Formation, Audit, Conseil, Développement, UX

WinRT – Silverlight – WPF – Android – Windows Phone

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

MICROSOFT MVP Silverlight 2012, 2011,
MVP Client App Dev 2010, MVP C# 2009
Membre du Developer Guidance Advisory Council Microsoft

L'IoC dans MvvmCross – Développement Cross-Plateforme en C#

MvvmCross est un framework MVVM cross-plateforme permettant de développer des applications sous WinRT, Windows Phone, Android et iOS.

Version 1.0 Septembre 2012

Sommaire

Présentation de l'article 2

Inversion de Contrôle et Injection de Dépendance 2

De quoi s'agit-il ? 3

Les problèmes posés par l'approche « classique » 3

Ce qui force à utiliser une autre approche 3

La solution 4

Les implémentations de l'IoC 4

L'injection de Dépendance 4

Le localisateur de Service 5

L'IoC dans MvvmCross 6

Services et consommateurs 6

La notion de service 6

L'enregistrement d'un service, le concept 8

L'enregistrement d'un service, le code 9

La consommation des services 9

vNext 12

Conclusion 12

Présentation de l'article

MvvmCross est un jeune framework cross-plateforme dont j'ai déjà parlé sur Dot.Blog en l'intégrant à une stratégie de développement plus globale pour le développement cross-plateforme (voir la partie 1 ici : http://www.e-naxos.com/Blog/post.aspx?id=f4a7648e-b4a3-468d-8901-aa9a5b9daa2f).

La version actuelle, parfaitement utilisable, est en train d'être modifiée pour créer « vNext » qui sera disponible prochainement. Je présenterai un article complet sur le framework à cette occasion.

Pour l'instant je vous propose d'aborder un thème bien ciblé qu'on retrouve dans de nombreux frameworks : l'Inversion de Contrôle.

L'exemple sera pris dans le code de MvvmCross, comme un avant-goût de l'article complet à venir…

Inversion de Contrôle et Injection de Dépendance

MvvmCross, comme de nombreux frameworks MVVM, propose de régler certains problèmes propres aux développements modernes avec une gestion d'Inversion de Contrôle et d'Injection de Dépendance. Il n'y a pas de lien direct entre MVVM et ces mécanismes. Ici, c'est avant tout la portabilité entre les plateformes qui est visée. Les applications utilisant MvvmCross pouvant bien entendu utiliser l'IoC pour leurs propres besoins.

La partie chargée de cette mécanique particulière se trouve dans le sous-répertoire `IoC` du framework, dans chaque projet spécifique à chaque plateforme. Pour gérer cette fonctionnalité Stuart Lodge est parti d'un framework existant, OpenNETCF.IoC (http://ioc.codeplex.com/).

Ce framework est lui-même une adaptation de SCSF (Smart Client Software Factory, http://msdn.microsoft.com/en-us/library/aa480482.aspx) et de CAB de Microsoft mais en version « light » et cross-plateforme adaptée aux unités mobiles.

OpenNETCF.IoC propose une gestion de l'IoC par le biais d'attributs, soit de constructeur, soit de champ, un peu à la façon de MEF (mais en moins automatisé que ce dernier). Il sait créer lors du runtime des liens entre des propriétés ou des constructeurs marqués (importation) et des services exposés (exportation) via le conteneur central, sorte de registre tenant à jour la liste des services disponibles.

Bien qu'utilisant une partie du code d'OpenNETCF.IoC, MvvmCross ne l'expose pas directement et propose ses propres services d'IoC (construits sur le code modifié d'OpenNETCF.IoC). D'ailleurs dans « vNext » cet emprunt s'approche de zéro, seule la partie conteneur étant conservée. Cela ne change rien aux explications ici présentées puisque MvvmCross propose sa propre gestion d'IoC qui ne change pas, peu importe le code qui l'implémente (isolation et découplage ont du bon, pour tous les types de projets !).

Même s'il peut être intéressant de regarder de plus près le fonctionnement de OpenNETCF.IoC il n'est donc pas nécessaire de connaître celui-ci pour faire de l'IoC sous MvvmCross qui expose ses propres mécanismes (même si, en interne, ceux-ci utilisent pour le moment un code emprunté à OpenNETCF.IoC…).

Le lecteur intéressé spécifiquement par OpenNETCF.IoC est invité à consulter la documentation de cette librairie que je ne détaillerai pas ici, au profit d'explications sur ce qu'est l'IoC et comment MvvmCross a choisi d'offrir ce service, ce qui sera bien plus utile au lecteur je pense.

De quoi s'agit-il ?

Le problème à résoudre est assez simple dans son principe : imaginons une classe dont le fonctionnement dépend de services (au sens le plus large du terme) ou de composants dont les implémentations réelles ne seront connues qu'au runtime. Comment réaliser cet exploit ? C'est toute la question … qui trouve sa réponse dans les principes de l'Inversion de Contrôle et l'Injection de Dépendance.

