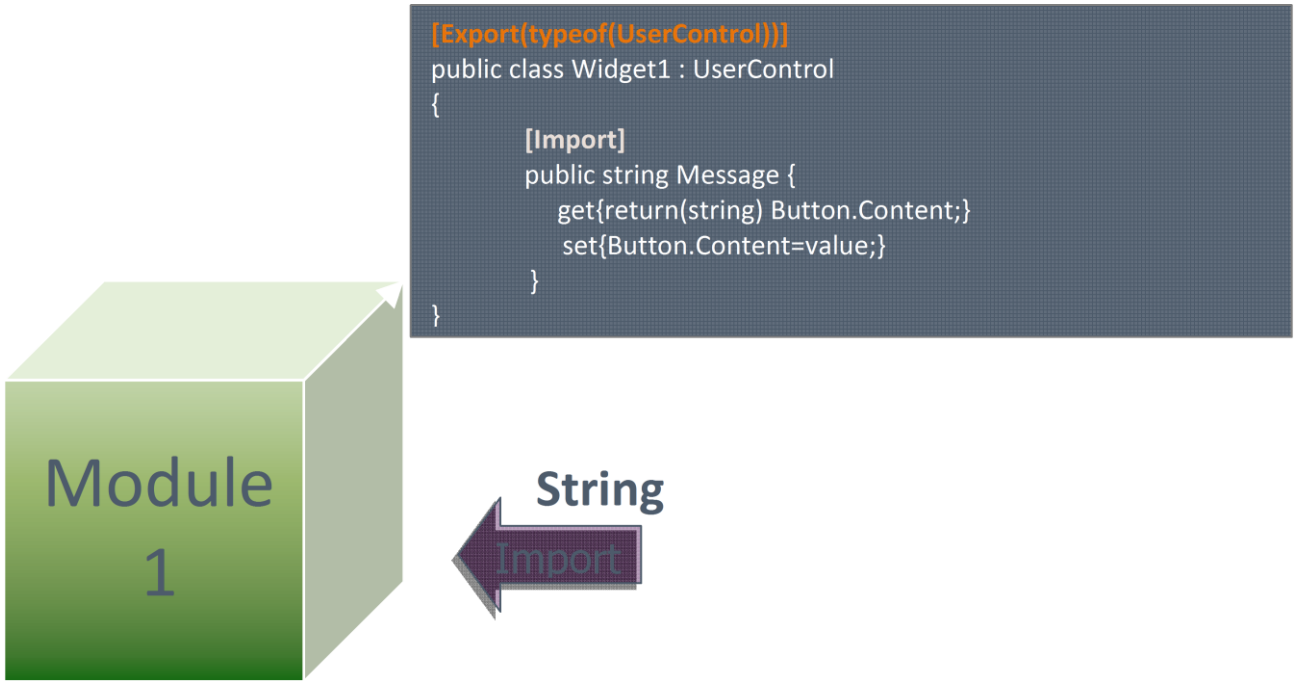


Figure 5 - le principe d'importation

Dans cet exemple, la propriété « `Message` », qui est parfaitement « normale » au départ, se trouve décorée d'un simple attribut `[Import]`. Cela signifie que la valeur initiale du message sera délivrée par une autre partie du logiciel.

Notre module est ainsi une partie exportée, disponible pour le grand jeu de Lego, mais il est lui-même consommateur d'autres parties. Il existe de fait une dépendance avec un autre module, mais ce dernier est inconnu. On sait seulement qu'il fournira une `String` pour « nourrir » la propriété. L'appareillement entre les « exportateurs » et les « importateurs » est effectué par MEF, sachant, comme on vient de le voir, qu'un module peut appartenir aux deux catégories à la fois.

Le découplage entre les parties est extrêmement fort (totale méconnaissance des autres parties) alors même que les interactions entre elles peuvent être nombreuses.

Sans MEF ces codes seraient interdépendants, auraient des références croisées les uns vers les autres et seraient impossibles à réutiliser ailleurs ou dans d'autres conditions sans casser ces dépendances. Avec MEF on conserve la souplesse (et la nécessité fonctionnelle) des dépendances mais celles-ci sont résolues « ailleurs », « autrement » sans créer de liens forts entre les parties.

Pour l'instant on constate que l'importation n'indique rien de plus que l'attribut Export. C'est-à-dire que les mécanismes par défaut de MEF seront utilisés pour gérer l'appareillement. Ce comportement par défaut repose sur le type des objets.

Dans le cas du module lui-même, l'exportation s'effectue via le type `UserControl`. Ce qui est très générique aussi mais qui peut à la rigueur être suffisant : chaque DLL ou XAP peut être construit pour ne proposer que des modules sous la forme de `UserControl`. Pas de risque qu'un `UserControl`

construit pour l'interface par exemple puisse se trouver parmi les modules : un tel composant ne sera bien évidemment pas marqué par l'attribut Export…

En revanche, pour la propriété de type `string`, les choses sont différentes. Des chaînes de caractères une application peut en exporter des dizaines. En tout cas si elle choisit d'utiliser ce mécanisme pour initialiser certaines chaînes il y a fort à parier qu'il y en ait plus d'une. L'attribut `Import` utilisé sur la propriété n'est alors vraiment pas suffisant pour permettre à MEF de savoir de quelle chaîne en particulier il est question.

Pour ce faire, l'attribut d'importation supporte qu'on lui passe le nom d'un contrat. Ce contrat est un nom, une chaîne de caractères et sera réutilisé par le module qui exporte la valeur. Le code de l'attribut devient alors :

```
[Import("Accueil.Message")]
        public string Message {…
```

Ainsi marquée, la propriété `Message` ne sera « nourrie » que par une exportation réutilisant le même nom de contrat.

La composition

C'est l'étape cruciale dans le processus MEF. Après avoir collecté les exportateurs et les importateurs, MEF doit les connecter, vérifier les dépendances, charger les modules annexes, etc…

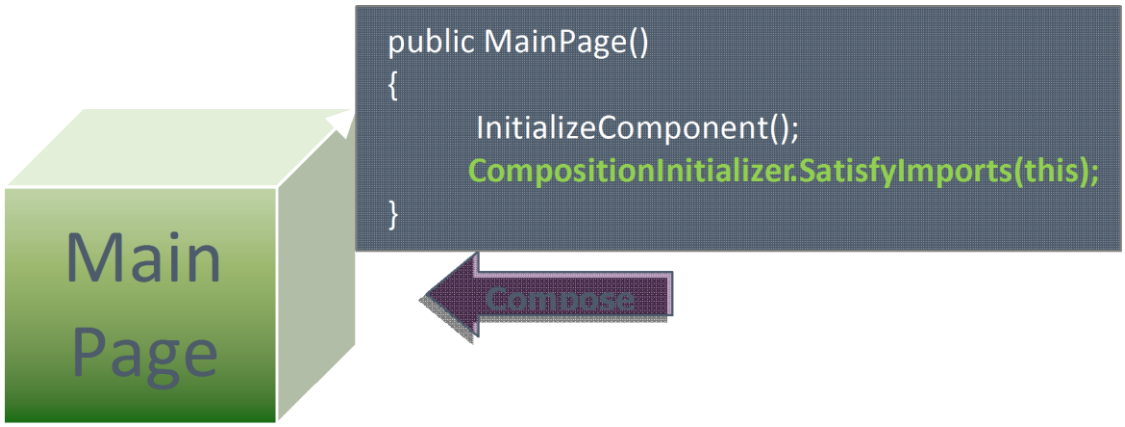


Figure 6 - le principe de composition

En général, mais nous verrons que cela n'est pas systématique, c'est à la page principale de l'application, le « *shell* » (la coquille qui gère tous les modules), que revient la charge de lancer la composition de MEF. L'appel, comme on le voit sur l'extrait de code ci-dessus, est explicite « `SatisfyImports(this)` », c'est-à-dire « satisfais les importations », le paramètre « `this` » prenant comme base l'instance courante : c'est un peu le début du fil d'Ariane que MEF remontera pour satisfaire toutes les importations et uniquement celles qui ont un intérêt pour l'instance passée en paramètre.

L'esprit de MEF

Il est essentiel de comprendre l'esprit de MEF et ce que son utilisation entraîne. **MEF n'est pas qu'une sorte de boite à outils permettant de gérer des plugins**. Cela serait terriblement réducteur.

*MEF est un framework d'**injection de dépendances** (qui est une forme d'**inversion de contrôle**).*

L'injection de dépendances consiste à éviter le couplage fort qui existe entre des classes qui s'utilisent les unes les autres.

Les modules affichables d'une application sont de ce type. Mais l'analogie avec la gestion de plugin fait perdre toute l'altitude nécessaire qu'il faut prendre pour voir que l'injection de dépendances va bien plus loin.

Elle impose un style, une sorte de pattern comme peut l'être MVVM.

L'application doit être conçue de façon totalement modulaire, pour les affichages, éventuellement, la découverte de plugins, pourquoi pas, mais **surtout au niveau de son architecture** interne par la mise en place notamment de classes de « services » qui exportent ces derniers sans savoir quelles autres classes vont les utiliser.

La modularité ne concerne donc pas seulement les pages ou les widgets affichés, elle est bien plus profonde et plus essentielle, elle touche à l'architecture même de l'application.

Tout ce qui est visuel est facile à démontrer selon le principe bien connu qu'un dessin vaut mille mots. De fait, toutes les démonstrations, et mes exemples n'échappent pas à la règle, utilisent plutôt des effets visibles (comme l'affichage de widgets) que de sombres calculs modularisés dans des classes de services…

Encore une fois, le travail de celui qui explique, par le choix des analogies qu'il fait, par le choix des moyens utilisés a tendance à « briser » la créativité du lecteur (ou de celui qui est formé). Engageant ce dernier sur des voies « évidentes », qui, tels les charriots romains ayant laissé des ornières dans la pierre des voies, force inconsciemment ou involontairement à suivre ensuite ces voies tracées lors de l'apprentissage. Montrer MEF par l'utilisation de widgets créée de telles ornières d'autant plus dangereuses qu'elles sont invisibles… C'est pour cela qu'ici je souhaite insister sur le véritable esprit de MEF, sa véritable finalité et toute la puissance qu'il libère. MEF n'est pas une gestion de plugin. MEF est un framework d'injection de dépendances qui force à penser la modularisation de toute l'architecture d'un logiciel.

J'espère, par cet avertissement, avoir corrigé, au moins un peu, l'effet pervers de ces voies invisibles que vont être les démonstrations à venir, vous évitant de les suivre aveuglément sans avoir même conscience de la beauté de tous les autres chemins qui s'offrent à vous…

Les métadonnées

Ce que nous venons de voir de MEF est l'essentiel. Le strict minimum pour faire fonctionner la librairie à l'intérieur d'un même XAP avec des importations très simples.

On conçoit aisément que dans la réalité il soit nécessaire d'échanger plus d'informations pour gérer les modules importés.

Par exemple, dans une application de type *dashboard* (tableau de bord), certains modules doivent peut-être, par vocation, être plutôt affichés en haut ou en bas de l'écran, à droite, au centre … Ce n'est bien entendu qu'un exemple d'information qu'un module peut avoir à communiquer sur lui-même pour en faciliter son utilisation. Dans un autre contexte d'autres informations pertinentes du même genre peuvent être nécessaires pour gérer convenablement chaque module.

MEF supporte la notion de métadonnées.

Exportation de métadonnées

Les métadonnées se précisent sous la forme d'un attribut qu'on ajoute à celui de l'exportation.

Pour suivre notre exemple de position d'affichage (qui, je le redis, n'est qu'un exemple et n'est pas forcément d'une pertinence absolue), le module pourra s'exporter de la façon suivante :

```
[ExportMetadata("Location",Location.Top)]
[Export(typeof(UserControl))]
public class Module1 : UserControl
```

Ici, "`Location`" est une énumération déclarée dans le code et autorisant, parmi d'autres, la valeur « `Top` ».

Importation de métadonnées

Exporter des métadonnées est une chose, encore faut-il pouvoir les récupérer !

C'est au moment de l'importation qu'on utilisera le type `Lazy<T>` qui autorise une syntaxe plus complète permettant d'indiquer le type des métadonnées :

```
[Export(typeof(UserControl))]
public class MainPage: UserControl
{
       [ImportMany(typeof(UserControl))]
       public IEnumerable<Lazy<UserControl, IWidgetMetadata>
       {    get;set;   }
}
```

Le code ci-dessus montre l'importation de multiples modules via l'attribut `ImportMany`, la propriété d'importation est ainsi en toute logique une liste, ou plutôt un `IEnumerable`. C'est dans cette déclaration qu'on utilise le type `Lazy<T>` auquel on passe le type du module attendu, ce type étant complété d'une interface définissant les métadonnées.

Les métadonnées personnalisées

Plutôt que la déclaration d'exportation que nous avons vue plus haut (un attribut `ExportMetaData` suivi d'un attribut `Export`) il est plus pratique dans de nombreux projets de créer son propre attribut d'exportation.

L'écriture est simplifiée et l'ensemble des informations importantes est fixé dans un attribut totalement dédié.

Par exemple, le code suivant :

```
[ExportMetadata("Location",Location.Top)]
[Export(typeof(UserControl))]
public class Module1 : UserControl
```

pourra se résumer à :

```
[ModuleExport(Location = Location.Top)]
public class Module1 : UserControl
```

Ici, `ModuleExport` est un attribut créé par héritage de l'attribut `Export`, il intègre directement les métadonnées spécifiques à cette exportation précise.

Contrôler les exportations via des attributs personnalisés n'est pas une obligation, mais, comme on le voit sur cet exemple, cela rend le code globalement plus clair, les intentions étant visibles. Alors qu'une fausse manipulation, une erreur de copier/coller peut faire « sauter » la ligne de métadonnées de la première formulation, l'attribut personnalisé ne peut être partiel. Il existe ou est oublié, mais cela se voit, et les métadonnées ont une valeur spécifique ou bien leurs valeurs par défaut. Au-delà de la stylistique c'est bien une robustesse accrue qu'offrent les attributs personnalisés.

Point intermédiaire

Nous venons de voir les grands principes de MEF. Le décor est planté mais cela reste malgré tout théorique et ne couvre pas toutes les facettes de MEF. Difficile de se lancer avec seulement ce que je viens de dire.

L'essentiel ici est d'avoir bien compris le principe général de MEF et comment il se met en œuvre dans une application.

La section suivante, par des exemples de code, va nous permettre de préciser les choses et de voir fonctionner MEF.

Comme il ne s'agit pas de reprendre in extenso la documentation de MEF, je vais tenter de balayer les principales utilisations qui réclament un peu de savoir-faire. Comme par exemple la gestion des catalogues dynamiques sous Silverlight, le mariage MEF et MVVM Light, etc…

Pour information la documentation complète de MEF se trouve ici :

http://mef.codeplex.com/wikipage?title=Guide&referringTitle=Home

Exemples pratiques

Planter le décor est nécessaire pour savoir de quoi on parle, mais vient un moment où les principes généraux et la documentation doivent céder la place à l'action, à du code qui montre comment faire.

Il reste beaucoup de facettes de MEF à présenter, et c'est au travers d'exemples concrets que nous allons progresser.

Le Hello World de MEF

Tradition oblige, voici un « hello world ! » avec MEF.

Le contexte est simple : un seul XAP, pas de MVVM. L'application n'est pas hébergée dans une application Web.

Juste un bouton qui affiche un message, ce message étant fourni par un module.

Les namespaces

Pour utiliser MEF dans une application Silverlight nous avons besoin d'ajouter deux références :

- `System.ComponentModel.Composition`
- `System.ComponentModel.Composition.Initialization`

Il faut aussi ajouter les `using` correspondants dans les unités de code concernées, bien entendu.

La seconde unité n'est utilisée que par le code qui initialisera MEF. La première est utilisée systématiquement.

Le visuel

L'application n'affiche qu'un bouton invitant à le cliquer, c'est sobre, c'est beau, c'est limpide, ça donne ça :

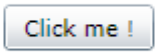


Figure 7 - Hello MEF !

La pureté du Design ne doit pas vous égarer, je sais, on resterait des heures à regarder une mise en page aussi parfaite… Mais il est temps de cliquer sur le bouton pour obtenir ça :

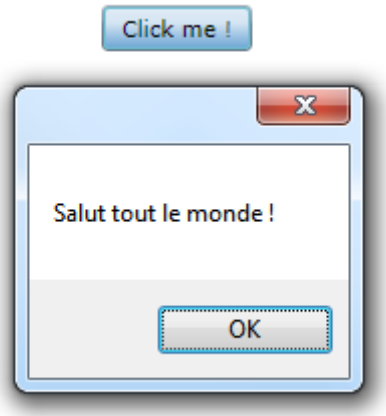


Figure 8 - Hello MEF (clic)

Je sais, au début ça fait un choc. Mais passé l'émotion bien naturelle, regardons le code si vous le voulez bien.

Le code

Premièrement j'ai créé un projet Silverlight (4.0 mais cela marche avec les versions suivantes de la même façon), sans le faire héberger dans un projet Web. Juste le minimum.

Dans la `MainPage` par défaut créée dans le projet par VS, j'ai juste ajouté, côté XAML, un simple bouton :

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
       <Button Content="Click me !"
          Height="23" HorizontalAlignment="Center" Name="btnHello"
          VerticalAlignment="Center" Width="75" Click="btnHello_Click" />
    </Grid>
</UserControl>
```

Il y en a plus long de déclarations préliminaires que de code utile puisque ce dernier se résume à une `Grid`, le `LayoutRoot` par défaut, et une balise `Button`.

Côté C# nous trouvons dans la `MainPage` le gestionnaire annoncé par le code XAML :

```
private void btnHello_Click(object sender, RoutedEventArgs e)
{ MessageBox.Show(Message); }
```

On comprend immédiatement au regard de la complexité de ce code pourquoi je suis MVP ☺

Ce code affiche le contenu de la propriété `Message` qui elle est définie de la sorte :

```
[Import("HelloWord.Message")]
public string Message { get; set; }
```

Enfin, MEF pointe le bout de son nez ! Je passe sur la déclaration de la propriété pour atteindre l'attribut : `Import`. Passé en paramètre : le nom du contrat. Ce nom est arbitraire mais devra correspondre à une exportation utilisant le même nom sur un type identique (`string`).

C'est tout pour la `MainPage`. Enfin presque. Mais progressons dans l'ordre. Nous venons de voir le code de la page principale de l'application, le « *shell* » en quelque sorte, celui qui consomme des modules. Ici le module est une instance unique d'une classe très simple (un `string`).

Regardons d'où vient ce fameux message.

```
namespace SilverlightApplication1
{
    public class MessagePart1
    {
        [Export("HelloWord.Message")]
        public string Hello = "Salut tout le monde !";
    }
}
```

La classe `MessagePart1` est une classe comme les autres, un peu vide certes, mais « normale ». Sans aucun signe ostentatoire pouvant choquer ou dérouter le développeur standard dans ses convictions intimes ou techniques.

Elle expose une chaîne de caractères (ce qui n'est pas très orthodoxe je le concède). Cette chaîne est décorée par l'attribut `Export` qui reprend le nom du contrat utilisé dans la `MainPage`.

On remarque ainsi que le nom de la chaîne elle-même qui est exportée n'a aucune importance puisque le contrat est nommé et que c'est ce nom qui conditionnera le travail de MEF.

C'est tout ?

Oui. Enfin il reste une chose à faire pour que cela fonctionne. Devinez-vous quoi ?

Dans la `MainPage` nous avons importé un « module » (rudimentaire, une chaîne, mais quelle que soit la classe le principe est le même), nous avons ensuite créé une classe exportant le « module » (la propriété `Hello` de `MessagePart1`).

Il manque la *glue*. L'action qui va coller les morceaux ensemble pour que cela fonctionne : la phase de **composition**.

Revenons au code de la `MainPage` et regardons son constructeur :

```
public MainPage()
 {
     InitializeComponent();
     CompositionInitializer.SatisfyImports(this);
 }
```

La seconde ligne effectue ce travail de composition.

C'est vraiment tout ?

Cette fois-ci oui.

N'est-ce pas déconcertant de simplicité ?

Alors allons un cran plus loin avec l'exemple suivant.

Gestion de plusieurs modules et métadonnées

Nous allons accélérer le pas et voir d'une part comment utiliser MEF pour gérer des « vrais » modules (dans le style d'une gestion de plugins) tout en étudiant au passage les métadonnées.

Pour l'instant les modules resteront rudimentaires, c'est le principe qui compte, le tout toujours limité à des classes faisant partie du même XAP.

Nous allons voir que les métadonnées peuvent s'utiliser soit directement, soit de façon fortement typées, soit par le biais d'un attribut personnalisé.

Commençons par le plus simple…

Le visuel

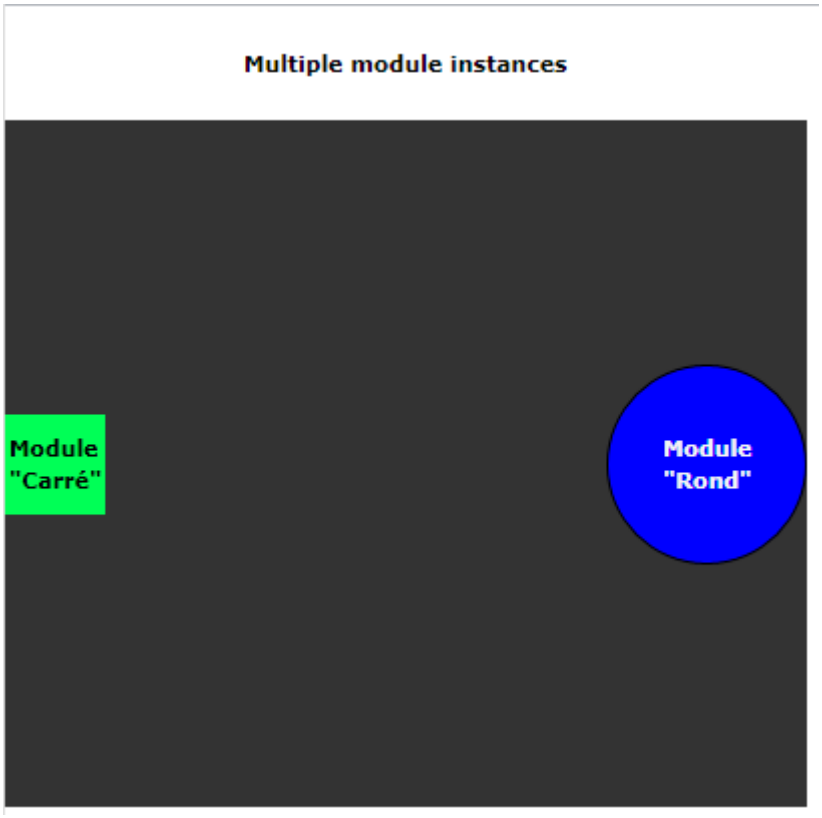


Figure 9 - Modules multiples

Je sais, le visuel de cet exemple est, bien plus encore que le précédent, un sommet artistique que seul un Designer confirmé pouvait créer. Un fond gris, un titre, un rond bleu et un carré vert, Miro et Picasso en seraient … verts de jalousie (ou bleus de peur !). Au moins vous ne serez pas perturbé par une débauche d'effets spéciaux !

Le principe de fonctionnement

Nous simulons ici un *dashboard*, c'est-à-dire un tableau de bord, un style d'application très répandu (gestion de portefeuille boursier, surveillance de processus industriels…). Nous disposons ainsi d'un *shell* (la `MainPage`) offrant une surface d'affichage (un `DockPanel` ici) pour les différents modules.

Chaque module propose la visualisation (et éventuellement la saisie) de données différentes. Notre application propose pour l'instant deux modules. Le premier est matérialisé par un carré vert de 50x50, identifié comme tel par un texte, et un rond bleu de 100x100, identifié identiquement. Ces modules d'exemple ne font absolument rien, mais il le font bien.

Comme la figure ci-dessus le laisse voir, le module Carré Vert est *docké* à gauche, le module Rond Bleu étant *docké* à droite.

S'agissant d'une application modulaire utilisant MEF, vous avez compris que le Carré Vert et le Rond Bleu sont deux modules exportés et reconnus automatiquement par le *shell*. Nous allons voir comment.

Vous remarquerez que les modules sont *dockés* selon un ordre précis. Ce sera l'affaire des métadonnées de fixer ce genre de chose. Nous le verrons plus loin (tout en rappelant que ce n'est

qu'un exemple et que fixer la position d'affichage dans un module n'est pas une excellente idée, sauf cas particulier).

Enfin, une chose que vous ne pouvez pas voir directement, chaque module est un `UserControl` différent.

Nous avons ici le schéma classique d'une application *dashboard*. Seuls les modules sont simplifiés, mais en reprenant le code de l'exemple et en complétant les `UserControl` nous obtiendrions une application réelle. Il n'y a donc de simplification que dans le contenu des modules, pas dans le fonctionnement de l'application ni dans l'utilisation de MEF.

Le lecteur notera que j'ai choisi d'utiliser un `DockPanel` pour servir de conteneur, cela est parfaitement arbitraire, l'application fonctionnerait de la même façon avec une `Grid`, un `Canvas` ou un `WrapPanel` par exemple. La seule différence concernerait les métadonnées qui dans l'exemple retourne une indication de *dockage* propre au `DockPanel`, il faudrait adapter cette information de positionnement pour un autre type de conteneur. Les modules pourraient aussi avoir des positions attribuées par le *shell*, ce qui semble plus plausible dans la réalité. La position d'affichage n'est qu'un exemple de métadonnées. Dans une application réelle les métadonnées peuvent retourner toutes sortes d'informations : nom du module, GUID, liste d'objets, etc… C'est vous qui décidez du contenu des métadonnées qui, de façon interne, sont représentées par un dictionnaire d'objets (donc une clé au format `string`, et une valeur de type `object`).

Le Code

Passons à l'essentiel : le code.

Les principes de base démontrés dans l'exemple précédent restent parfaitement valides :

- Exportation
- Importation
- Composition

La façon de réaliser chacune de ces étapes comporte juste quelques variations.

La MainPage

Son code XAML est très simple, comme je le disais plus haut : un `DockPanel` dans une `Grid` et un `TextBlock` pour afficher le titre :

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:toolkit=http://schemas.microsoft.com/winfx/2006/xaml/presentation/toolkit
    x:Class="SilverlightApplication1.MainPage"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400" Width="400" Height="400">

    <Grid x:Name="LayoutRoot" Background="White">
       <Grid.RowDefinitions>
              <RowDefinition Height="0.143*"/>
              <RowDefinition Height="0.857*"/>
       </Grid.RowDefinitions>
```

```
        <toolkit:DockPanel x:Name="MainDock" LastChildFill="False"
                                        Grid.Row="1" Background="#FF333333"/>
        <TextBlock TextWrapping="Wrap" d:LayoutOverrides="Width, Height"
                   HorizontalAlignment="Center" VerticalAlignment="Center"
                   FontWeight="Bold">
                   <Run Text="Multiple"/><Run Text=" module instances"/></TextBlock>

    </Grid>
</UserControl>
```

La surface d'affichage des modules sera le `DockPanel` qui se nomme `MainDock`.

Le code C# de MainPage

Comme dans l'exemple précédent, la `MainPage` est le *shell*, le réceptacle qui gère les modules. Il doit ainsi dans son constructeur appeler le moteur de composition de MEF puis il doit agir sur les modules pour les afficher (il pourrait les traiter d'une autre façon, ou plus tard). Pour cela il doit proposer une propriété publique qui recevra la liste des modules. La seule différence avec le premier exemple est que ce dernier gérait une importation d'instance unique (une chaîne de caractères) alors que le présent exemple va gérer une collection d'objets importés.

```
namespace SilverlightApplication1
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
            CompositionInitializer.SatisfyImports(this);
            displayModules();

        }

        [ImportMany]
        public Lazy<UserControl,IDictionary<string, object>>[] Modules { get; set; }

        private void displayModules()
        {
            foreach (var control in Modules)
            {
                var dock = (control.Metadata.ContainsKey("Location"))
                                    ? (Dock) control.Metadata["Location"]
                                    : Dock.Bottom;

                control.Value.SetValue(DockPanel.DockProperty,dock);
                MainDock.Children.Add(control.Value);
            }
        }
    }
}
```

Le constructeur ne présente aucune différence avec l'exemple précédent en dehors du fait qu'il appelle une méthode `displayModules` dont le rôle sera d'afficher chaque module.

La première différence notable se trouve dans la propriété importée. La propriété `Modules` est d'abord décorée avec l'attribut `ImportMany` au lieu de `Import`.

La signification semble évidente : `ImportMany` importe « *many* », plusieurs, modules, là où `Import` n'importe qu'une seule instance.

Il est donc naturel que la propriété décorée, `Modules`, soit un simple `IEnumerable<T>`. Cela serait en fait le cas si nous ne gérions pas de métadonnées. Mais comme nous souhaitons récupérer de telles métadonnées le type de `Modules` est `Lazy<T,M>`. Où le type `T` représente le type des objets importés (ici `UserControl`) et `M` le type des métadonnées (un dictionnaire générique `string` / `object` comme indiqué plus haut). Ici, `Modules` est déclarée comme étant une `Array` d'objets `Lazy<T,M>`.

Mais il s'agit là d'une simple tournure, il serait possible de déclarer Modules de cette façon :

```
public IEnumerable<Lazy<UserControl,IDictionary<string, object>>> Modules {get; set;}
```

On retrouve alors `IEnumerable<>`. Cela ne fait aucune différence en réalité, c'est juste un choix stylistique.

La méthode `displayModules()` est plus intéressante puisqu'elle va nous permettre de récupérer les fameux modules et leurs métadonnées qui permettront de les positionner dans la surface d'affichage (le `DockPanel`).

`displayModules()` ne contient qu'une boucle `foreach` sur la propriété `Modules`. Elle va ainsi énumérer l'un après l'autre tous les modules qui auront été découverts par MEF.

La boucle se divise en trois étapes :

- Récupération des métadonnées pour obtenir l'information de placement du module,
- Modification de l'emplacement du `UserControl` en conséquence,
- Ajout du module à la surface d'affichage.

Les éléments de `Modules` ne sont pas réellement des `UserControl`, ce sont des `Lazy<T,M>`, rappelez-vous. De fait cette classe encapsule chaque `UserControl` importé qui est disponible au travers de sa propriété `Value`. `Lazy<T,M>` offre une autre propriété au sens immédiat : `MetaData`.

Grâce à cette dernière nous pouvons récupérer l'information de position que chaque module a exporté (nous allons voir bientôt comment).

Une fois l'emplacement connu, et la propriété `Dock` du `UserControl` fixée, ce dernier est ajouté à l'arbre visuel du `DockPanel`.

Le code des modules

Nous n'étudierons que le code d'un seul module sachant que le second est fait exactement pareil (en dehors de sa forme) et que vous disposez du code source avec l'article.

Au hasard… ce sera le Rond Bleu.

Tout d'abord il s'agit d'un `UserControl`. Dans un autre type d'application cela pourrait être n'importe quoi, pas forcément quelque chose qui s'affiche. On peut très bien modulariser du pur code (des méthodes de calcul, des effets visuels ou sonores, des sons, des collections d'images, du code envoyant des emails par différentes méthodes, …). Une application *navigationnelle* pourra, par

exemple, exporter des instances de `Page`, offrant ainsi à l'utilisateur un site composé selon ses besoins ou sa licence.

Bref, ici c'est un `UserControl`. Pour le Rond Bleu sa taille est fixée à 100x100 pixels avec un `TextBlock` centré indiquant le nom du module. Tout cela est très simple et se reproduit en quelques secondes. Ce n'est pas du côté de XAML qu'il faut chercher les astuces de cet article.

Le code C# du contrôle est le suivant :

```
namespace SilverlightApplication1
{
    [Export(typeof(UserControl))]
    [ExportMetadata("Location", Dock.Right)]
    public partial class RoundModule : UserControl
       {
              public RoundModule()
              {
                     // Required to initialize variables
                     InitializeComponent();
              }
       }
}
```

Le `UserControl` est construit « normalement ». Une fois encore, MEF n'impose rien au niveau des objets exportés. Si c'est un descendant de `Page` ou de `UserControl`, on le programme sans se soucier de MEF, idem s'il s'agit d'une classe de service.

MEF ne se fait voir qu'en décorant la classe `RoundModule` (le `UserControl`).

Tout d'abord il y a l'attribut d'exportation. Ici nous devons préciser le type. Un objet peut apparaître sous sa propre classe (en général sa classe mère si on gère plusieurs instances comme dans cet exemple) ou bien sous la forme d'une *Interface*. Cette dernière option est largement préférable dans la réalité. Travailler par contrat est un découplage de plus, et une sécurité intéressante (pas d'accès aux champs non pris en compte par l'Interface) et une garantie de maintenabilité plus grande.

Ainsi, l'attribut `Export` utilise ici le type `UserControl` mais pourrait très bien préciser le type d'une interface particulière dans le cas de plugins réels.

La vraie différence avec l'exemple précédent se situe dans le second attribut `ExportMetaData`. Il ne s'agit pas d'une obligation, comme je le disais au début de cet exemple, j'ai choisi d'ajouter la gestion des métadonnées pour éviter de multiplier de trop le nombre des projets. Celui que nous étudions en ce moment montre deux choses distinctes : la gestion des instances multiples et celle des métadonnées.

Sans les métadonnées nous ne pourrions pas récupérer l'information de placement utilisée par le `DockPanel`, ce serait au *shell* de décider où placer les modules. De même nous n'utiliserions pas `Lazy<T,M>` pour déclarer l'importation mais simplement un `IEnumerable<UserControl>`. Enfin, nous n'aurions pas, naturellement, d'attribut `ExportMetaData` dans les modules…

Mais nous en avons un ici. En fait nous pourrions en avoir autant que nous voulons, un pour chaque métadonnée exportée. Dans cet exemple nous n'exportons qu'une seule information, le *docking*. Ainsi nous ne trouvons qu'un seul attribut `ExportMetaData`.

Il prend en argument deux valeurs. La première est la clé. C'est-à-dire le nom de la métadonnée (ici `Location`, *position* en anglais). La seconde est la valeur (ici `Dock.Right`, dans le second module la valeur est `Dock.Left`, c'est la seule chose qui change en dehors de la couleur et de la forme).

Rappelez-vous que les métadonnées sont gérées par un `Dictionary<string,object>`. On comprend mieux la raison d'être des deux paramètres, ce sont les deux paramètres de la méthode `Add()` d'un dictionnaire générique de ce type.

Point Intermédiaire

Cet exemple est plus proche de la réalité tout en étant une abstraction qui ne doit pas vous faire perdre de vue que presque tous les choix effectués sont purement arbitraires. Le danger d'un exemple est d'orienter inconsciemment le lecteur, de bloquer sa créativité et son imagination en lui soumettant des voies toutes tracées.

Ici, il faut se rappeler que la gestion des métadonnées n'est pas une obligation et n'a rien à voir avec celle des instances multiples de modules. Il faut avoir conscience que les modules sont des `UserControl` uniquement parce que cela permet d'avoir un résultat visuel facilitant la compréhension mais qu'il pourrait s'agir de code pur n'affichant rien. Il faut se méfier aussi de l'utilisation directe qui est faite des modules dans le constructeur du *shell*, on pourrait très bien utiliser les modules à un autre moment sans les afficher tout de suite. Enfin, concernant la gestion des métadonnées il faut se souvenir qu'il s'agit d'un simple dictionnaire et qu'on peut y mettre ce qu'on veut. Le choix d'indiquer une option de positionnement à l'écran permet de voir (au sens visuel) le résultat ce qui est pratique pour une démonstration, dans une application réelle le positionnement des modules serait plutôt géré par le *shell* et les métadonnées reflèteraient d'autres informations plus pertinentes pour le fonctionnement de l'ensemble (nom du module pour l'afficher dans un menu par exemple, icône de la fonction représentée…).

Les métadonnées sont d'une grande importance dans beaucoup de situations réelles. Nous avons vu comment les gérer de façon simple. Cela fonctionne efficacement mais ces données ne sont pas typées. Dans une gestion de modules réelle il est préférable de typer ces informations.

Il existe deux façons de typer les métadonnées :

- En déclarant une interface
- En utilisant un attribut d'exportation personnalisé.

Voyons maintenant chacune de ces options.

Métadonnées fortement typées

Dans l'exemple précédent nous avons mis en évidence l'utilisation des métadonnées. Notre exemple n'utilisait qu'une seule métadonnée (le *docking*) l'exploitant au travers du dictionnaire fourni par défaut.

Cette approche est directe et ne réclame pas de code supplémentaire. Toutefois dans un logiciel bien conçu laisser des zones de « non typage » présente toujours un risque.

Il serait bien plus judicieux d'utiliser des métadonnées fortement typées plutôt qu'un simple dictionnaire d'objets.

MEF supporte cette possibilité avec très peu de code. Il suffit de créer une interface regroupant les métadonnées sous la forme propriétés en lecture seule et d'utiliser cette interface dans la déclaration de l'importation. Lors de l'exploitation du champ `MetaData` de l'objet `Lazy<T,M>` (un module retourné par MEF) cette propriété apparaitra alors comme étant du type de l'interface souhaitée. L'accès aux valeurs est alors plus direct (par le nom des propriétés de l'interface) et totalement « *safe* » du point de vue du typage.

Pour illustrer cette façon de faire, je vous propose de reprendre l'exemple précédent en le modifiant de la façon suivante :

- Ajout d'une seconde métadonnée (la couleur de fond de l'objet)
- Création d'une interface contenant les deux propriétés (*docking* et couleur)
- Modification de l'importation (pour spécifier l'interface)
- Modification de la méthode `displayModules()` (utilisation de l'interface pour fixer *docking* et couleur de chaque module affiché).

Ces modifications sont légères mais laissent apparaitre deux difficultés qui n'ont rien à voir avec MEF. J'ai hésité avant de les traiter ici, c'est hors sujet, mais d'un autre côté le code source seul ne permettrait pas forcément de comprendre pourquoi j'ai fait tel ou choix d'implémentation.

Le lecteur m'excusera donc la petite digression qui va suivre et qui ne concerne pas MEF mais qui, j'en suis convaincu, l'intéressera tout autant.

Digression

Vous êtes prévenu, si seul MEF, rien que MEF vous intéresse vous pouvez sauter ce passage…

Binder le Background d'un UserControl à l'un de ses éléments

En réalité le problème est plus vaste et concerne toutes les propriétés qu'offre un `UserControl`. Vous allez comprendre rapidement.

Prenons le « module Rond Bleu ». Puisque maintenant la couleur de l'objet peut être fixée par une métadonnée (ce qui sera détaillé plus loin) nous devons pouvoir changer la couleur `Background` du `UserControl` et que ce changement soit réellement appliqué au cercle.

Ce cercle est une `Ellipse` qui possède une propriété `Fill`.

La classe `UserControl` possède une propriété `Background`.

Ce que nous voulons c'est tout bêtement relier les deux. De telle sorte qu'une modification de la propriété Background du `UserControl` change le `Fill` de `l'Ellipse`.

C'est simple. Légitime. Cela se résout par un simple binding direz-vous.