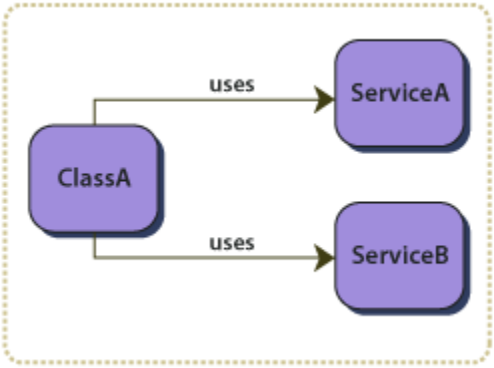

Figure 1 - L'inversion de Contrôle, l'objectif

Les problèmes posés par l'approche « classique »

Selon ce cahier des charges, dès qu'on y réfléchit un peu, on voit surgir quelques problèmes épineux si on conserve une approche « classique » :

- Pour remplacer ou mettre à jour les dépendances vous devez faire des changements dans le code de la classe (la classe A dans le petit schéma ci-dessus), cela n'est bien entendu pas ce qu'on veut (c'est même tout le contraire !).
- Les implémentations des services doivent exister et être disponibles à la compilation de la classe, ce qui n'est pas toujours possible dans les faits.
- La classe est difficile à tester de façon isolée car elle a des références « en dur » vers les services. Cela signifie que ces dépendances ne peuvent être remplacées par des mocks ou des stubs (maquette d'interface ou squelette non fonctionnel de code).
- Si on souhaite répondre au besoin, la classe utilisatrice contient alors du code répétitif pour créer, localiser et gérer ses dépendances. On tombe vite dans du code spaghetti…

Ce qui force à utiliser une autre approche

De cette mécanique de « remplacement à chaud » de code par un autre, ou tout simplement d'accès à un code inconnu lors de la compilation se dégage des besoins :

- On veut découpler les classes de leurs dépendances de telle façon à ce que ces dernières puissent être remplacées ou mises à jour sans changement dans le code des classes utilisatrices.

- On veut pouvoir construire des classes qui utilisent des implémentations inconnues ou indisponibles lors de la compilation.
- On veut pouvoir tester les classes ayant des dépendances de façon isolée, sans faire usage des dépendances.
- On souhaite découpler la responsabilité de la localisation et de la gestion des dépendances du fonctionnement des classes utilisatrices.

La solution

Dans l'idée elle est simple : il faut déléguer la fonction qui sélectionne le type de l'implémentation concrète des classes de services (les dépendances) à un composant ou une bibliothèque de code externe au code utilisateur des services et utiliser un mécanisme d'initialisation des classes qui sache « remplir les trous » (des variables) avec les bonnes instances.

Cette solution fait l'objet d'un pattern, l'Inversion de Contrôle (IoC).

Les implémentations de l'IoC

Il existe plusieurs façons d'implémenter l'IoC. Par exemple Prism en utilise deux qui sont l'Injection de Dépendance (Dependency Injection) et le Localisateur de service (Service Locator).

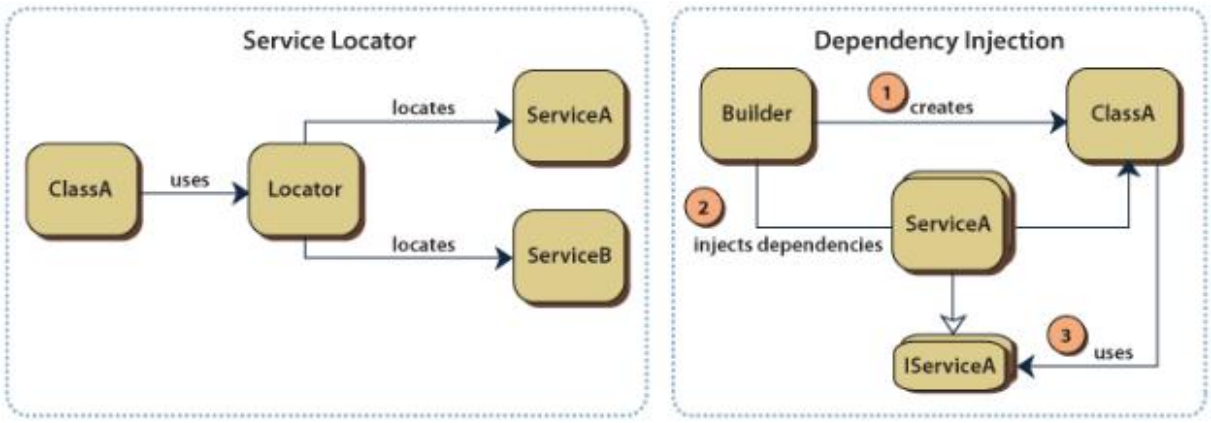

Figure 2 - Injection de Dépendance et Localisateur de Service

L'injection de Dépendance

L'injection de dépendance est une façon d'implémenter le principe de l'Inversion de contrôle, avec ses avantages et ses limitations, ce qui justifient parfois, comme Prism le fait, d'en proposer d'autres.