Hélas Silverlight ne supporte pas toutes les subtilités du binding relatif de WPF. De fait il n'est pas possible d'utiliser la syntaxe complète (notamment avec `FindAncestor` sur le type `UserControl`) que nous pourrions mettre en œuvre directement sous WPF.

Certains petits futés pensent avoir trouvé la solution en exploitant *l'Element Binding*, en reliant la propriété `Fill` de `l'Ellipse` à la propriété `Background` du `UserControl`. Cela fonctionne. Mais,

trois fois hélas l'astuce ne fonctionne que si la cible est nommée. Ce qui oblige à ajouter un `x:Name` au `UserControl`.

Le premier problème qui se pose est que ce nom est codé en dur dans le XAML du contrôle. Que se passe-t-il si le nom est changé par code ? Le binding sera cassé. Mais cela n'est pas le plus gênant. Cette solution interdit tout simplement d'utiliser deux fois le même `UserControl`, la création de la seconde instance retournera une erreur (nom dupliqué).

Heureusement il reste une troisième possibilité : effectuer le binding dans le constructeur du `UserControl`, à cet endroit nous avons accès à « `this` » qui représente l'instance du `UserControl` et nous n'avons donc pas besoin d'un nom pour créer *l'Element Binding* entre la propriété de `l'Ellipse` et celle de son hôte (le `UserControl`)…

Cela explique pourquoi les constructeurs de deux modules ont été modifiés pour intégrer chacun un binding par code.

Pour le Carré vert (qui va changer de couleur donc), la couleur du fond est celui de sa `Grid LayoutRoot`, le binding est donc celui-ci :

```
LayoutRoot.SetBinding(Panel.BackgroundProperty, new Binding { Source = this, Path = new
PropertyPath("Background") });
```

La propriété `Background` de la `Grid`, qui provient de son ancêtre `Panel`, est bindée à la propriété `Background` de la source « `this` », donc de l'instance du `UserControl`.

Pour le module Rond Bleu (qui lui aussi changera de couleur) le binding est le suivant :

```
Circle.SetBinding(Shape.FillProperty, new Binding {Source = this, Path = new
PropertyPath("Background")});
```

Ici, c'est la propriété `Fill` de l'Ellipse (qui s'appelle `Circle`), propriété héritée de `Shape`, qui est bindée à la propriété `Background` de la source « `this` », donc l'instance du `UserControl`.

Cette approche est à demi satisfaisante mais elle permet d'éviter l'incohérence de la quatrième solution que certains retiennent : tout simplement ajouter une nouvelle propriété de dépendance au `UserControl` et ignorer superbement la propriété `Background` originale… C'est sûr, ça marche. Mais d'une part le code d'une propriété de dépendance est plus lourd que le binding montré ci-dessus, et d'autre part laisser une propriété `Background` qui ne fonctionne pas pour en créer un doublon (*`BackgroundBis`*, *`BackgroundVrai`* ? quel nom lui donner ?) m'apparait une hérésie fonctionnelle et stylistique.

Donc, j'opte pour le binding dans le constructeur. Une variante plus fiable pour des objets plus long à s'initialiser que nos petits modules consiste à placer le binding non pas dans le constructeur mais dans le `Loaded` de l'élément concerné (la `Grid` ou `l'Ellipse` de nos modules).

Bien entendu le même problème se poserait si nous voulions utiliser une autre propriété que `Background` du `UserControl`. La solution serait aussi la même.

Colors n'est pas une énumération

Satanées couleurs ! On pensait s'en être sorti avec le binding de la couleur de fond et voilà que nous tombons sur un autre problème !

Silverlight expose une classe `Colors` qui retourne des couleurs par leur nom. Cela est bien pratique.

Comme nous voulons passer la couleur de fond des modules dans leurs métadonnées, il semblerait évident d'écrire par exemple un code de ce genre :

```
[ExportMetaData("Background",Colors.Cyan)]
```

Tristement, nous sommes bien obligés de constater que le compilateur n'en veut pas… Un argument d'un attribut ne peut être qu'une constante, un `typeof` ou quelques autres possibilités, mais en aucun cas le *nom d'une propriété*.

En effet, `Colors` n'est pas une énumération… IntelliSense en nous donnant la liste des couleurs lorsqu'on tape « `Colors.` » nous fait oublier qu'ici il nous donne *la liste des propriétés statiques de la classe* `Colors`. Du coup `Colors.Cyan` est le nom d'une propriété statique et en aucun cas un item d'une énumération…

Une telle écriture est donc interdite, il va falloir trouver autre chose pour passer la couleur dans les métadonnées.

Pour faire simple nous passerons une chaîne de caractères indiquant le nom de la couleur. La version longue consisterait à déclarer une énumération de toutes les couleurs puis de convertir chaque élément dans sa bonne valeur ARGB. J'ai opté pour quelque chose de plus court ici.

C'est pourquoi vous trouverez le code suivant dans la `MainPage` :

```
private Color getThisColor(string colorString)
{
   var colorType = (typeof(Colors));
   if (colorType.GetProperty(colorString) != null)
    { var o =
              colorType.InvokeMember(colorString,
                                     BindingFlags.GetProperty,
                                     null, null, null);
      if (o != null) return (Color)o;
    }
   return Colors.Black;
 }
```

Cette séquence prend le nom d'une couleur (*case sensitive*) et via la réflexion recherche la propriété qui le porte dans `Colors` et se sert du résultat pour retourner la couleur de type `Color`.

L'avantage de ce code est qu'il tient en quelques lignes. La solution de l'énumération serait à mettre en œuvre dans une vraie application pour définir notamment toutes les couleurs qu'on retrouve dans WPF (alors que `Colors` sous Silverlight ne contient que très peu de couleurs).

Fin des digressions

Vous savez maintenant pourquoi le code source fourni avec cet article utilise un binding dans les constructeurs des modules et pourquoi le paramètre couleur est passé dans les métadonnées d'exportation sous la forme d'une chaîne de caractères…

Revenons à nos moutons !

Métadonnées via une Interface

Comme expliqué juste avant les petites digressions, nous allons procéder de la sorte :

- Ajout d'une seconde métadonnée (la couleur de fond de l'objet)
- Création d'une interface contenant les deux propriétés (*docking* et couleur)
- Modification de l'importation (pour spécifier l'interface)
- Modification de la méthode `displayModules()` (utilisation de l'interface pour fixer *docking* et couleur de chaque module affiché).

Ajout de la seconde métadonnée

L'ajout d'une seconde métadonnée est purement déclaratif dans les exports, pour chaque module nous aurons ainsi les déclarations suivantes :

Pour le Rond :

```
    [Export(typeof(UserControl))]
    [ExportMetadata("Location", Dock.Right)]
    [ExportMetadata("Background", "Purple")]
    public partial class RoundModule : UserControl
```

Pour le Carré :

```
    [Export(typeof(UserControl))]
    [ExportMetadata("Location",Dock.Left)]
    [ExportMetadata("Background", "Yellow")]
    public partial class SquareModule : UserControl
```

> *Je ne sais pas si cela provient de `Resharper` (excellent add-on dont je ne pourrais me passer pour coder) ou si VS sait le faire aussi par défaut, mais il se peut que la seconde déclaration de `ExportMedata` vous indique une alerte du type « attribut dupliqué ». Il suffit d'ignorer cet avertissement. Ici, nous multiplions les `ExportMetadata` par nécessité : un attribut par métadonnée.*

Création de l'interface des métadonnées

Cette interface contient toutes les métadonnées utilisées. Elle est constituée d'une suite de propriétés portant le même nom que les métadonnées et ne disposant que d'un accesseur en lecture. Voici l'interface `IModule` créée pour notre exemple :

```
namespace SilverlightApplication1.Contract
{
    public interface IModule
    {
        Dock Location { get; }
        string Background { get; }
```

```
    }
}
```

Modifier l'importation

La séquence d'importation, au lieu d'indiquer un dictionnaire générique doit maintenant préciser le nom de l'interface (cela se passe dans le code de `MainPage`) :

```
[ImportMany]
public IEnumerable<Lazy<UserControl, IModule>> Modules { get; set; }
```

Modification de la séquence d'affichage

C'est ici qu'on peut voir tout l'intérêt de l'interface et de son typage (toujours dans `MainPage`) :

```
        private void displayModules()
        {
            foreach (var control in Modules)
            {
                var dock = control.Metadata.Location;
                var colorName = control.Metadata.Background;
                var color = getThisColor(colorName);
                control.Value.SetValue(DockPanel.DockProperty, dock);
                control.Value.Background = new SolidColorBrush(color);
                MainDock.Children.Add(control.Value);
            }
        }
```

La propriété `MetaData` de l'objet module est automatiquement typée en tant que `IModule`, notre interface. Récupérer les données est donc direct et fortement typé (la chaîne est bien une `string`, le *docking* est bien représenté par une valeur de l'énumération `Dock`, tout cela sans transtypage depuis le type `object`).

Filtrage des métadonnées

L'utilisation d'une interface créée une sorte de « vue » sur les métadonnées. Et cette vue peut être considérée comme un *filtre* : seuls les modules exportés remplissant toutes les métadonnées de l'interface seront considérés comme valides et seront importés.

Il est essentiel de se rappeler de cet effet de bord. Soit pour s'en servir (mais c'est une voie douteuse que je vous incite à éviter), soit pour éviter que certains modules ne s'exportent pas (créant un bogue qui peut être difficile à trouver si on ne pense pas à ce filtrage implicite).

Pour éviter cette situation il existe un attribut à utiliser dans la déclaration de l'interface : `DefaultValueAttribute`. Sur la base de l'interface de notre exemple voici comment celle-ci pourrait être décorée :

```
namespace SilverlightApplication1.Contract
{
    public interface IModule
    {
        Dock Location { get; }
        [DefaultValue("Red")]
        string Background { get; }
    }
}
```

Par l'ajout de cet attribut, si un module ne spécifie pas de valeur pour la métadonnée `Background`, celle-ci prendra la valeur « `Red` » et ne sera pas considérée comme manquante, de fait le module ne sera pas filtré (donc ne sera pas exclu) lors de son importation. En revanche, l'oubli de la métadonnée `Location` exclura le module (je n'ai pas placé de valeur par défaut).

Point intermédiaire

Nous avons vu comment utiliser des métadonnées. Nous venons de voir comment rendre cette utilisation plus orthodoxe en typant ces informations via une interface et comment définir des valeurs par défaut sur cette dernière.

Mais il reste quelque chose d'assez peu pratique : la multiplication des attributs `ExportMetaData`. Cela soulève quelques problèmes :

- Le code est long surtout si on utilise de nombreuses métadonnées;
- Il est facile d'oublier une ligne de métadonnée, rien ne l'indiquera ni ne nous avertira du problème ;
- L'oubli d'une métadonnée filtrera le module qui ne sera plus reconnu et donc ne sera plus exporté ; ou, si une valeur par défaut a été spécifiée, c'est cette valeur peut-être pas appropriée qui sera utilisée ;
- Plus c'est long à écrire, plus on utilise copier/coller, et plus on introduit de bogues (après le « coller » il est facile d'oublier de modifier l'une des lignes) ;
- Enfin, ce n'est pas vraiment élégant.

Ces critiques peuvent paraître accessoires comparées aux nombreux avantages que les métadonnées confèrent à MEF.

Bien que secondaires, ces considérations nous poussent à vouloir quelque chose de plus clair, plus élégant et offrant moins de prise aux bogues et oublis éventuels.

Il existe une solution. Elle consiste à créer son propre attribut d'exportation.

C'est ce qu'on nous allons étudier à la section suivante.

Attribut d'exportation personnalisé

Rappelons la progression jusqu'à maintenant :

1) Un import simple d'une chaîne (le message du premier exemple de code)
2) Importation de multiples instances avec ajout de métadonnées
3) Amélioration des métadonnées avec un typage fort via une interface.

L'étape de simplification ultime consiste donc à faire en sorte de fusionner en un seul attribut tous les attributs d'exportation utilisés : `Export` et `ExportMetaData`.

Notre code sera propre, clair, les intentions seront visibles, les accès aux métadonnées typées, et nous disposerons alors d'une base saine mimant en tout point ce qui doit être fait dans une application réelle.

La création d'un attribut d'exportation est vraiment simple une fois qu'on a franchi les étapes précédentes. Cet attribut réemploie l'interface déjà définie mais de façon implicite, via les noms des propriétés qu'il expose.

Voici le code pour mieux comprendre :

```
namespace SilverlightApplication1.Contract
{
    [MetadataAttribute]
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public class ModuleAttribute : ExportAttribute
    {
        public ModuleAttribute() : base(typeof(UserControl)) { }

        public Dock Location { get; set; }
        public string Background { get; set; }

    }
}
```

Le code est vraiment court. La classe créée hérite de `ExportAttribute` et elle décorée par deux attributs : `MetadataAttribute` qui indique que cet attribut d'exportation gère aussi les métadonnées (optionnel), et `AttributeUsage` qui explique à MEF quels types d'éléments de code peuvent être décorés par le nouvel attribut. Dans ce code j'ai indiqué `AttributeTargets.Class` car les modules sur lesquels j'utiliserai l'attribut sont des classes. On pourrait autoriser l'utilisation du nouvel attribut sur des méthodes, des propriétés, etc… Le paramètre `AllowMultiple` est initialisé à `false` car tel qu'il est conçu notre nouvel attribut ne doit s'utiliser qu'une fois par classe.

Le corps de la classe se résume à publier un constructeur qui ne fait qu'appeler celui de base en passant le *type des objets décorés* puis à définir les propriétés correspondant aux métadonnées.

Le constructeur appelle celui de sa classe parent (`ExportAttribute`) en lui passant le type `UserControl`, car nos modules sont exportés sous cette forme. Rappelez-vous qu'il est possible d'exporter les modules sous la forme d'une interface commune (cas plus réaliste pour des plugins), ce serait alors le type de cette interface qui serait indiqué ici. Une erreur classique (et qui fait planter l'application) consiste à passer le type de l'interface des métadonnées au lieu du type du module exporté. Méfiez-vous !

Les propriétés portent des noms rigoureusement identiques à ceux utilisés dans l'interface de métadonnées. Le type doit lui aussi être respecté.

Notre application est désormais dotée d'une classe d'exportation personnalisée, il ne reste plus qu'à s'en servir…

Pour le Carré cela donnera :

```
    [Module(Location = Dock.Left, Background = "Yellow")]
    public partial class SquareModule : UserControl
```

Pour le Rond :

```
    [Module(Location = Dock.Right, Background = "Purple")]
```

```
    public partial class RoundModule : UserControl
```

Rien d'autre ne change. Nous avons juste rendu le code plus lisible. L'utilisation d'un attribut personnalisé permet aussi d'afficher clairement *l'intention*. Ici l'attribut s'appelle `ModuleAtrribute` (ce que C# permet de tronquer en `Module` tout court, sans le suffixe). A la seule lecture de ce code nous comprenons que l'objet marqué est un « Module » (ce qui a un sens dans notre application).

N'oubliez-pas en effet, que MEF permet d'exporter et d'importer de nombreuses choses au sein d'une même application ! Il peut exister des modules, des services (code d'exportation de données en CSV, en XML…), des messages, etc…

Si tout ce petit monde est marqué du seul attribut standard « `Export` » il sera difficile de comprendre ultérieurement pourquoi quelque chose ne marche pas (un message exporté comme un module par exemple). L'utilisation d'attributs d'exportation spécialisés évite ces confusions et facilite la maintenance du code tout en le rendant plus clair.

Point intermédiaire

Arrivé à ce point de l'article vous commencez certainement à avoir une bonne idée de ce qu'est MEF. Même si sur le fond cela est certainement vrai (ou il faut que j'arrête d'écrire des articles !) l'impression est peut-être trompeuse car il reste beaucoup de choses à dire sur MEF et son utilisation « en vrai ».

Hélas tout ne pourra pas être couvert dans cet article qui dépasse pourtant les 70 pages. N'oubliez pas la documentation de MEF !

Organiser les déclarations

Une chose qui n'a pas été dite à propos de tous ces éléments c'est comment on doit les organiser dans une application.

Et c'est un point important si on veut conserver une unité à l'ensemble et ne pas se prendre les pieds dans le tapis.

En effet, une application réelle proposera vraisemblablement bien plus de modules que les exemples des pages précédentes, bien plus d'exportations, bien plus d'importations en divers endroits.

Pour que l'application soit maintenable, il est ainsi conseillé de créer et de déployer un assemblage à part qui ne contient que les contrats (généralement des interfaces, voire des classes abstraites). Je parle ici des contrats sous lesquels les modules (ou autres) apparaissent, pas des métadonnées. Tous les modules, toutes les parties, tous les XAP éventuels formant l'application feront référence à cet assemblage.

On garantit ainsi l'unicité et l'uniformité indispensable de ces déclarations.

Cela peut sembler plus lourd, mais dans une application réelle utilisant intensivement MEF, donc de nature hautement modulaire, vous verrez vite que ce conseil est payant.

Il est aussi conseillé de définir dans un autre (ou plusieurs autres) assemblage(s) toutes les interfaces de métadonnées ainsi que les attributs d'exportation personnalisés.

Une telle organisation s'avère de plus bien adaptée au travail en équipe.

Dans la suite de l'article je vais tenter de répondre à d'autres questions pratiques sur l'utilisation de MEF. Mais faisons un point sur certains aspects abordés jusqu'ici.

Les différentes exportations

Avec MEF tout tourne autour des exportations : c'est le point de départ. Ce qui peut être exporté, et comment cela est exporté conditionne ce qui pourra être importé et comment cela pourra être exploité.

Les exemples que nous avons vus jusqu'ici portent sur l'exportation de classes visuelles (des `UserControl`), le premier exemple portait sur une propriété (un message). Pour bien comprendre les possibilités de MEF il semble important de bien saisir quelles sont les possibilités de l'exportation.

Exportation de parties composables

Une exportation de partie composable (*Composable Part level export*) est une action réalisée lorsque une partie composable s'exporte elle-même. C'est le cas des `UserControl` des exemples précédents. Dans ce cas la classe est simplement décorée d'un attribut `Export`.

Exemple :

```
[Export]
public class UnePartieComposable {  ... }
```

Exportation de propriétés

Une classe peut exporter une ou plusieurs propriétés sans être elle-même une partie composable et exploitable comme telle. C'est le cas du premier exemple où un champ de type `string` est exporté (le message). Il s'agit, pour être tout à fait exact, d'un cas non pas identique mais similaire puisque nous parlons ici d'exportation de propriétés et non de champs. Mais, nous l'avons vu, cela est possible et fonctionne de fait de la même façon.

Exemple d'exportation et d'importation d'une propriété :

```
public class Configuration
  {
    [Export("Timeout")]
    public int Timeout
    {
      get {return int.Parse(ConfigurationManager.AppSettings["Timeout"]);}
    }
  }
  [Export]
  public class UsesTimeout
  {
    [Import("Timeout")]
    public int Timeout { get; set; }
  }
```

La valeur entière `Timeout` de la classe `Configuration` est exportée sous le nom de contrat "`TimeOut`" alors que sa classe mère (`Configuration`) n'est pas composable. La valeur est ensuite importée par la classe `UsesTimeout` qui utilise l'entier en déclarant une propriété décorée par

l'attribut `Import` utilisant le même nom de contrat. La classe mère (`UsesTimeout`) participe, en outre, au jeu de composition puisqu'elle est marquée par `Export`, ce qui n'est qu'une possibilité et non une obligation.

Exportation de méthodes

C'est un cas que nous n'avons pas vu dans cet article car je préfère le principe d'exportation d'une classe de services, quitte à ce qu'elle ne contienne qu'une seule méthode, que l'exportation d'une ou plusieurs méthodes d'une classe qui ne serait pas composable.