L'Injection de Dépendance consiste à créer dynamiquement (injecter) les dépendances entre les différentes classes en s'appuyant généralement sur une description (fichier de configuration ou autre). Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

C'est ce qu'on peut voir sur le schéma ci-dessus, à droite (Dependency Injection).

Il existe un « `Builder` » par lequel on passe pour créer des instances de `classA`. Ce builder ou factory recense les services proposés, comme ici le `ServiceA`, qui exposent des interfaces, ici `IServiceA`. La classe consommatrice des services (`classA`) se voit ainsi « injecter » l'implémentation concrète `ServiceA` lors de son initialisation. Elle ne connait pas cette classe concrète et ne fait

qu'utiliser une variable de type `IServiceA`. C'est cette variable que le `Builder` initialisera après avoir construit l'instance de `ClassA`. Le code de `classA` n'est donc pas lié « en dur » au code de `ServiceA`. En modifiant une configuration (dans le code ou dans un fichier) il est possible d'indiquer au `Builder` d'injecter un service `ServiceA` bis ou ter sans que cela n'impose une quelconque modification de `ClassA`.

L'inconvénient majeur de ce procédé est d'obliger la création de `ClassA` via le `Builder`. Il faut donc que le code applicatif soit pensé dès le départ en fonction de ce mode particulier d'instanciation qui peut ne pas convenir dans certains cas de figure.

Une implémentation possible de l'Injection de Dépendance se trouve par exemple dans MEF qui fait aujourd'hui partie de .NET. C'est d'ailleurs la solution retenue par le framework MVVM Jounce auquel j'ai consacré un très long article vers lequel je renvoie le lecteur intéressé.

MEF efface l'inconvénient de l'Injection de Dépendance telle que je viens de la décrire en automatisant le processus, c'est-à-dire en cachant le `Builder`. Les classes ou champs sont marqués avec des attributs (d'exportation ou d'importation) qui sont interprétés lors de leur instanciation. MEF se chargeant de tout.

MEF n'étant pas disponible sur toutes les plateformes, MvvmCross a été dans l'obligation de proposer une approche personnalisée ne se basant par sur MEF, au contraire d'un framework comme Jounce qui, en retour, n'existe que pour Silverlight…

Le localisateur de Service

Le localisateur de services est un mécanisme qui recense les dépendances (les services) et qui encapsule la logique permettant de les retrouver.

Le localisateur de services n'instancie pas les services, il ne fait que les trouver, généralement à partir d'une clé (qui permet d'éviter les références aux classes réelles et permet ainsi de modifier les implémentations réelles).

Le localisateur de services propose un procédé d'enregistrement qui liste les services disponibles ainsi qu'un service de recherche utilisé par les classes devant utiliser les dépendances et qui se chargent elles-mêmes de les instancier. Le localisateur de service peut ainsi être vu comme un catalogue de types (des classes) auquel on accède par des clés masquant le nom de ces types.

C'est ce qu'on peut voir dans la partie gauche du schéma ci-avant (Service Locator).

Le framework MVVM Light utilise une version très simplifiée du localisateur de service pour découpler les Vues des ViewModels (ce code se trouve dans le fichier `ViewModelLocator.cs`). En revanche MVVM Light ne propose pas d'Injection de Dépendance et son interprétation du localisateur de services restent assez frustre bien qu'étant souvent suffisante et ayant été construit dans l'optique du support de la blendabilité, ce que MvvmCross ne propose pas encore.

On le voit encore dans cet article comme dans les nombreux autres que j'ai écrits sur MVVM qu'hélas il n'existe pas encore d'unification des implémentations de ce pattern ni des services qui gravitent autour, chaque framework proposant « sa » vision des choses. Il ainsi bien difficile pour le

développeur qui ne les connait pas tous de faire un choix éclairé. Sauf en lisant régulièrement Dot.Blog, bien entendu !

J'en profite pour rappeler au lecteur que j'ai traité MVVM Light dans un très long article, et ici aussi je ne pourrais que renvoyer le lecteur intéressé à ce dernier (on retrouve facilement tous ces articles traitant de MVVM en faisant une recherche sur ce terme dans Dot.Blog).

L'IoC dans MvvmCross

Maintenant que les principes de l'IoC ont été exposés, il est temps d'aborder la façon dont MvvmCross aborde cette problématique.