Ce jeu qui consisterait à « piocher » ici et là des petits morceaux ne me plait guère et présente le risque important de tomber rapidement dans du code spaghetti…

Mais si je peux m'autoriser quelques conseils, je ne suis pas un censeur, et comme l'exportation des propriétés ou des champs (que je déconseille pour les mêmes raisons), je me dois de vous parler de l'exportation des méthodes. Il serait bien prétentieux de penser que si je ne vois aucune bonne justification à la chose personne sur terre n'en trouvera jamais une ! Et si ce cas se présente à vous, il faut savoir que cela est possible.

Une partie peut donc en effet exporter des méthodes sans être elle-même une « partie composable ».

Une chose à savoir : en raison d'une limitation du Framework (selon la documentation de MEF d'où sont tirés les morceaux de code de cette section) l'exportation des méthodes est limitée à celles ne possédant pas plus de 4 arguments.

Exemple :

```
public class MessageSender
  {
    [Export(typeof(Action<string>))]
    public void Send(string message)
    {   Console.WriteLine(message);    }
  }
  [Export]
  public class Processor
  {
    [Import(typeof(Action<string>))]
    public Action<string> MessageSender { get; set; }

    public void Send()
    {  MessageSender("Processed");   }
  }
```

Dans cet exemple, la classe `MessageSender` exporte la méthode `Send` qui prend en paramètre un message. L'exportation est typée comme une `Action<string>`.

La classe `Processor` fait une importation de la méthode sous la forme d'une propriété qui est utilisée ensuite par sa propre méthode `Send`.

On notera que l'exportation qui est faite ici par l'utilisation d'un type peut se faire aussi par un simple nom de contrat (comme dans notre premier exemple avec la chaîne de caractère `Message`). Les signatures doivent bien entendu rester parfaitement compatibles.

Les exportations héritables

MEF supporte un mode d'exportation très particulier, les exportations *héritables*. Il s'agit en fait de pouvoir décorer une classe ou une interface dont l'exportation est automatiquement héritée par les classes qui les implémentent.

Cela est très intéressant quand on souhaite rendre MEF totalement *transparent*.

Imaginons une interface `ILogger` proposant une méthode `Log(string message)`. Les développeurs vont pouvoir créer plusieurs classes implémentant cette interface. L'une fera des logs dans un fichier texte, l'autre en XML, l'autre via le réseau avec TCP, etc. Toutes ces classes implémenteront `ILogger`. On reste dans de la programmation habituelle, classique, *sans aucune référence à MEF*.

Imaginons maintenant, presque à leur insu pourrait-on dire, qu'on souhaite que toutes les classes implémentant `ILogger` puissent participer à une composition. Le programme principal devra être à même de découvrir toutes les versions de `ILogger` afin que l'utilisateur puisse choisir celui qui lui convient par exemple.

Chaque classe implémentant `ILogger` ne connaît rien de MEF, les développeurs qui les ont écrites non plus.

Mais en décorant l'interface `ILogger` de l'attribut `InheritedExport`, comme par magie, toutes les classes implémentant `ILogger` seront visibles par MEF et pourront être composées… Pratique non ?

Voici un exemple :

```
[InheritedExport]
public interface ILogger {  void Log(string message); }
public class Logger : ILogger {  public void Log(string message); }
```

Ce code reprend ce que je viens de dire : l'interface `ILogger` définit un contrat constitué d'une seule méthode, `Log`, qui permet de logger un message.

L'interface elle-même est « MEF-aware », celui qui l'a écrite connaît MEF.

Mais regardons la classe `Logger` définie juste après : elle ne fait qu'implémenter `ILogger`, rien de plus. Pourtant cette classe pourra participer « sans le savoir » à une composition MEF basée sur l'interface `ILogger`, comme si `Export(typeof(ILogger))` la décorait…

Un procédé original qui peut même permettre, après coup, et pour peu qu'on travaille déjà avec des interfaces, de « MEFiser » et de modulariser une application existante.

Public ou Private ?

MEF supporte la découverte de parties publiques et non publiques. Cela est automatique et ne nécessite aucune action particulière. Mais en revanche il faut savoir que ce mécanisme n'est pas supporté dans les environnements à *confiance partielle*, comme peut l'être Silverlight… Sous Silverlight on prendra comme règle que ne peuvent être composées que des parties (au sens large : classes, propriétés, méthodes…) qui sont publiques.

Cela semble finalement naturel et logique.

Mais il faut se méfier des évidences… Par exemple une application pourrait fort bien vouloir utiliser MEF pour composer des parties totalement `private`, sans vouloir violer aucun principe de la programmation objet. Sous Silverlight cela n'est pas possible. Composable = publique. A se rappeler donc…

Lazy<T> et le chargement différé

Dans les exemples que nous avons vu j'ai volontairement introduit la gestion des métadonnées pour grouper les thèmes afin de conserver une taille raisonnable à cet article (ce qui n'est déjà plus le cas, mais j'aurai essayé !).

Dans ces exemples, il a été montré que pour gérer les métadonnées, et surtout pouvoir les récupérer, il fallait définir les importations avec le type `Lazy<T,M>`, ou `T` est le type de l'objet importé et `M` le type des métadonnées.

Cela est vrai (heureusement). Pour récupérer les métadonnées, typées ou non, il faut utiliser `Lazy<T,M>` car seule cette déclaration permet de spécifier que les métadonnées doivent être gérées et est capable de les retourner en encapsulant les parties dans un objet incluant une propriété `Metadata`. On a vu d'ailleurs que dans ce cas l'objet exporté lui-même est accessible via la propriété `Value` et non pas directement.

Si tout cela est bien vrai je vous ai caché une partie de la vérité. Il est temps de l'avouer : cette technique pour récupérer les métadonnées n'est qu'une astuce. Officielle certes, mais une astuce tout de même car il n'existe pas d'autres moyens d'obtenir ces données spéciales dans MEF.

C'est une astuce car comme son nom le laisse supposer depuis le départ, `Lazy<T>` ou `Lazy<T,M>`, les deux existent, ont surtout été conçues pour permettre le *chargement tardif* des objets d'une composition. On peut ainsi utiliser `Lazy<T>` si on souhaite faire du chargement tardif sans pour autant utiliser de métadonnées.

Lazy veut dire paresseux en anglais. Le *Lazy loading* consiste à charger des objets « paresseusement » c'est-à-dire tout le contraire de « rapidement ».

Dans le jeu des compositions de MEF il faut prendre en compte le temps de découverte des parties puis de leur initialisation sachant que MEF gère les dépendances et qu'une partie composable peut réclamer, par un effet de bande, l'initialisation de nombreuses autres parties liées par des dépendances.

Une application « moderne » est avant tout une application qui se charge et qui réagit vite. Sur un PC, mais aussi, il faut en tenir compte aujourd'hui, sur des machines beaucoup moins puissantes que sont les Smartphones et les Tablettes… Et avec Silverlight il faut aussi *prendre en compte la lenteur d'Internet*.

Une application utilisant MEF ne le fera pas pour un ou deux modules. Une application modulaire se basant sur MEF le fera certainement pour des « tas » de modules et parties, parfois réparties sur plusieurs XAP qui devront être téléchargés via Internet (ou un Intranet dans le meilleur des cas).

Or, que se passe-t-il si une application tente de découvrir et de charger toutes les parties composables ? Cela va être long, voire très long. Tout le contraire de ce qu'on cherche à obtenir.

Pour cette raison le type `Lazy<T>` ou `Lazy<T,M>` quand on spécifie les métadonnées, est avant tout conçu pour *différer le chargement des parties composables*. La découverte est effectuée au plus vite, mais le chargement est mis en attente.

Jusqu'à quand ?

Jusqu'à ce que la propriété `Value` soit lue…

Pour faire du *Lazy Loading* avec MEF, il suffit donc de remplacer le type de la partie composable par un `Lazy<T>` dans la déclaration de l'importation.

Ainsi l'hypothétique déclaration :

```
[Import]
public IMessageSender Sender { get; set; }
```

passera en mode *Lazy Loading* en écrivant :

```
[Import]
 public Lazy<IMessageSender> Sender { get; set; }
```

Ceci est valable pour les importations simples (comme ci-dessus) autant que pour les importations multiples, avec, ou comme ici, sans métadonnées.

Chargement différé immédiat

Derrière ce titre en forme de paradoxe se cache une stratégie assez souvent utilisée : elle consiste à proposer un Shell le plus léger possible et, dès que celui-ci est affiché, à démarrer une tâche de fond qui télécharge tout (ou partie) des modules à utiliser. Le temps que l'utilisateur saisisse son login par exemple, qu'une première page soit affichée, l'application peut avoir le temps de télécharger tous les modules complémentaires…

Ici on utilise bien un chargement différé ou tardif, mais en lançant ce chargement dès le début de l'exécution du programme.

L'utilisateur dispose ainsi d'une application qui se charge très vite et le temps de chargement des modules est masqué astucieusement.

Au fur et à mesure que les modules arrivent et sont découverts, des entrées de menu peuvent s'ajouter, des boutons de navigation apparaître, etc.

C'est une stratégie intéressante qui semble malgré tout éloignée du but du *Lazy Loading*. En réalité ici on vise principalement le chargement rapide du logiciel, c'est tout. Les modules sont tous chargés en mémoire systématiquement. Certaines applications profitent mieux de la modularisation de cette façon. Comme toujours, les décisions doivent être prises projet par projet, en fonction des impératifs propres à chacun.

Je ne traiterai pas cet exemple ici, faute de place, mais voici un code exemple issu de la documentation CodePlex de MEF qui montre de façon assez simple la stratégie :

```
public class App : Application {
  public App() {
    this.Startup += this.Application_Startup;
    this.Exit += this.Application_Exit;
    this.UnhandledException += this.Application_UnhandledException;

    InitializeComponent();
  }

  private void Application_Startup(object sender, StartupEventArgs e)
  {
    var catalog = new AggregateCatalog();
    catalog.Catalogs.Add(CreateCatalog("Modules/Admin"));
    catalog.Catalogs.Add(CreateCatalog("Modules/Reporting"));
    catalog.Catalogs.Add(CreateCatalog("Modules/Forecasting"));

    CompositionHost.Initialize(new DeploymentCatalog(), catalog);
    CompositionInitializer.SatisfyImports(this)

    RootVisual = new MainPage();
  }

  private DeploymentCatalog CreateCatalog(string uri) {
    var catalog = new DeploymentCatalog(uri);
    catalog.DownloadCompleted += (s,e) => DownloadCompleted();
    catalog.DownloadAsync();
    return catalog;
  }

  private void DownloadCompleted(object sender, AsyncCompletedEventArgs e) {
    if (e.Error != null) {
      MessageBox.Show(e.Error.Message);
    }
  }

  private Lazy<IModule, IModuleMetadata>[] _modules;

  [ImportMany(AllowRecomposition=true))]
  public Lazy<IModule, IModuleMetadata>[] Modules {
    get{return _modules;}
    set{
      _modules = value;
      ShowModules()
  }

  private void ShowModules() {
    //logic to show the modules
  }

}
```

MEF, MVVM et Chargement Dynamique de XAP

Jusqu'à maintenant nous avons vogué par vent arrière sur une mer calme. Ici on se retrouve au Vendée Globe (donc en solitaire, sans escale et sans assistance), lors du passage du cap Horn par gros temps…

En effet, nous allons voir comment créer une application modulaire utilisant MEF capable de charger dynamiquement des modules au *runtime* suite à une action utilisateur (depuis un module découvert

par MEF), les modules d'extension se trouvant dans un XAP secondaire situé sur le serveur, le tout en respectant la pattern MVVM et en limitant la taille de chaque exécutable. Pire, dirais-je, nous allons ajouter une gestion de catalogue de modules permettant de charger d'autres parties en fonction du profil de l'utilisateur !

Ceux qui survivront auront le droit à mes chaleureuses félicitations ☺

Même si j'ai bien chargé la barque, le projet que nous allons étudier reste malgré tout un simple exemple. La réalité est toujours plus complexe et soulève toujours mille nouvelles questions. La sagesse impose de garder cette évidence à l'esprit.

Mon but n'est d'ailleurs pas d'écrire un livre sur MEF ni de me substituer à la documentation du Framework. Je souhaite seulement vous donner le coup de pouce nécessaire et suffisant pour vous permettre de prendre le large tout seul. Certains préfèrent aux courses dangereuses en solitaire la sécurité des croisières en compagnie d'un capitaine expérimenté qui saura les conduire à bon port, j'en profite ainsi pour rappeler à ces derniers que je vends mes services ☺

Le projet

Le projet reprend l'esprit des derniers exemples de code de cet article : il y a un *shell*, la vue principale, qui permet d'afficher des widgets (des gadgets, des parties, des modules, tout cela revenant au même).

Le dernier exemple présenté permettait, via les métadonnées MEF, d'indiquer la position de l'affichage de chaque module. Nous reprendrons cette idée en jouant ici sur deux conteneurs (une partie haute et une partie basse) et ajouterons aux métadonnées d'autres informations comme le nom du widget afin qu'il puisse être affiché dans une `Listbox` qui présentera ainsi tous les modules chargés.

L'autre aspect que notre projet devra couvrir est le chargement dynamique des modules. De fait les « widgets » seront chargés et découverts de plusieurs façons différentes :

- Un premier widget sera découvert dans l'application principale et sera immédiatement affiché dans le shell.
- Ce widget comporte un bouton permettant de charger à la demande d'autres widgets se trouvant dans un XAP secondaire sur le serveur.
- Un profil utilisateur simplifié sera affiché dans la vue principale. Le changement de profil entrainant le chargement de nouveaux modules adaptés à ce profil, via un catalogue XML se trouvant sur le serveur.

Cela représente trois types de chargement et de découverte des modules. Certains sont internes à l'application principale, d'autres proviennent de XAP externes. Le chargement des modules externes étant lié soit à une action utilisateur, soit à une décision de l'application.

Si nous nous arrêtions là cela ferait déjà beaucoup… Mais il manquerait une autre approche couramment utilisée : La découverte de widgets par le biais d'un catalogue sur le serveur.

En effet, la gestion de catalogue de MEF est parfaite pour WPF. De base MEF sait retrouver tous les modules d'un sous-répertoire donné par exemple. Mais rappelez-vous que Silverlight ne peut pas

accéder aux répertoires du serveur. Il est donc impossible d'utilise ces mécanismes fournis avec MEF. Il va être nécessaire d'implémenter notre propre gestion de catalogue.

Une petite séquence de captures d'écran va éclaircir tout cela avant de passer à l'étude du code.

Le projet en action

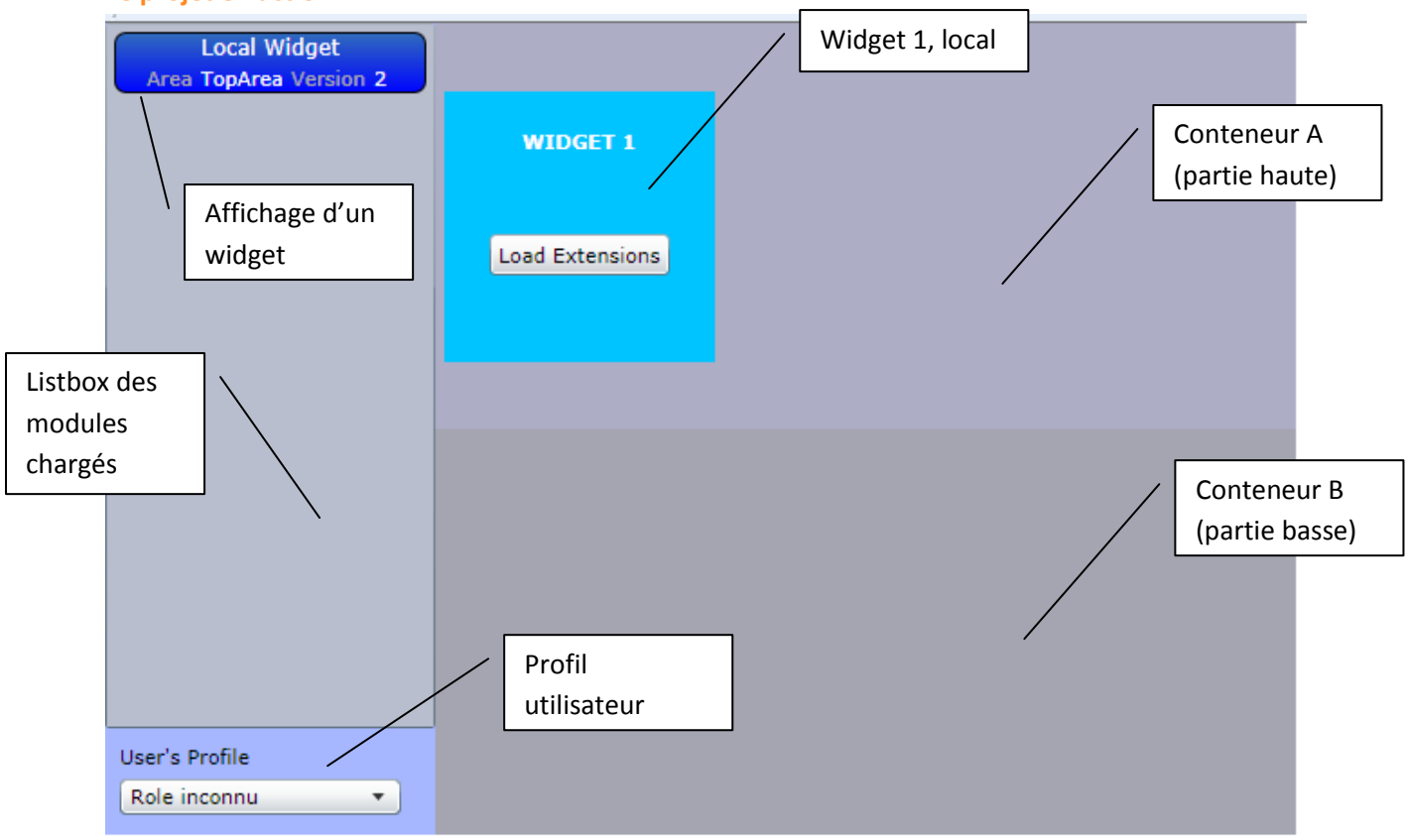


Figure 10 - DynamicXAP - Le Shell au lancement

Comme le montre la capture (figure 10), le shell se présente sous la forme de deux parties verticales, à gauche la `Listbox` affichant les modules chargés et en bas le profil utilisateur, à droite l'affichage des widgets coupé en deux horizontalement, avec un conteneur A en partie supérieure et un conteneur B en partie inférieure.

Ce sont les métadonnées des widgets qui indiquent la position d'affichage (en haut ou en bas) ainsi que le nom qui apparaît dans la `Listbox`.

Chaque module chargé est ainsi affiché dans la `Listbox` et présente les informations suivantes :

- Le nom du widget (tel que précisé dans ses métadonnées)
- Le nom de la zone où il est affiché (`TopArea` ou `BottomArea`)
- La version du widget (information factice uniquement pour ajouter quelques métadonnées de plus).

Vous noterez que dès le lancement de l'application le shell présente un premier widget. Ce dernier existe dans le XAP principal. Il est découvert automatiquement par le mécanisme d'initialisation de MEF. Ce widget présente la particularité d'offrir un bouton « `Load Extensions` » (charger les extensions). Un clic sur ce bouton entraîne le chargement des widgets contenus dans le XAP

« extensions.XAP », ce dernier est donc téléchargé depuis le serveur puis « dépiauté » par MEF pour y découvrir les modules.

Cliquons sur le bouton :

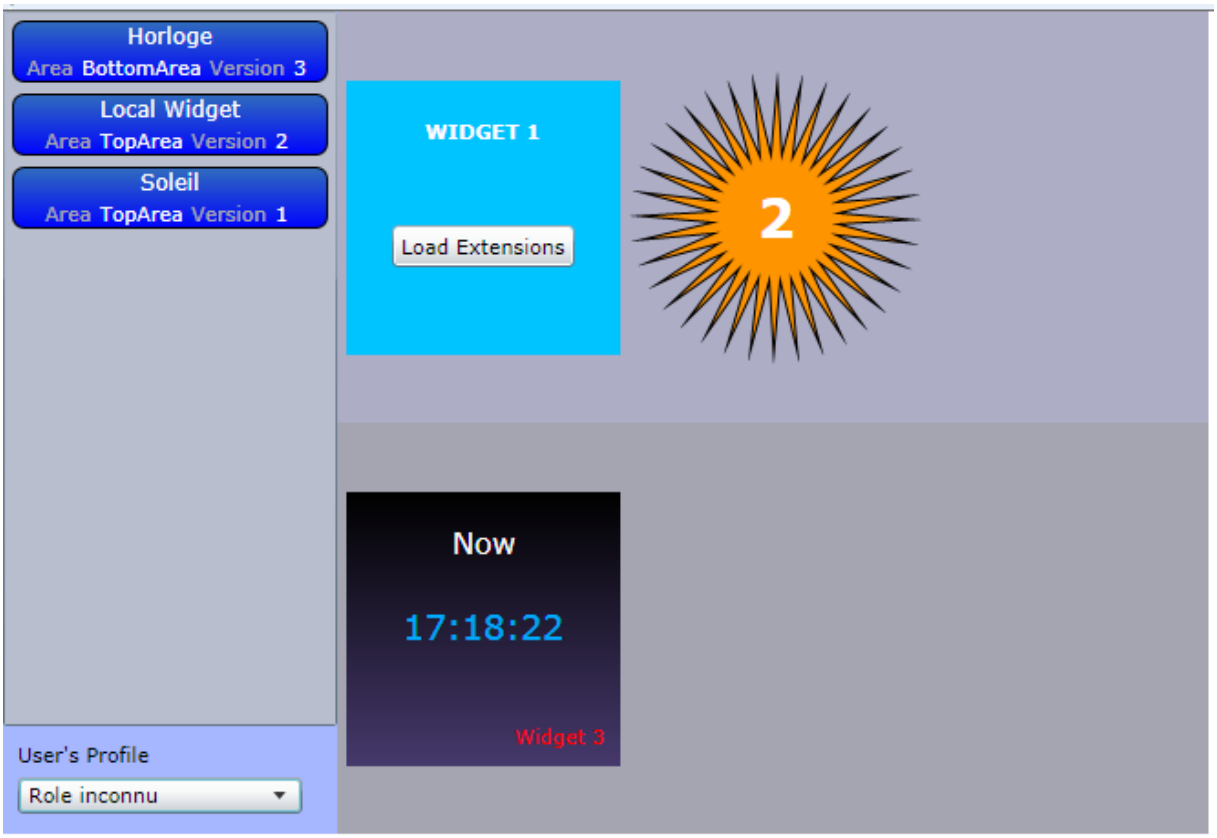


Figure 11 - DynamicXAP - Chargement des extensions

La capture ci-dessus nous montre le shell après que nous avons[1] cliqué sur le bouton « Load Extensions ».

La Listbox affiche désormais trois widgets (classés par ordre alphabétique de leur nom). Les trois widgets (le premier ainsi que les deux nouveaux) sont répartis entre le conteneur A et le conteneur B selon les indications portées par leurs métadonnées respectives (ce qu'on peut vérifier dans la Listbox).

Pour l'instant la partie gérant le profil utilisateur indique « *Rôle inconnu* ». Je concède volontiers que si je fais beaucoup d'efforts lors de l'écriture d'un article comme le présent, je n'hésite pas mêler anglais, français et franglais dans les applications exemples… *Mea culpa est*.

Cliquons sur la ComboBox du rôle et choisissons le rôle « *Administrateur* » :

[1] C'est bien l'indicatif qui s'emploie après « après que », même si le subjonctif, condamné par les grammairiens, est d'usage de plus en plus fréquent. Le subjonctif ne s'emploie qu'après « avant que ». D'où la confusion ? C'était la petite minute de « Maitre Capello » qui nous quitté cette année en mars à 88 ans…

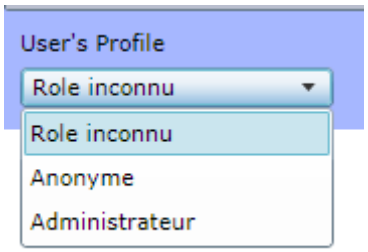


Figure 12 - DynamicXAP - Le rôle utilisateur

L'affichage devient :

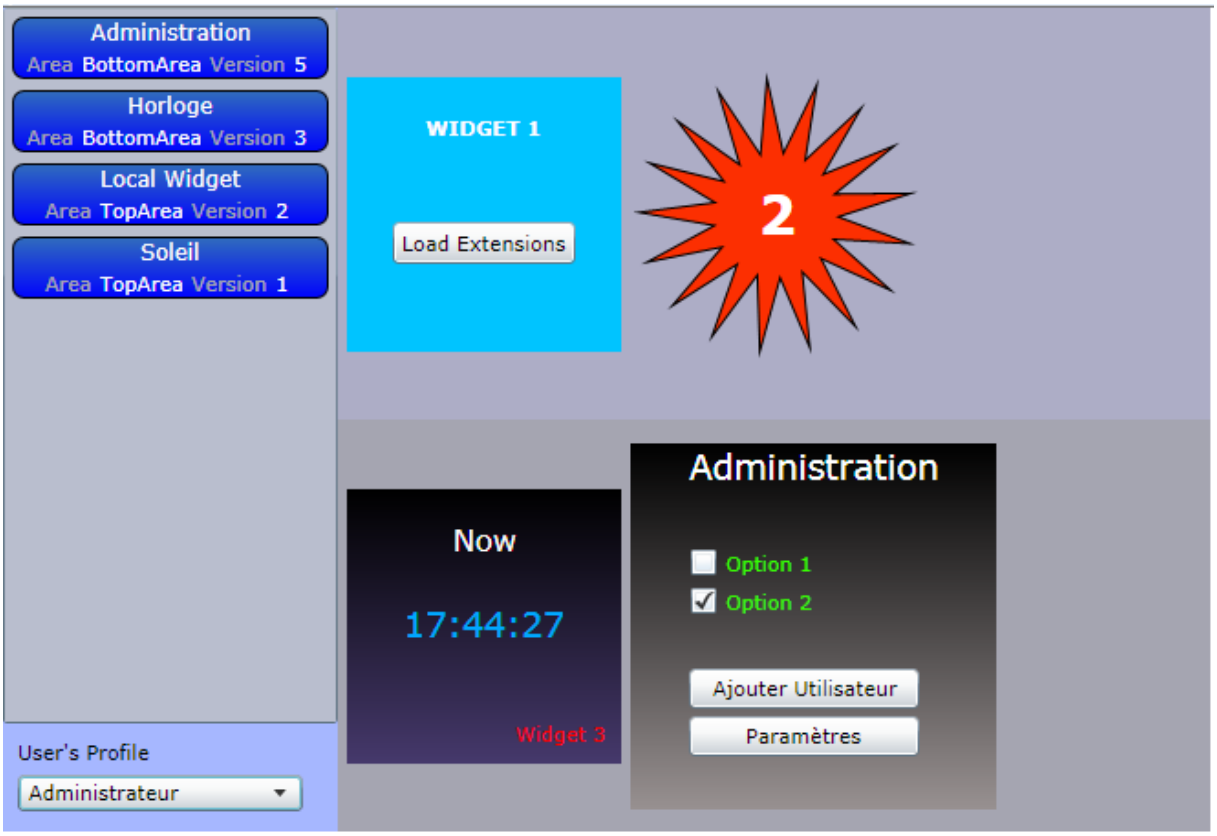


Figure 13 - DynamicXAP - Rôle Administrateur

La sélection du rôle Administrateur a déclenché l'affichage du widget d'administration (totalement fictif).

Si nous avions choisi le rôle « *Anonyme* », l'affichage serait :

Figure 14 - DynamicXAP - Rôle Anonyme

Le widget « `Anonymous Gadget` » est désormais affiché dans le conteneur A (puisque ses métadonnées précisent l'emplacement « `TopArea` ».

Bien entendu, le rôle de l'utilisateur serait géré en interne suite à un login dans une application réelle. Ici le choix est laissé dans l'interface pour simplifier la démonstration. S'il s'agit d'un autre type de sélection, comme l'affichage de certaines palettes d'outils par exemple, le choix d'une `ComboBox` se révèlerait mieux approprié.

De même, si vous testez directement l'application (ce que je vous incite à faire !) vous verrez rapidement que le choix d'un rôle n'annule pas la situation précédente et qu'au final on se retrouve avec les modules de tous les rôles affichés. Dans la réalité il faudrait ajouter une séquence de « nettoyage » qui cacherait les modules devenus inutiles.

Mais l'application est déjà bien assez complexe comme cela…

Ce que nous n'étudierons pas

Dès qu'une application grossit un peu elle doit faire face à de nombreux petits soucis accessoires qu'il faut bien gérer. Bien que centrée sur MEF mon application de démonstration n'échappe pas à la règle.

Elle contient ainsi une certaine quantité de code qui n'est pas directement lié à ce que je veux faire voir, code que je ne commenterai pas ici.

Néanmoins il m'a semblé important de vous indiquer les quelques « astuces » utilisées :

ListBox

Lorsqu'on fournit un `DataTemplate` à une `ListBox`, un phénomène fâcheux apparait : chaque instance du `DataTemplate` s'adapte à la taille de son contenu ce qui donne des items de tailles différentes, et ce n'est pas très beau. A cela une raison, l'un des conteneurs par défaut a son placement horizontal en mode « `Left` » ou lieu de « `Stretch` ». Il faut templater la `ListBox` pour modifier ce comportement. Voir mon billet : « *Largeur de DataTemplate et Listbox* » (http://www.e-naxos.com/Blog/post.aspx?id=ee204aa8-caa0-4211-92a4-9595fc20b712).

ComboBox

La `ComboBox` affiche une énumération (les différents rôles gérés par l'application), or les énumérations ne sont pas des collections et le Framework de Silverlight ne contient pas la méthode `GetValues` de WPF pour obtenir une liste des valeurs possibles.

De même, les valeurs internes de l'énumération ne sont pas forcément celles qu'on souhaiterait afficher.

La première chose à rappeler est que les énumérations sont parfaites pour matérialiser les états internes d'une application, mais qu'elles ne devraient jamais être utilisées dans les interfaces. Exposer directement des états internes dans l'UI créé un couplage code / UI non souhaitable et complique le travail de localisation. Mais parfois, comme dans l'exemple proposé, cela peut sembler pratique.

Ce dernier aspect est discutable lorsqu'on voit la quantité de code à ajouter (trois classes au moins plus le code XAML) pour finalement utiliser une énumération dans une `ComboBox`.

Regardez comment cela est fait, c'est toujours instructif, mais évitez d'utiliser cette astuce dans vos applications.

Voir mes billets suivants autour du même sujet :

« Conversion d'énumérations générique et localisation » (http://www.e-naxos.com/Blog/post.aspx?id=f197deab-5f45-4674-894d-a0d64c35ce8a)

Silverlight : « `Enum.GetValues` qui n'existe pas, binding et autres considérations » (http://www.e-naxos.com/Blog/post.aspx?id=8d8f4a13-698b-4666-b68e-d8fe9a7d0f1e)

Par où commencer ?

C'est la question qui se pose maintenant.

Par exemple, faut-il créer un projet avec le modèle fourni par le toolkit MVVM Light ou plutôt partir d'une solution vierge et ajouter MVVM par petites touches ?

On peut aussi se demander si MEF ne peut pas à lui tout seul remplacer un toolkit MVVM, les vues et les modèles de vue pouvant être connectés via des Import / Export sans l'aide de l'indirection d'un service locator comme MVVM Light le fait.

L'injection de dépendances façon Prism ne fait-elle pas un doublon avec l'esprit même de MEF ?

Et Unity ? Ça peut servir où cela serait mélanger trop de choses proches ensemble ?

Comment répartir le code entre combien de projets ? Cela aussi est important.

Et puis, faut-il utiliser MVVM Light, Jounce, Prism ou Caliburn.Micro ?

Toutes ces questions ramènent à réussir l'ordonnancement des différentes couches et à choisir correctement chaque librairie.

Prism, MEF, Jounce, Unity, MVVM Light, Caliburn ?

Ce n'est pas le sujet de cet article mais comment entrer dans le vif de ce même sujet si on n'a pas une idée, au moins vague, des différences principales entre ces toolkits ou frameworks alors même que nous allons en utiliser au moins deux ?

Alors en quelques mots :

MVVM Light

C'est un toolkit qui se focalise uniquement sur l'application de la pattern MVVM. Il fournit un système de messagerie et quelques classes de base (pour les ViewModels par exemple) et un service Locator simplifié permettant de créer un découplage entre Views et ViewModels. L'intérêt de MVVM Light est de ne faire que MVVM sans vouloir s'occuper de tous les problèmes possibles et d'être conçu pour assurer la « *blendability* » c'est-à-dire la capacité de disposer de données pour faciliter le design sous Blend, point sur lesquels les autres font l'impasse. MVVM Light, par son ancienneté (relative) utilise un style moins « moderne » que d'autres librairies comme Jounce par exemple. La Blendabilité est assurée par des mécanismes de code pur, là où Jounce utilise plus astucieusement les nouvelles possibilités de Blend (comme la déclaration de données de Design).

Prism

Il ne s'intéresse pas directement à MVVM mais fournit la plupart des outils permettant de mettre en œuvre la pattern et ces cousines tel MVC. Toutefois il s'occupe aussi d'autres questions (comme la gestion des commandes et des modules) et se destine plus à la gestion des applications basées sur des modules visuels avec, par exemple, la notion de régions dans lesquelles les modules viennent s'afficher. C'est plus un guide qu'un toolkit mais étant fourni depuis le début aussi sous la forme de code, Prism est devenu au fil du temps un toolkit à part entière tout en se refusant de l'être. Position ambiguë. Il est riche, bien pensé, mais complexe et il faut donc du temps pour le maîtriser.

MEF

C'est aujourd'hui une partie du Framework Silverlight et de .NET 4.0. Cela lui donne un statut différent, c'est un morceau du Framework tout simplement. MEF vise la modularité comme Unity mais favorise un modèle d'extensibilité « externe », via des modules extérieurs à l'application, de type plugin, même si l'approche « intérieure » est possible et même la plus simple. Il sait découvrir et charger des modules depuis des répertoires, il sait extraire des modules d'un XAP. Assez simple de prime abord, comme vous pouvez le constater au fil de cet article, ce n'est pas forcément simpliste non plus… Reste que MEF est une partie du framework et qu'il faut savoir s'en servir, un peu comme LINQ dans un autre domaine.

Unity

Cette librairie vise aussi à la modularité mais dans un esprit plus « statique » et s'utilise plus comme une technologie « intérieure », c'est-à-dire que personne ne voit, ne sait ni n'a besoin de savoir qu'une application utilise Unity. Alors que MEF se destine plutôt à la « croissance extérieure » comme stratégie. Unity est avant tout un toolkit d'injection de dépendances ce qui en fait un allié des projets utilisant beaucoup de tests unitaires ou de mocks. Les ressemblances avec MEF sont toutefois nombreuses (MEF est aussi basé sur l'injection de dépendances).

Jounce

Jounce est un nouveau toolkit pour Silverlight (là où les autres s'appliquent aussi à WPF et à WP7 généralement). Son but est de fournir des blocs de base pour l'écriture d'application de gestion modulaires qui utilisent les patterns MVVM et MEF. Jounce s'inspire ouvertement de Prism et de Caliburn.Micro avec lesquels il peut s'utiliser tout en pouvant s'en passer. Un peu ambigu aussi, car Jounce n'était au départ qu'une librairie « personnelle » de l'auteur qui se défendait de publier un « framework de plus », tout en le faisant… Mais c'est une librairie très intéressante à plus d'un titre. La version 1.0 qui vient d'être releasée en fait un concurrent sérieux de MVVM Light en apportant une note de « fraicheur » dans le traitement des problèmes. Jounce est tellement intéressant que je vais lui consacrer un prochain article.

Caliburn.Micro

Caliburn était presque plus complexe que Prism. Son auteur est reparti de zéro en reprenant l'essentiel. Caliburn.Micro n'est pas qu'une version simplifiée de Caliburn, c'est une nouvelle version qui remplace totalement Caliburn. Ce toolkit est un mélange de différentes solutions pour Silverlight, WPF et WP7 permettant de simplifier l'écriture des applications suivant MVVM, MVP ou même MVC. Son approche est différente des autres et contient beaucoup de bonnes idées. La nouvelle version « Micro » le rend certainement plus envisageable que l'ancienne, par trop complexe à mon goût (et même à celui de son auteur !).

L'heure du choix

En voici une belle brochette de toolkits, de quoi écrire des articles jusqu'à ma lointaine retraite !

Mais il faut faire des choix, c'est toujours difficile.

D'un autre côté, dites-vous qu'un monde où le choix n'existe pas n'est que très rarement un endroit où il faut bon vivre…

Personnellement j'évite de conseiller Prism pour sa complexité. Seules des équipes vraiment motivées, soudées, homogènes et bien formées peuvent s'en servir, cas de figure qui n'est pas le plus courant dans la réalité.

J'ai écarté Caliburn pour les mêmes raisons. La version Micro remet cette décision en cause et je conseille au lecteur de tester cette nouvelle mouture. J'ai hésité longtemps d'ailleurs entre Caliburn.Micro et Jounce pour le thème de mon prochain article. Même si mon choix se porte plutôt sur Jounce, que cela ne vous empêche pas de vous faire votre propre idée sur Caliburn.Micro !

Unity est séduisant. Ce toolkit et utilisé par Prism pour faire de l'injection de dépendance. Mais MEF est intégré au Framework aujourd'hui. Autant maîtriser et utiliser ce qui est fourni dans la boîte

avant d'aller chercher plus loin. Surtout que si les ressemblances sont nombreuses les nuances font, à mon sens, pencher la balance vers MEF.

En conclusion, le modèle le plus simple à comprendre et à mettre en œuvre aujourd'hui, même s'il ne fait pas tout (surtout parce qu'il ne fait pas tout !) consiste à marier MEF et MVVM Light.

Ce n'est pas la panacée, les autres toolkits sont pleins de bonnes idées aussi et de services absents de cette association. Mais par expérience je préfère conseiller à mes clients des toolkits assez faciles à apprendre, quitte ponctuellement à emprunter un morceau de code à d'autres toolkit (ils sont tous open source) plutôt que de les lancer sur des « monstres ». Lorsque leur équipe évolue, qu'untel s'en va, que deux nouveaux arrivent, il ne faut pas six mois pour avoir de nouveau une équipe opérationnelle. C'est à mon sens un des critères essentiels.

De fait, l'exemple qui va suivre utilisera MEF + MVVM Light. Chacun pourra adapter selon ses goûts et ses besoins. La présence de MVVM Light est tellement … légère que lui substituer un autre framework plus complexe ne posera pas de problème à ceux qui en maîtrisent déjà un.

> *Ayant traité de MVVM et MVVM Light dans de très longs articles, j'ai choisi de conserver MVVM Light ici principalement pour éviter de « mettre la charrue avant les bœufs ». Comme on le verra, le mariage MVVM Light +MEF n'est pas idéal. Jounce que j'aborderai dans un prochain article a l'avantage d'être construit d'emblée comme un toolkit MVVM se reposant sur MEF. En toute logique c'est cette solution là que nous devrions choisir. Mais dans cet article présentant MEF, je préfère utiliser MVVM Light sur lequel le lecteur pourra trouver toutes les explications dans mes précédents papiers alors que je n'ai pas encore abordé Jounce… Cela va venir, mais un article après l'autre !*

L'architecture globale

Le partage des tâches

Les toolkits et frameworks étant choisis, il faut décider de l'organisation générale et de la séparation des tâches.

Par exemple, MVVM Light offre un locator permettant de lier les vues aux ViewModels. Utilisant des champs statiques cette classe est la clé de voute de la *blendability* de MVVM Light. Le locator assure aussi le découplage entre les vues et l'implémentation de leurs ViewModels.

Conserver le locator de MVVM Light n'aurait pas de sens puisque dans ce cas il en serait terminé de la découverte dynamique de vues que nous allons mettre en œuvre. En revanche le découplage des vues et des ViewModels est essentiel, mais cela MEF le permet très bien puisqu'il gère l'injection de dépendances naturellement. Donc nous n'utiliserons pas le locator de MVVM Light.

Cependant, MEF ne propose pas de procédé de communication comme l'`EventAggregator` de Prism. Ni même de service de gestion des commandes. Mais cela existe dans MVVM Light (qui offre une messagerie pour les communications et la classe `RelayCommand` pour les commandes).

De même MVVM Light offre une classe de base pour les ViewModels qui supportent `INotifyPropertyChanged`. MEF ne se charge pas de cet aspect.

Finalement, le couple MEF + MVVM Light se révèle un choix intéressant. Deux toolkits assez simples, car concentrés sur un problème précis chacun, qui se marient bien puisque ne se recouvrant pas ou très peu.

L'organisation de la solution

La solution Visual Studio se décompose en 6 projets :

- `Contract`, un projet de type Silverlight Class Library, il contient la définition des contrats
- Shell, l'application principale Silverlight
- `Shell.Web`, l'application Web qui héberge la précédente ainsi que tous les XAP et le catalogue XML
- Extensions, le premier XAP d'extensions dynamiques. Projet Silverlight classique sans `MainPage` ni `App.Xaml`.
- `Extensions2`, le second XAP d'extensions dynamiques. Construit comme le précédent et contenant les modules pour le profil utilisateur « *anonyme* »
- `AdminExtensions`, troisième fichier XAP d'extensions dynamiques. Construit comme les deux précédents et contenant les modules pour le profil « `Admin` »

Le projet Silverlight principal (*Shell*) est créé comme une application Silverlight classique accompagnée de son projet Web d'hébergement (`Shell.Web`).

Le projet Web n'offre aucun service particulier, il sert juste de centralisateur via son répertoire `ClientBin` qui héberge tous les XAP et le catalogue. Il propose aussi une page de test html permettant d'exécuter l'application Shell. Au final le projet pourrait être déployé sur un serveur Linux, cela ne poserait aucun problème. Ni ASP.NET, ni aucun service Web ou WCF n'étant utilisé (ce qui réclamerait un serveur IIS).

Dans la pratique c'est l'application `Shell` accompagnée de `Shell.Web` qui ont été créées en premier. Le projet `Contract` a été écrit en créant le `Widget1` intégré à l'application `Shell`. Mais cela s'explique par le fait que j'ai construit l'exemple « à la volée » sans écrire d'analyse préalable. Dans un contexte réel le premier projet écrit serait `Contract` qui définit l'ensemble des contrats des modules.

C'est d'ailleurs par-là que nous allons commencer la visite du code.

Le code de l'exemple

Il est bien entendu hors de question de lister ici le contenu C# et XAML des six projets qui constituent la solution… cela serait vraiment fastidieux.

Les contrats

Dans une application de ce type il est indispensable de fixer les contrats dès le départ (quitte à faire quelques aménagements en cours de développement). Les contrats cela regroupe aussi bien les classes de base, les classes abstraites que les interfaces partagées par les modules, les interfaces fixant les métadonnées utilisées avec MEF ou les attributs MEF d'exportation personnalisés. Selon la taille du projet il peut devenir judicieux de séparer au moins en deux packages les contrats, d'un côté les contrats métiers ou fonctionnels et de l'autre les contrats spécifiques à MEF. Ici tout est regroupé dans le projet `Contract`.

C'est un projet de type *Silverlight Class Library*. Il référence l'unité `System.ComponentModel.Composition`.

> *Il est important de noter que les unités référencées, comme celles de MEF, sont déjà déployées par le projet principal `Shell`, de fait, pour diminuer la taille des XAP satellites ces unités sont marquées « **Copie Locale = False** » dans VS. La modularité qu'offre MEF diminue la taille de l'application principale, mais aussi celle des projets contenant les modules, chaque centaine de Ko gagnée rend au final l'application plus réactive et moins consommatrice de ressources réseau.*

IWidgetBehavior

Cette interface définit le comportement commun de tous les widgets. Dans une gestion modulaire il est important de *penser par contrat*. Le `Shell` ne peut pas, et ne doit surtout pas, connaître les classes de chaque module. *Il doit pouvoir manipuler les modules de façon générique*.

S'il existe plusieurs types de modules (ce qui est souvent le cas : modules visuels, modules de services…) il existe plusieurs interfaces de ce type.

Le code de l'interface est :

```
namespace Contract
{
    public interface IWidgetBehavior
    {
        bool IsActive { get; set; }
        bool IsSelected { get; set; }
        string WidgetName { get; set; }
        WidgetArea Area { get; set; }
        int Version { get; set; }
    }
}
```

Ce contrat définit des propriétés, pour la plupart « factices » uniquement là pour l'exemple. Dans la réalité il contiendrait aussi des méthodes permettant d'agir sur les modules ou leur demander d'effectuer les tâches pour lesquels ils sont conçus, voire d'énumérer les commandes de menu qu'ils contiennent et qui devraient être ajoutées au menu principal du `Shell`. Il n'y a pas de limite, tout dépend de l'application et de ce que doivent faire les modules.

Les propriétés `IsActive` et `IsSelected` ne servent pas dans l'exemple.

La propriété `WidgetName` retourne le nom du module tel qu'il doit se présenter à l'utilisateur (dans la `ListBox`). Il pourrait être issu directement du module, dans notre exemple il sera alimenté après coup par le `Shell` en fonction des métadonnées des widgets qui définissent déjà le nom à afficher.

La propriété `Area` est elle aussi un « repiquage » des métadonnées. Elle indique l'emplacement où le module doit être affiché (en haut ou en bas).

La propriété `Version`, elle aussi reprise des métadonnées et représente la version du module. Il est intéressant de pouvoir gérer cette information dans une application modulaire. Les parties pouvant venir de plusieurs XAP, ces derniers pouvant être dans le cache du browser ou d'un proxy, en cas de problème chez un utilisateur cette information de version peut s'avérer indispensable pour le

débogage. Ici ce n'est qu'un entier, dans la réalité on utilisera une version complète exprimée en 3 ou 4 groupes (ex : `2.3.58.599`).

WidgetInfo

Cette classe est une implémentation rudimentaire de l'interface précédente. Elle sera utilisée pour créer la liste affichée par la `Listbox`. Pourquoi cette redondance puisque les modules contiennent cette information et que les métadonnées fixent les principales informations de leur côté ?

En réalité, la liste des modules qui sera créée par MEF, comme dans les derniers exemples, sera une liste de `Lazy<T,M>`. Le champ `Value` pointera vers le `UserControl` définissant le widget. La lecture de `Value` entraîne le chargement du module (puisque nous sommes en mode *Lazy Loading*). Le module, une fois instancié appartient à la collection créée par MEF. Puisqu'il s'agit de modules visuels, ils seront affichés par le `Shell`. Chaque module étant un `UserControl`, la parenté visuelle de ce dernier sera donc fixée par le `Shell` qui l'ajoutera à l'un des deux conteneurs.

Si j'utilise maintenant la liste des modules retournée par MEF pour nourrir la `ListBox`, le `UserControl` contenu dans `Value` va être « capturé » par cette dernière. Or il appartient déjà à l'arbre visuel. Un objet ne peut pas être deux fois dans l'arbre visuel à des endroits différents.

Pour simplifier les choses et éviter ce problème, une liste de `WidgetInfo` sera créée par le `Shell` lorsque la liste des modules sera complétée par MEF. Cette liste, totalement déconnectée des widgets eux-mêmes pourra alors être affichée dans la `ListBox` sans que se pose le problème évoqué.

Ensuite, les widgets exposent, nous l'avons vu, un certain nombre d'informations (voire de méthodes) via l'interface `IWidgetBehavior`. L'implémentation de cette interface peut s'envisager de deux façons : soit chaque widget implémente l'interface directement, soit, surtout lorsque qu'il s'agit d'une liste de propriétés, chaque widget peut se contenter de créer un champ qui, lui, est une instance d'une classe qui implémente déjà tout le contrat, ce qui simplifie grandement les choses.

La classe `WidgetInfo` peut être utilisée de cette façon. L'un des widget que nous verrons s'en sert de cette façon pour la démonstration.

Le code de `WidgetInfo` :

```
namespace Contract
{
    public class WidgetInfo : IWidgetBehavior
    {
        public WidgetArea Area { get; set; }
        public string WidgetName { get; set; }
        public bool IsSelected { get; set; }
        public bool IsActive { get; set; }
        public int Version { get; set; }
    }
}
```

Une simple implémentation de `IWidgetBehavior` donc.

WidgetArea

Puisque chaque widget indique l'endroit où il doit être affiché, cette information sera stockée sous la forme d'une valeur puisée d'une énumération :

```
namespace Contract
{
    public enum WidgetArea
    {
        TopArea,
        BottomArea,
    }
}
```

Deux possibilités : le conteneur du haut ou celui du bas.

IWidgetMetaData

Nous touchons au premier élément « purement » MEF : l'interface définissant les métadonnées fortement typées.

```
namespace Contract
{
    public interface IWidgetMetadata
    {
        WidgetArea Area { get; }
        string Name { get; }
        int Version { get; }
    }
}
```

Zone d'affichage (`Area`), nom « *user friendly* » du widget (`Name`) et version du widget. Les métadonnées de notre exemple restent très limitées.

WidgetExportAttribute

Second élément purement MEF du projet `Contract`, `WidgetExportAttribute` est l'attribut d'exportation personnalisé qui sera utilisé pour décorer les `UserControls` utilisés comme des modules.

```
namespace Contract
{
    [MetadataAttribute]
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public class ExportWidgetAttribute : ExportAttribute
    {
        public ExportWidgetAttribute()
            : base(typeof(UserControl)) { }

        public WidgetArea Area { get; set; }

        public string Name { get; set; }

        public int Version { get; set; }

    }
}
```

L'attribut est une classe héritant de `ExportAttribute` et décorée d'attributs MEF.

`MetadataAttribute` indique que l'attribut d'exportation transporte aussi des métadonnées typées, cela n'est pas obligatoire mais semble plus judicieux. Dès lors qu'on décide de gérer des

métadonnées MEF, il semble cohérent de les intégrer à l'attribut d'exportation ce qui simplifie leur déclaration.

`AttributeUsage` fixe l'utilisation qui sera faite de l'attribut notamment en limitant les éléments qu'il sera à même de décorer. Ici j'ai fixé `AttributeTargets.Class`, ce qui signifie que l'attribut ne pourra servir qu'à exporter des classes (et non des propriétés ou des assemblages par exemple). `AllowMultiple` est à false parce que cet attribut ne peut être utilisé qu'une seule fois par classe qu'il décore.

Le constructeur ne fait qu'appeler le constructeur hérité en passant le type de `UserControl`. C'est sous cette classe que seront exportés les widgets. Il pourrait s'agir d'une classe mère s'ils descendaient tous d'une telle classe, ou bien, le plus souvent, il s'agira de l'interface définie pour fixer le comportement des widget. Dans notre projet il serait ainsi possible de faire apparaitre tous les widgets sous la forme de l'interface `IWidgetBehavior` (qu'il faudrait agrémenter d'une propriété retournant l'instance du `UserControl` pour pouvoir l'ajouter à l'arbre visuel).

Le reste du code est formé des propriétés qui reprennent les valeurs des métadonnées.

Le Shell

C'est l'application centrale, celle qui va recenser et afficher les modules. C'est une application Silverlight classique. Lors de sa création j'ai validé la création du projet Web correspondant (`Shell.Web`) qui servira de *shell* lui aussi mais de *shell technique* (sans rapport avec MEF) en regroupant tous les XAP dans son répertoire `ClientBin` et qui contiendra le catalogue de module que nous verrons plus loin.

Le `Shell` suit la pattern MVVM. Il utilise pour cela MVVM Light et référence l'unité `GalaSoft.MvvmLight.SL4`. Le projet n'est pas créé à partir du modèle fourni avec MVVM Light. L'unité est référencée, c'est tout, nous verrons plus loin les endroits précis où MVVM Light est exploité.

De la même façon le Shell référence l'unité principale de MEF plus l'unité spécifique à l'initialisation de MEF. La copie locale est laissée à `true`, ce seront les seules copies nécessaires de ces unités. Les autres XAP utilisant MEF n'auront besoin que d'une référence sans copie locale.

Ces unités sont :

- `System.ComponentModel.Composition`
- `System.ComponentModel.Composition.Initialization`

Le Shell contient une `MainPage` et un `App.Xaml` comme tout projet Silverlight de base. Il possède aussi trois sous-répertoires :

- `Helpers`, des unités de code utilitaires
- `ViewModels`, les ViewModels (de `MainPage` et de `Widget1`)
- `Widgets`, qui contient les widgets du XAP principal (une seul, `Widget1`)

Le `Shell` référence aussi le projet `Contract`. Ici aussi la copie locale est à `true` mais tous les autres XAP utilisant `Contract` n'auront besoin que d'ajouter la référence (avec copie locale à `false`).

IDeploymentService

L'une des particularités de l'exemple en cours est de permettre le chargement de modules depuis des XAP secondaires stockés sur le serveur.

Comme je l'ai déjà expliqué, MEF dispose de plusieurs services de découverte mais ils utilisent tous des procédés non exploitables sous Silverlight comme la recherche dans des répertoires. En mode OOB et avec *elevated trust* une application Silverlight peut éventuellement se servir de ces méthodes (je ne l'ai pas testé), mais une application OOB c'est plus du WPF que du Silverlight et, en tout cas ce n'est pas le Silverlight pour le Web dont je parle dans cet article. Et dans ce cas de tels accès directs à des répertoires ne marchent pas.

Pour Silverlight (Web) Microsoft a ajouté un système d'initialisation de MEF particulier, celui utilisé dans la première série d'exemples de cet article, et qui se borne à recenser les modules dans le XAP en cours.

C'est très bien, mais pas suffisant pour charger des modules externes, ce qui est l'un des avantages de MEF.

Etonnamment MS ne fournit pas, au moins pour l'instant, de méthode intégrée à MEF pour charger des XAP externes facilement.

Il est donc nécessaire de développer sa propre solution.

MEF utilise la notion de catalogue, voire de catalogues agrégés, pour créer ses compositions. Lorsqu'on utilise MEF comme je l'ai fait depuis le début on n'a pas à se focaliser sur ce genre de choses. Mais pour écrire un chargeur de XAP qui s'intègre à MEF, il le faut… Heureusement le site CodePlex de MEF nous donne quelques voies (voir http://mef.codeplex.com/wikipage?title=DeploymentCatalog) que nous pouvons améliorer.

Mais pour commencer il est préférable de définir une interface qui masquera l'implémentation qui pourra ainsi évoluer au fil du temps.

Cette interface c'est `IDeploymentService` :

```
namespace Shell
{
    public interface IDeploymentService
    {

        void AddXap(string relativeUri,
                    Action<AsyncCompletedEventArgs> completedAction);
        void RemoveXap(string uri);
    }
}
```

L'interface définit un contrat de deux méthodes, `AddXap` pour ajouter un XAP au catalogue de MEF en cours et `RemoveXap` qui permet d'enlever un XAP du catalogue.

On remarque que `AddXap` prend en paramètre une chaîne de caractères indiquant l'URI (relative) du XAP. Lorsque tous les XAP sont stockés ensemble dans le répertoire `ClientBin` de l'application

serveur, il suffit de passer le nom du XAP (ex : « `extensions.xap` »). Pour `RemoveXap` il faut utiliser la même chaîne que celle passé à `AddXap`.

Enfin, on note la présence d'un second paramètre à la méthode `AddXap`. Il s'agit d'une action de type `Action<AsyncCompletedEventArgs>` qui sera appelée automatiquement par le service de chargement une fois le XAP chargé et les modules découverts et instanciés. La signature nous rappelle que l'opération est de type asynchrone, le chargement du XAP n'est donc pas bloquant.

La question qui peut venir à l'esprit maintenant est « *comment fait-on pour connaître à l'avance le nom du XAP à charger puisque justement nous sommes en train de créer un système modulaire qui peut avoir à découvrir de nouvelles sources ?* ».

C'est une bonne question, et la réponse est simple : de base cela n'est pas possible. Une autre réponse plus satisfaisante est : certes, mais nous allons voir plus loin comment le faire en gérant un catalogue au format XML stocké sur le serveur. Réponse plus rassurante ☺

DeploymentCatalogService

Cette classe implémente le service de chargement de XAP défini par l'interface `IDeploymentService`.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using GalaSoft.MvvmLight.Messaging;

namespace Shell
{
    [Export(typeof(IDeploymentService))]
    public class DeploymentCatalogService : IDeploymentService
    {
        private static AggregateCatalog aggregateCatalog;

        readonly Dictionary<string, DeploymentCatalog> _catalogs;

        public DeploymentCatalogService()
        {
            _catalogs = new Dictionary<string, DeploymentCatalog>();
        }

        public static void Initialize()
        {
            aggregateCatalog = new AggregateCatalog();
            aggregateCatalog.Catalogs.Add(new DeploymentCatalog());
            CompositionHost.Initialize(aggregateCatalog);
        }

        public void AddXap(string uri,
                Action<AsyncCompletedEventArgs> completedAction = null)
        {
            DeploymentCatalog catalog;
            if (!_catalogs.TryGetValue(uri, out catalog))
            {
                catalog = new DeploymentCatalog(uri);
                if (completedAction != null)
                    catalog.DownloadCompleted += (s, e) => completedAction(e);
                else
                    catalog.DownloadCompleted += catalog_DownloadCompleted;
```

```
                catalog.DownloadAsync();
                _catalogs[uri] = catalog;
            }
            aggregateCatalog.Catalogs.Add(catalog);
        }

        static void catalog_DownloadCompleted(object sender,
                                              AsyncCompletedEventArgs e)
        {
            if (e.Error != null)
            {
                Messenger.Default.Send<NotificationMessage<Exception>>(
                      new NotificationMessage<Exception>(e.Error,"Exception"));
                //throw new Exception(e.Error.Message, e.Error);
            }
        }

        public void RemoveXap(string uri)
        {
            DeploymentCatalog catalog;
            if (_catalogs.TryGetValue(uri, out catalog))
            {
                aggregateCatalog.Catalogs.Remove(catalog);
            }
        }
    }
}
```

Il n'est pas évident de commenter ce code sans être obligé d'entrer dans les détails de fonctionnement de MEF (ce pourquoi la documentation de MEF est faite). Mais, de très haut, disons que la classe gère un catalogue agrégé (un catalogue qui peut contenir d'autres catalogues). Elle utilise les fonctions de base de MEF (comme `DeploymentCatalog`) pour charger et analyser le XAP, considéré comme un catalogue, et ajouter ce dernier au catalogue agrégé si tout a fonctionné.

L'action passée en paramètre à `AddXap` est exécutée en fin de traitement, qu'il y ait erreur ou non. Si aucune action n'est fournie (on peut passer `null`) le code fournit sa propre méthode de fin de traitement qui, en l'état, ne fait que détecter s'il y a eu erreur et, dans l'affirmative, lève une exception.

Dans cette version, lever une exception s'adapte assez mal à la situation. Quant à fournir à chaque appel de `AddXap` le code d'une action cela ne serait pas très pratique. J'ai choisi d'utiliser la messagerie de MVVM Light pour diffuser le message d'erreur.

Toute classe de l'application souhaitant être avertie qu'une erreur s'est produite peut ainsi s'abonner à ce message et le traiter comme elle le souhaite : pas d'action à écrire, aucun couplage avec aucun code existant, adaptabilité à tout code futur garantie.

Pour l'application `Shell`, c'est la vue principale `MainPage` qui s'occupe de récupérer le message et qui l'affiche tout simplement dans une boîte de dialogue, responsabilité conforme à MVVM. Une application réelle réagira de façon plus sophistiquée en gérant un log, en proposant une solution à l'utilisateur (ce qui est une bonne idée pour améliorer l'UX).

Dernier point sur `DeploymentCatalogService` : il faut l'initialiser. Cela s'effectue au plus tôt dans l'application, l'appel à la méthode `Intialize()` (qui est statique) est donc logiquement placée dans

App.Xaml sur l'évènement StartUp avant l'assignation de RootVisual (qui pourrait fort bien être un module découvert par MEF) :

```
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            DeploymentCatalogService.Initialize();
            this.RootVisual = new MainPage();
        }
```

Pour terminer sur cette classe, vous aurez remarqué qu'elle est exportée (elle est décorée par un attribut Export de MEF). Cela signifie que MEF recensera la classe, en créera une instance qui pourra être importée ailleurs par d'autres modules qui auront alors accès aux fonctions de chargement d'un XAP externe. C'est ce qui sera fait, nous le verrons plus loin.

MainPage

Il s'agit de la page principale de l'application, le véritable « *shell* » visuel.

Son code XAML peut être étudié depuis le code source fourni. Voici le code-behind :

```
namespace Shell
{
    public partial class MainPage : UserControl, IPartImportsSatisfiedNotification
    {
        public MainPage()
        {
            InitializeComponent();
            CompositionInitializer.SatisfyImports(this);
            DataContext = ViewModel;
            Messenger.Default.Register<NotificationMessage>(this,
                                        delegate { displayWidgets(); });
            Messenger.Default.Register<NotificationMessage<Exception>>(this,
              (n) => {if (n.Notification == "Exception") showException(n.Content); });
        }


        [Import]
        public MainPageViewModel ViewModel { get; set; }

        public void OnImportsSatisfied()
        {
            displayWidgets();
        }

        private void displayWidgets()
        {
            MainContainer.Children.Clear();
            SecondaryContainer.Children.Clear();
            Dispatcher.BeginInvoke(() =>
               { try
                   {
                     foreach (var widget in ViewModel.Widgets)
                      {
                        if (widget.Metadata.Area == WidgetArea.TopArea)
                                    MainContainer.Children.Add(widget.Value);
                        else SecondaryContainer.Children.Add(widget.Value);
                      }
                   }
                catch (Exception e)
                    { showException(e); }
                });
        }
```

```
        /* exceptions */

        private void showException(Exception exception)
        { Dispatcher.BeginInvoke(() =>
          MessageBox.Show("Exception:" + Environment.NewLine + exception.Message)); }
    }
}
```

La première chose à noter est l'importation du ViewModel. C'est MEF qui réalisera la connexion entre la Vue et son ViewModel. Nous n'utilisons donc pas le locator de MVVM Light pour cette opération. C'est un point essentiel dans la façon de « marier » MEF et un toolkit MVVM. Le découplage est assuré par MEF, inutile d'utiliser d'autres procédés.

Ensuite, on peut voir que le `DataContext` de la Vue est assignée dans le constructeur du code-behind. Il est initialisé en utilisant le ViewModel importé par MEF.

On notera aussi le support par la classe de l'interface `IPartImportsSatisfiedNotification` qui appartient à MEF. Elle définit un contrat très simple constitué d'une seule méthode `OnImportsStatisfied` qui est appelée lorsque les importations ont été satisfaites (qu'il y ait erreur ou non, ce qu'on peut savoir par l'argument de la méthode).

En supportant cette interface, notre Vue peut prendre en charge l'affichage des modules. La *recomposition* étant permise (grâce à `ImportMany(AllowRecomposition = true` dans le ViewModel), à chaque fois que de nouveaux modules seront ajoutés la méthode sera appelée, permettant à l'application de recréer l'interface visuelle en accord avec l'ensemble des modules découverts.

La Vue contient en outre du code non lié à MEF comme l'enregistrement dans son constructeur de deux notifications utilisant la messagerie de MVVM Light. La première notification demande à la vue de rafraîchir son affichage, la seconde est plus générique : tout code provoquant une exception peut, au lieu de laisser l'exception suivre son cours, la stopper et la transmettre via une notification. La Vue principale tient ici son rôle centralisateur (celui de *shell*) et affichera ainsi les messages d'erreur peu importe leur provenance.

Dans une application réelle cette séquence sera améliorée, avec par exemple le log de l'erreur, voire l'envoi d'un message via un Service Web ou autre afin que le serveur puisse monitorer les erreurs des postes clients.

Les affichages sont insérés dans des appels `BeginInvoke` de la classe `Dispatcher` qui s'assure que ces modifications du visuel de l'application sont réalisées par le thread principal. En effet, certains évènements (traitement des exceptions, demande de rafraîchissement de l'affichage) peuvent être initiés par des threads secondaires qui n'ont pas le droit d'accéder à l'interface visuelle. Des tests supplémentaires pourraient être effectués pour savoir si l'appel à `BeginInvoke` est nécessaire ou non. Dans cet exemple c'est majoritairement le cas et le temps d'affichage n'est pas critique. Il est généralement plus efficace de tester la nécessité d'un appel à `BeginInvoke` et l'éviter s'il n'est pas nécessaire. Tout dépend du contexte et c'est donc à vous de décider en fonction de ce que fait votre application.

Terminons sur un aspect essentiel lié à MEF : La Vue principale, en tant que classe, n'est pas exportée. C'est le niveau le plus haut, le *shell*. Cette particularité lui permet d'appeler dans son constructeur `CompositionInitializer.SatisfyImports(this)`, ce qui déclenche la satisfaction des importations. Cet appel ne peut pas être effectué par une classe qui est elle-même exportée. Ce qui interdit par exemple de placer ce code fonctionnel dans le ViewModel (posant d'ailleurs un petit problème quand à l'application stricte de MVVM qui voudrait que cela soit le ViewModel de la `MainPage` qui se charge de cet appel, ce qu'il ne peut pas faire étant lui-même exporté).

Les Bindings de la MainPage

Sans publier la totalité du XAML de cette page, retenons le binding de la `Listbox` :

```
<ListBox ItemsSource="{Binding WidgetsInfo}" …
```

`WidgetsInfo` est une propriété de type collection du ViewModel. C'est la liste des informations de chaque module affiché.

Le code XAML comporte en outre la définition d'un `DataTemplate` permettant de mettre en forme ces données.

D'autres définitions et bindings concernent la `ComboBox` du profil utilisateur. Pour ne pas tout mélanger nous verrons cela plus loin.

MainPageViewModel

Il s'agit du ViewModel de la page principale. La classe est exportée par un attribut MEF `Export`, ce qui permet à la Vue de récupérer son ViewModel automatiquement comme nous l'avons vu plus haut.

L'importation et l'exportation du ViewModel est ici réalisée directement sur le type de ce dernier. Cela est discutable puisque la Vue est ainsi obligée de connaître la classe du ViewModel créant un couplage qu'on cherche généralement à éviter, de surcroît avec MVVM.

Dans la réalité, je n'ai jamais rencontré de situation où, en cours de développement d'une application ou même après, d'un seul coup on se met à écrire un nouveau ViewModel ayant un nom de classe différent et qu'on souhaite lié à une Vue existante. C'est une situation purement théorique. C'est en cela d'ailleurs que certaines applications strictes de la pattern MVVM me paraissent très artificielles, voire néfastes. D'où, d'ailleurs, mon billet « Faut-il brûler la pattern MVVM ? » (http://www.e-naxos.com/Blog/post.aspx?id=64f3750f-dca5-487d-af3c-2e9d033e4b3f) que je vous invite à lire, le sujet débordant le cadre de cet article.

Je considère en effet qu'interdire à la Vue de connaître son ViewModel impose trop de contraintes au regard des faibles avantages purement théoriques de ce découplage. Une Vue est conçue pour un ViewModel et réciproquement, c'est la réalité. Rares sont les applications re-routant une Vue vers un autre ViewModel (ou réciproquement). Face à cette « interdiction » de MVVM qui apparaît purement artificielle, je préfère rester simple et conserver un code maintenable, quitte à violer la pattern (ce qui me fait penser que MVVM n'est pas une vraie pattern, mais simplement un principe

intéressant qu'il ne faut surtout pas prendre au pied de lettre comme on le ferait avec une vraie pattern de type Gang Of Four).

Revenons au ViewModel de la `MainPage`… et concentrons-nous sur ses fonctions vitales :

```
    [Export]
    public class MainPageViewModel : ViewModelBase, IPartImportsSatisfiedNotification
    {


        [ImportMany(AllowRecomposition = true)]
        public Lazy<UserControl, IWidgetMetadata>[] Widgets { get; set; }

        public IEnumerable<WidgetInfo> WidgetsInfo { get; set; }

        public void OnImportsSatisfied()
        {
            var l = new List<WidgetInfo>();
            foreach (var widget in Widgets)
            {
                var w = new WidgetInfo
                            {
                                    Area = widget.Metadata.Area,
                                    WidgetName = widget.Metadata.Name,
                                    IsSelected = false,
                                    IsActive = true,
                                    Version = widget.Metadata.Version
                            };

                l.Add(w);
            }
            WidgetsInfo = l.OrderBy(x => x.WidgetName).ToList();
            RaisePropertyChanged("WidgetsInfo");
            Messenger.Default.Send(new NotificationMessage("UpdateDisplay"));
        }
…
```

Tout d'abord la classe est décorée par `Export`. C'est comme cela qu'elle peut être importée par la Vue. Comme mentionné plus haut, l'exportation et l'importation sont effectuées en se basant sur le type du ViewModel ce qui impose que la vue connaisse cette classe. Cela n'est pas si grave dans la réalité, mais est en désaccord avec l'orthodoxie MVVM. Comme expliqué plus haut, je l'assume.

Pour rester conforme à la pattern MVVM il faudrait exporter les modules sous un nom de contrat de type Interface ou classe mère. Ici par exemple j'utilise `ViewModelBase` comme classe mère pour les ViewModel. Cette classe est issue de MVVM Light et offre quelques services comme le support de `INotifyPropertyChanged`. On pourrait fort bien exporter tous les ViewModel sous ce type (en ajoutant un nom de contrat MEF). On pourrait aussi définir une interface commune à tous les ViewModels, la faire supporter par chacun d'eux et les exporter sous le type de cette Interface. Dans ce cas la Vue récupèrerait un ViewModel typé faiblement (une interface ou une classe mère) et ne pourrait pas s'en servir autrement que pour initialiser son DataContext.

Cette façon de faire est un bon moyen de rester dans l'orthodoxie MVVM, qui de toute façon, fait que les interactions entre la Vue et son ViewModel sont basées uniquement le databiding. La Vue ne doit normalement pas avoir, dans son code-behind, à appeler directement du code de son ViewModel.

Même si cette approche me semble inutilement rigide, c'est un compromis qui pourra satisfaire ceux qui veulent coller au plus près de MVVM.

D'autres librairies orientées MVVM utilisent la notion de bootstrap décrivant les associations entre Vue et ViewModel ou d'autres procédés de ce genre. Le choix étant assez large, chacun fera en fonction de ce qui se rapproche le mieux de sa façon de voir les choses et d'interpréter MVVM.

L'ensemble des widgets est importé par les lignes suivantes selon un principe déjà étudié dans cet article :

```
[ImportMany(AllowRecomposition = true)]
public Lazy<UserControl, IWidgetMetadata>[] Widgets { get; set; }
```

Le ViewModel propose aussi la propriété suivante :

```
public IEnumerable<WidgetInfo> WidgetsInfo { get; set; }
```

C'est elle qui est liée par binding à `ItemsSource` de la `Listbox` de la Vue qui affiche la liste des modules chargés.

Le ViewModel supporte aussi l'interface MEF `IPartImportsSatisfiedNotification`, ce qui lui permet de fabriquer la liste `WidgetsInfo` à chaque fois que de nouveaux modules sont ajoutés.

D'autres fonctions sont prises en compte par le ViewModel mais nous les couvrirons plus loin dans l'article.

Le Widget1

C'est le premier module de notre application. Le seul qui se trouve défini dans le XAP principal. Ce n'est pas une obligation, cela permet uniquement d'illustrer cette possibilité. Le XAP principal contient très souvent des modules élémentaires indispensables au fonctionnement du `Shell` comme un dialogue de paramétrage par exemple (qui lui-même peut intégrer des modules variables, chaque module ayant des paramètres venant « s'imbriquer » automatiquement dans le dialogue général des paramètres).

Les modules (ou widgets, parties…) lorsqu'ils sont visuels peuvent bien entendu suivre la pattern MVVM et disposer de leur propre ViewModel, lui aussi découvert par le jeu des exportations et importations. C'est le cas du `Widget1` dont l'unique commande est gérée par un ViewModel à titre de démonstration.

Visuel et binding

Visuellement, `Widget1` se présente comme cela (sous VS) :

Figure 15 - Widget1 - Visuel

Un fond bleu, un titre figé (qui n'est pas issu des métadonnées) et un bouton. C'est ce dernier le plus important puisqu'il va déclencher le chargement volontaire d'un catalogue d'extensions.

Nous y reviendrons plus loin.

Le bouton est lié par binding à une commande exposée par le ViewModel du widget (via le `DataContext` initialisé par la Vue, voir le code-behind à la section suivante) :

```
<Button Content="Load Extensions" Height="23" HorizontalAlignment="Center" Name="button1"
VerticalAlignment="Bottom" Width="100" VerticalContentAlignment="Bottom"
Margin="25,0,25,48"
Command="{Binding LoadExtensions}"/>
```

Le code-behind

Le code-behind du `UserControl` « `Widget1` » commence ainsi :

```
namespace Shell
{

    [ExportWidget(Area = WidgetArea.TopArea, Name = "Local Widget", Version = 2)]
    public partial class Widget1 : UserControl, IWidgetBehavior,
                                                    INotifyPropertyChanged
    {
        public Widget1()
        {
            InitializeComponent();
        }

        [Import]
        public Widget1ViewModel ViewModel
        {
            get { return (Widget1ViewModel)DataContext; }
            set { DataContext = value; }
        }
…
```

On note la présence de notre attribut d'exportation personnalisé `ExportWidget` qui permet à la fois de marquer la classe comme étant exportée dans MEF et de fournir les métadonnées.

Le ViewModel est importé comme celui de la `MainPage`. Petite variante pour le `DataContext` : c'est le constructeur qui dans l'exemple précédent récupérait le ViewModel pour initialiser le `DataContext` alors qu'ici c'est la propriété `ViewModel` qui se charge de lire et d'écrire le DataContext.

Cette seconde approche est à utiliser partout. L'initialisation du `DataContext` n'est effectuée qu'une fois la propriété correctement importée par MEF (ce qui déclenche le setter) donc une fois la classe correctement construite. Dans le code de la `MainPage` on peut se demander comment la propriété `ViewModel` peut déjà être accessible dans le constructeur alors que pour l'initialiser MEF est bien obligé de créer l'instance (donc d'exécuter son constructeur). Qui de la poule ou de l'œuf ? C'est tout simple : la Vue principale n'est pas exportée… et c'est dans son constructeur qu'est appelée la méthode de composition de MEF. Tout naturellement, à la ligne suivant cet appel la propriété `ViewModel` est correctement initialisée… Cela n'est utilisable que dans la Vue principale, c'est pourquoi le `Widget1` montre la façon plus classique de procéder (et qu'on pourrait appliquer à la Vue Principale par souci d'homogénéité d'ailleurs).

Le problème de l'interface métier

Les widgets, comme nous l'avons vu, supportent l'interface métier `IWidgetBehavior`. Cette dernière ne sert pas à grand-chose dans l'exemple mais sa présence simule un cas réel : les plugins (ou widgets) sont le plus souvent pilotés via une interface. Et comme nous allons le voir, dans le contexte MEF + MVVM cela n'est pas forcément évident. D'où l'intérêt d'aborder ce sujet.

> *A noter qu'il faut comprendre ici « interface métier » au sens le plus large du terme. Selon ce qu'on mettra dans cette interface elle sera réellement une interface métier ou une simple interface. La nuance peut être importante car du code métier, avec MEF, aurait plutôt tendance à être dans des classes de services importées par les ViewModels par exemple. Vouloir placer des interfaces purement métier sur les widgets eux-mêmes est certainement un problème artificiel qui ne sert que les besoins de la démonstration. Dans la réalité l'architecture intègrera le code métier (et les interfaces correspondantes) dans une approche différente. Toutefois, même s'il ne s'agit pas de code métier « pur », le problème de piloter les widgets via des interfaces reste entier, ce qui justifie d'aborder la question.*

Dans notre exemple cette interface ne fixe que des propriétés mais il est évident que dans un cas réel elle devrait aussi proposer des méthodes permettant d'interagir avec les widgets. Cette interaction est très étendue et dépend du projet. Peut-être les widgets ont-ils besoin d'être synchronisés entre eux (ID d'une fiche client, article ?) ou bien doivent-ils être capables de retourner la liste des commandes qu'ils supportent pour qu'elles s'intègrent au menu du Shell, et bien d'autres choses encore…

Si le principe des interfaces et leur utilité n'ont pas besoin d'être présentés, ce qui nous interroge c'est de savoir « qui » va supporter cette interface ?

En effet, les widgets sont exportés en tant que `UserControl`. Ils sont importés par le `Shell` qui les affiche. Le `Shell` reçoit une énumération de widgets, soit directement des `UserControl` soit, comme dans cet exemple, des `Lazy<T,M>` qui possèdent une propriété `Value` retournant le `UserControl` ce qui revient au même au final.

Or, ces widgets sont des éléments d'interface visuelle. Il pourrait d'ailleurs s'agir de descendants de `Page` plutôt que de `UserControl` dans un système navigationnel. Ou tout autre type pour des services mais là le problème que je vais vous exposer ne se poserait pas.

Un module de service c'est une classe, du code. Sous MVVM ou non cela ne change rien. Sous MEF en revanche ce code sera plutôt placé dans des classes de services qui seront exportées pour être utilisées partout où cela est nécessaire.

Mais un module qui est affichable, un `UserControl` ou une `Page`, sous MVVM c'est une Vue. Et qui dit Vue dit deux choses : d'une part la Vue ne gère rien elle-même et d'autre part il existe un ViewModel qui se charge justement de cette gestion.

Si nous exportons bien les ViewModels qui sont ainsi importés par leurs Vues, et si nous exportons bien aussi les Vues (les `UserControl`) qui sont ainsi recensées et affichées par le `Shell`, ce dernier ne connait pas les ViewModels… Il pourrait les connaître au travers des Vues mais cela ne serait vraiment pas orthodoxe.

Si un module expose une interface (métier ou non), selon MVVM c'est le ViewModel qui devrait implémenter celle-ci, la Vue ne pouvant pas posséder de code métier ou fonctionnel, juste du code lié à l'affichage.

Toutefois, si nous suivons la pattern, le `Shell` n'aura pas accès à l'interface pour piloter les modules puisqu'il voit les Vues et non les ViewModels…

Et si nous implémentons l'interface au niveau de la Vue nous dérogeons à l'une des règles principales de MVVM.

Cornélien !

Dans une vision purement MVVM la solution consiste à implémenter l'interface au niveau des ViewModels et de baser tous les échanges entres les widgets et entre les widgets et le `Shell` sur des messages.

C'est une solution « propre » mais elle fait intervenir des messages asynchrones qui sont malgré tout très pénibles à gérer. Pour obtenir une solution plus utilisable il faudrait utiliser un Workflow **Jounce** qui offre un procédé très intéressant pour sérialiser les processus asynchrones (Pour plus d'information voir mon billet « Silverlight: Sérialiser les tâches asynchrones » - http://www.e-naxos.com/Blog/post.aspx?id=1429db2d-e248-4968-8356-403cdd11aede).

Une telle implémentation dépasserait le cadre de l'exemple en cours, et mélanger MVVM Light et Jounce ne serait pas judicieux, on opte pour l'un ou pour l'autre. Ici je n'aborderai pas la communication entre les widgets ou entre ces derniers et le shell. Mais c'est aussi ce qui fait la différence entre un article, même aussi long que peut-être le présent, et une véritable application qui demande des mois de travail !

Dans l'exemple en cours ce sont ainsi les Vues qui supportent l'interface « métier ». De fait l'interface est accessible au `Shell` puisqu'il possède la liste des modules créés par MEF et que ces modules sont les Vues. Cela permet de régler ponctuellement le problème mais il ne s'agit pas d'une solution digne d'un environnement de production, il faut en avoir conscience.

Ainsi, la vue doit gérer `INotifyPropertyChanged` pour signaler les modifications des valeurs des propriétés de l'interface, de même elle doit implémenter les propriétés, ce qui l'oblige à se

comporter comme un ViewModel et qui n'est vraiment pas satisfaisant en dehors de cette démonstration. C'est pourquoi on trouve dans la Vue de `Widget1` le code suivant :

```
#region IWidgetBehavior Members

        private bool isActive;
        public bool IsActive
        {
            get { return isActive; }
            set
            {
                if (isActive == value) return;
                isActive = value;
                doPropertyChanged("IsActive");
            }
        }

        private bool isSelected;
        public bool IsSelected
        {
            get { return isSelected; }
            set
            {
                if (isSelected == value) return;
                isSelected = value;
                doPropertyChanged("IsSelected");
            }
        }

        private string widgetName;
        public string WidgetName
        {
            get { return widgetName; }
            set
            {
                if (widgetName == value) return;
                widgetName = value;
                doPropertyChanged("WidgetName");
            }
        }

        private WidgetArea area;
        public WidgetArea Area
        {
            get { return area; }
            set
            {
                if (area == value) return;
                area = value;
                doPropertyChanged("Area");
            }
        }


        private int version;
        public int Version
        {
            get { return version; }
            set
            {
                if (version == value) return;
                version = value;
                doPropertyChanged("Version");
            }
        }
```

```
        #endregion



        #region INotifyPropertyChanged Members

        private void doPropertyChanged(string property)
        {
            var p = PropertyChanged;
            if (p == null) return;
            p(this, new PropertyChangedEventArgs(property));
        }

        public event PropertyChangedEventHandler PropertyChanged;

        #endregion
```

Widget1ViewModel

Cette classe est le ViewModel du widget. La seule fonction exposée par le ViewModel est la prise en compte de la commande de chargement.

```
namespace Shell
{
    [Export]
    public class Widget1ViewModel : ViewModelBase
    {
        public Widget1ViewModel()
        {
            LoadExtensions = new RelayCommand(
              () => Service.AddXap("extensions.xap",
                                   args =>
                                       {
                                         if (args.Error != null)
                                         MessageBox.Show(
                                                   "Error loading extensions"
                                                   + Environment.NewLine
                                                   + args.Error.Message);
                                        }));
        }

        [Import(typeof(IDeploymentService))]
        public IDeploymentService Service { get; set; }

        public ICommand LoadExtensions { get; private set; }

    }
}
```

La commande est exposée comme un `ICommand`, l'interface Silverlight gérée par les boutons. Toutefois la commande `LoadExtensions` est créée en partant de la classe `RelayCommand` de MVVM Light. Cette classe évite la création d'une classe implémentant `ICommand`. L'action à réaliser est enregistrée comme une expression Lambda (qui en contient une autre, la réponse à l'appel de `AddXap()`).

On remarque que le service de déploiement est importé (suivant son interface `IDeploymentService`), c'est par le biais de ce dernier que la commande va pouvoir ajouter l'extension « `extensions.xap` » via la méthode `AddXap()` du service.

La commande donne l'ordre au service de déploiement de charger le XAP en question et gère la réponse pour pister une éventuelle erreur. Si une exception est retournée par le service, la commande se contente de la transmettre à qui voudra bien la gérer, sous la forme d'un message MVVM Light.

On sait que le Vue principale du `Shell` s'est abonnée aux messages de notification de ce type et qu'elle affichera une boite de dialogue montrant le contenu de l'erreur. Mais cette connaissance n'est pas nécessaire.

> *A noter : lorsqu'on construit une application utilisant la messagerie MVVM il est intéressant de nommer toutes les notifications. Pour ce faire il est important de déclarer une énumération qui reprend le nom de chaque message. On utilise alors l'énumération (avec `ToString()`) comme texte de la notification. Le premier avantage est d'éviter qu'une faute de copie n'interdise le traitement du message, grâce à l'énumération le nom de la notification est toujours le même. Le second avantage est qu'il est plus facile de retrouver tous les morceaux de code qui font usage de tel ou tel message (un outil comme Resharper avec sa fonction 'Find Usage' est particulièrement utile dans un tel cas). Cela permet de vérifier que tous les messages émis sont bien reçus et traités au moins une fois par un récepteur.*

Le ViewModel n'est qu'un adaptateur pour la Vue, les traitements doivent être délégués au BOL ou à des services. C'est ce qui est fait ici pour le chargement du XAP.

Le service est importé via MEF, ce qui signifie que, quelque part, il existe une instance de ce service qui a été exportée… Revenez quelques pages en arrière, à la section abordant le `Shell`, et vous retrouvez l'interface `IDeploymentService` (page 54) ainsi que son implémentation qui est exportée.

Le chargement dynamique

Le mécanisme complet du chargement dynamique est le suivant :

1. `Widget1` est découvert et affiché par le `Shell`.
2. Ce module propose un bouton pour charger d'autres extensions. Pour ce faire il utilise le service de déploiement préalablement exporté.
3. Le service de déploiement va charger le XAP, l'analyser et ajouter au catalogue agrégé qu'il gère les nouveaux modules.
4. En supportant l'interface `IPartImportsSatisfiedNotification`, le ViewModel de la Vue principale est averti quand l'opération est terminée.
5. Il déclenche alors sa séquence de rafraîchissement de l'UI
6. L'utilisateur voit apparaître les nouveaux modules.

Le lecteur peut revoir la séquence visuelle simplifiée dans les captures d'écran de la section Le projet en action, page 41.

Les premières extensions

Les extensions chargées par le `widget1` se situent dans un projet à part, `Extensions`, Il s'agit d'une application Silverlight standard dont nous avons supprimé le fichier `App.Xaml` et la `MainPage`.

Ce projet ajoute deux références (avec copie locale = false) :

- `System.ComponentModel.Composition`

- Et `Contract`.

La première référence concerne MEF, la seconde notre projet regroupant les contrats, dont l'attribut d'exportation MEF personnalisé.

En dehors de cela le projet contient deux `UserControl`

- `Widget2`
- `Widget3`

`Widget2` se présente sous cette forme :

Figure 16 - Widget2 de Extensions.XAP

Une animation est lancée au chargement (l'étoile est animée et change de couleur)

`Widget3` se présente comme suit :

Figure 17 - Widget3 de Extensions.XAP

Il affiche l'heure (le `TextBlock` central) via un `DispatcherTimer` initialisé par la vue.

Ces deux widgets n'offrant qu'un comportement purement visuel il n'y a aucune raison de leur adjoindre un ViewModel. Vous n'en trouverez donc pas dans le projet `Extensions`.

Chaque Vue (chaque `UserControl`) est exporté de la façon suivante :

```
[ExportWidget(Area = WidgetArea.TopArea, Name = "Soleil", Version = 1)]
    public partial class Widget2 : UserControl, IWidgetBehavior, INotifyPropertyChanged
```

et

```
[ExportWidget(Area = WidgetArea.BottomArea, Name = "Horloge", Version = 3)]
    public partial class Widget3 : UserControl, IWidgetBehavior, INotifyPropertyChanged
```

On y retrouve le problématique support de l'interface `IWidgetBehavior`.

En dehors de ces quelques déclarations le projet `Extensions` ne contient rien d'autre. Il a été ajouté au projet `Shell.Web` (Propriétés / Applications Silverlight) afin que son XAP soit automatiquement placé dans le `ClientBin` de ce dernier (j'ai refusé en revanche l'ajout d'une page de test, ce qui ne sert pas à grand-chose ici).

Catalogue de modules

J'ai indiqué que cet exemple montrerait aussi comment gérer un catalogue de modules. Il est temps de passer à l'examen de ce problème.

Problème ? Oui, car en effet MEF ne sait pas gérer automatiquement le recensement des extensions sur un serveur depuis Silverlight. A cela une bonne raison : la sécurité très fermée de Silverlight ne l'autorise pas, que cela soit pour MEF ou qui que ce soit d'autre.

Dès lors, puisqu'il n'est pas possible de lire le contenu d'un répertoire distant, les outils de MEF permettant de cataloguer automatiquement les extensions sur disques ne sont plus utilisables.

On pourra regretter que Microsoft n'ait pas su faire une exception pour MEF, ce dernier faisant partie du Framework ce qui aurait tout simplifié. Mais créer des exceptions c'est aussi prendre le risque de créer des failles de sécurité…

Ainsi, nous nous retrouvons avec MEF, capable de découvrir des extensions, ce qui en fait en grande partie son intérêt, mais uniquement sous WPF. Sous Silverlight, de base, seuls les modules exportés et importés à l'intérieur du XAP principal sont gérés.

Nous avons vu que l'ajout d'un service de déploiement nous permettait de charger des XAP externes et de résoudre ce problème. Mais en partie seulement : les XAP chargés le sont nominativement, comme « `extensions.xap` ».

S'il faut connaître à l'avance le nom des XAP avant de les charger, il n'y a plus vraiment de « modularité » au sens plugin du terme.

Mais ne vous désespérez pas !

Tout comme nous avons su ajouter le service de déploiement pour charger des XAP externes, je vais vous montrer comment implémenter une solution tout à fait satisfaisante pour le problème de la découverte de nouveaux modules.

En réalité la solution est plutôt simple : Dans le répertoire `ClientBin` (ou l'un de ses sous-répertoire, à vous de voir) nous plaçons le fichier `Catalog.xml`.

Ce fichier se borne à énumérer tous les modules disponibles.

Comme Silverlight, avec le `WebClient`, permet facilement de télécharger un fichier qui provient de son propre serveur (ou d'un autre en suivant les règles d'un fichier de police à placer dans la racine du serveur), l'application peut récupérer le fichier XML et le traiter.

Lorsque les développeurs mettent à disposition un nouveau module il leur suffit d'ajouter une entrée dans le fichier XML situé sur le serveur. C'est une modification simple, légère et centralisée.

Le catalogue XML peut ainsi se borner à une simple liste de noms de fichiers XAP. Toutefois il est fréquent que les utilisateurs ne voient qu'une partie des modules. Par exemple en fonction de la licence qu'ils ont acquise ou bien en fonction d'un rôle (déterminé suivant le login). On peut penser à tout autre type de filtrage du même genre, tous reviennent à la même solution : il faut ajouter une ou plusieurs informations dans le catalogue pour que l'application puisse filtrer ce qu'elle peut (ou doit) charger.

Pour notre exemple j'ai choisi un filtrage sur le rôle de l'utilisateur. Comme je n'allais pas ajouter toute une gestion d'utilisateurs réelle pour cela, j'ai juste placé une `ComboBox` en bas à gauche du `Shell`. Par défaut le rôle est inconnu. Dès qu'on sélectionne un nouveau rôle (`Anonyme` ou `Administrateur`) le mécanisme de découverte, filtrage et chargement des nouveaux modules s'enclenche.

S'agissant d'une démo que j'ai voulu garder assez simple, le ménage n'est pas effectué lorsqu'on sélectionne un nouveau rôle. Les modules s'ajoutant aux précédents. Mais dans la réalité l'utilisateur ne pourrait pas choisir son rôle, c'est la gestion des utilisateurs, suite à un login valide, qui attribuerait un rôle fixe le temps de la session.

Je passerai sur la gestion de la `ComboBox`, ses deux bindings au ViewModel du `Shell` pour gérer à la fois la liste des rôles et le changement d'item sélectionné. Ce qui compte c'est ce qui se passe lorsqu'un rôle est sélectionné.

Chargement du catalogue

Lorsqu'on sélectionne un rôle dans la Vue, la propriété `CurrentUserRole` du ViewModel de la `MainPage` du `Shell` est mise à jour. Cette propriété est déclarée comme suit :

```
        private UserRole currentUserRole = UserRole.Unknown;
        public UserRole CurrentUserRole
        {
            get { return currentUserRole; }
            set
            {
                if (currentUserRole == value) return;
                currentUserRole = value;
                RaisePropertyChanged("CurrentUserRole");
                LoadModules(currentUserRole);
            }
        }
```

On note que le setter entraîne, outre la notification classique de changement de valeur, l'appel à `LoadModules(role)`.

C'est cette dernière qui va se charger de rapatrier le fichier XML, de l'analyser et de décider des modules à charger.

Pour utiliser le service de déploiement, le ViewModel déclare une importation de celui-ci :

```
        [Import(typeof(IDeploymentService))]
        public IDeploymentService Service { get; set; }
```

Vient ensuite le code de `LoadModules(role)` :

```
        private void LoadModules(UserRole userRole)
        {
            var wc = new WebClient();
            wc.OpenReadCompleted += (s, e) =>
            {
                if (e.Error!=null)
                {
                    Messenger.Default.Send(new NotificationMessage<Exception>
                                                (e.Error,"Exception"));
                    return;
                }
                var streamInfo = e.Result;

                var xElement = XElement.Load(streamInfo);

                var modulesList = from m in xElement.Elements("ModuleInfo")
                                  select m;
                if (!modulesList.Any()) return;
                foreach (var module in modulesList)
                {
                    var roleAttribute = module.Attribute("UserRole");
                    if (roleAttribute == null) continue;
                    if (string.Compare(roleAttribute.Value,userRole.ToString(),
                              StringComparison.InvariantCultureIgnoreCase)!=0)
                        continue; // user role mismatch, no module.

                    var moduleAttribute = module.Attribute("XapFilename");
                    if (moduleAttribute != null)
                    {
                        var moduleName = moduleAttribute.Value;

                        Service.AddXap(moduleName,null);
                    }
                }
            };
            wc.OpenReadAsync(new Uri("Catalog.xml", UriKind.Relative));
        }
```

Le principe reste assez simple, décomposons-le :

Un `WebClient` est déclaré. Son évènement `OnReadCompleted` se voit assigner une expression Lambda. L'ouverture asynchrone du catalogue XML est déclenchée par `OpenReadAsync`.

Cet appel finira par invoquer l'expression Lambda définie précédemment.

Le gestionnaire de `OnReadCompleted` commence par vérifier qu'il n'y a pas d'erreur. S'il y en a une, elle est transmise via la messagerie MVVM Light selon un principe que j'ai déjà expliqué.

S'il n'y a pas d'erreur Linq to XML est utilisé par analyser le fichier catalogue reçu. Les tests sont rudimentaires, je ne compare ici que la stricte égalité entre le nom du rôle de l'utilisateur géré par l'application et celui indiqué dans le catalogue. Il est évident que dans la réalité cela serait un peu plus sophistiqué. Mais le principe resterait le même.

Ensuite c'est facile, si l'entrée du catalogue correspond au filtre, un appel est fait au service de déploiement pour qu'il charge le XAP dont le nom a été récupéré dans le catalogue.

Ce chargement entraînera la notification déjà gérée par le ViewModel du `Shell`, notification qui permet à ce dernier de rafraîchir l'affichage, et donc de présenter les nouveaux modules ainsi découverts…

Le fichier `Catalog.xml` ressemble à cela :

```
<ModulesInfos>
  <ModuleInfo XapFilename="Extensions2.xap" UserRole="Anonymous" />
  <ModuleInfo XapFilename="AdminExtensions.xap" UserRole="Admin"/>
</ModulesInfos>
```

Il est bien évident que vous pouvez le personnaliser à souhait selon les besoins de votre application. La seule information véritablement indispensable est le nom du fichier XAP.

Et d'où viennent ces deux autres XAP d'ailleurs ?

Les extensions supplémentaires

Pour montrer la fonction de découverte dynamique de modules via un catalogue XML encore fallait-il disposer d'extensions…

Il m'a donc fallu rajouter deux autres fichiers XAP à la solution. L'un chargé pour le rôle « `Anonyme` » et l'autre pour le rôle « `Administrateur` ».

J'ai repris la même « recette » que celle expliquée pour l'extension « `extensions.xap` ».

`Extensions2.XAP` ne contient qu'un widget, `Widget4`, qui se présente comme cela :

Figure 18 - Widget4

`AdminExtensions.XAP` ne contient aussi qu'un seul widget, `Widget5`, dont l'aspect est un peu plus travaillé pour faire illusion :

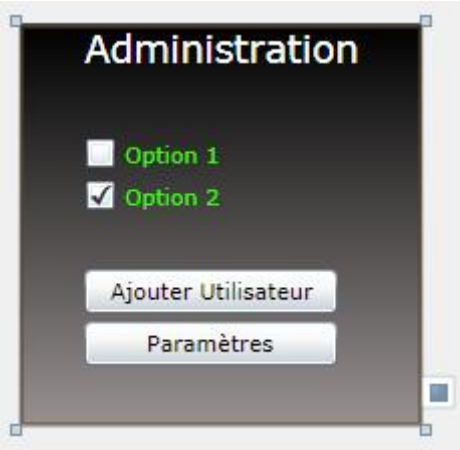


Figure 19 - Widget5

Aucune des commandes affichées n'est fonctionnelle, il s'agissait juste de donner un petit look « administration » à ce widget.

Il est évident que chaque XAP supplémentaire pourrait définir de nombreux widgets. Le catalogue XML pourrait aussi définir plusieurs XAP différents à charger pour un même rôle. Tout cela fait la différence entre une démonstration et une application réelle…

Conclusion

Il est temps de conclure. Arrivé ici et après toutes ces dizaines de pages j'ai l'impression de n'avoir qu'effleuré le sujet… Mais je pense aussi vous avoir donné les armes pour comprendre MEF et surtout pour l'utiliser dans de vraies applications en vous ayant aidé à résoudre certains problèmes délicats comme le chargement dynamique ou la découverte de nouveaux modules. D'autres problèmes sont soulevés et ne sont pas forcément traités, mais cela reflète la réalité de telles architectures d'une part, et celle d'un simple article qui doit bien s'arrêter quelque part…

Merci aux lecteurs qui auront atteint cette ligne de conclusion. Comme je le disais, ceux qui arriveront au bout vivants auront le droit à mes chaleureuses félicitations. Vous les méritez largement ☺

Olivier Dahan
MVP Silverlight