Dans le principe nous avons vu que MvvmCross utilise un code modifié de OpenNETCF.IoC, mais que ce dernier n'est pas directement exposé au développeur, il est juste utilisé en interne. A la place MvvmCross propose d'autres procédés.

Quels sont-ils ?

Services et consommateurs

Le monde de l'IoC est très manichéen : les classes peuvent être rangées en trois catégories, celles qui proposent des services, celles qui les consomment, et … celles qui ne participent pas l'IoC !

Comme dans toutes les sociétés il existe tout de même des individus plus difficiles à classer que les autres, notamment ici les classes qui exposent des services tout en en consommant d'autres. Cela est parfaitement possible et c'est la gestion d'IoC qui s'occupe de détecter et d'éviter les boucles infinies qui pourraient se former. Par exemple grâce à l'utilisation de singletons. Mais la charge de ce contrôle de la logique de l'application reste le plus souvent au développeur.

La notion de service

Un service est proposé par une classe. Une même classe peut, éventuellement, offrir plusieurs services, mais cela n'est pas souhaitable (le remplacement d'un service entraînant en cascade celui des autres, situation qui fait perdre tout son charme à l'IoC).

En revanche plusieurs classes ne peuvent pas au même moment proposer le même service. C'est une question de logique, le gestionnaire d'IoC ne saurait pas quelle instance ou quelle classe retourner (sauf à faire un tirage au sort ce qui pourrait déboucher sur un concept intéressant de programmation régit par les lois du chaos…).

Il y a deux façons de proposer un service sous MvvmCross : soit en enregistrant une instance d'une classe précise (un singleton), soit en enregistrant un type concret.

La nuance a son importance et sa raison d'être.

Certains services sont « uniques » par nature. Par exemple on peut proposer, comme MvvmCross le fait, un service de log, ou bien le support du cycle de vie d'une application (tombstoning notamment). On comprend aisément que de tels services ne peuvent exister en même temps en plusieurs exemplaires dans la mémoire de l'application, cela n'aurait aucun sens. Une application n'a qu'un seul cycle de vie, et, pour simplifier les choses, un système de log ne peut pas exister en différentes versions à la fois.

Autant le premier cas (cycle de vie) est une question de logique pure, autant le second (le log) est un choix assumé par MvvmCross. On pourrait supposer qu'il soit possible d'enregistrer plusieurs services de logs (une trace dans un fichier texte et un système d'envoi de messages via Internet par exemple) et qu'ils soient tous appelés sur un même appel au service de log.

Cela compliquerait inutilement l'implémentation du framework alors même que l'un de ses buts est d'être le plus léger possible puisque tournant sur unités mobiles. On comprend dès lors cette limitation de bon sens.

Ainsi ces services uniques doivent-ils être proposés par des singletons (vrais ou « simulés » par la gestion de l'IoC).

Le développeur prudent implémentera ces services sous la forme de singletons vrais, c'est-à-dire de classes suivant ce pattern. Toutefois la gestion de l'IoC est capable de reconnaitre un tel service unique et sait fournir aux consommateurs l'unique instance enregistrée. De ce fait la classe de service peut ne pas être un singleton vrai sans que cela ne soit vraiment gênant (sauf qu'il n'existe pas de sécurité interdisant réellement la création d'une autre instance par d'autres moyens, situation qu'un singleton vrai verrouille).

D'un autre côté il existe des services qui ont tout intérêt à être ponctuels ou du moins à pouvoir exister en plusieurs instances au même moment. Soit qu'ils sont utilisés dans des threads différents gérant chacun des tâches distinctes, soit pour d'autres raisons.

C'est le cas par exemple du service d'effet sonore proposé par MvvmCross. A l'extrême on peut imaginer un jeu gérant plusieurs objets sur l'écran, chacun émettant un son différent. Le service d'effet sonore ne doit pas être un singleton qui ne jouerait qu'un seul son à la fois.

MvvmCross propose de nombreux services, tous, en dehors du Log et de la gestion du cycle de vie de l'application sont de types multi-instance.

Le choix est parfois évident, comme pour les effets sonores, il l'est moins pour d'autres services comme la composition d'un numéro de téléphone ou le choix d'une image dans la bibliothèque de la machine. Ces tâches imposent un dialogue ou un fonctionnement qui ne peuvent exister qu'en un seul exemplaire au même moment.

Alors pourquoi leur donner la possibilité d'être multi-instance ?

La raison est fort simple : qui dit capacité multi-instance suppose le support du cas de zéro instance, là où le singleton oblige un minimum d'une instance…

La création de services uniques sous la forme d'instances statiques est très logique pour des services ne pouvant exister qu'en un seul exemplaire à la fois, mais hélas cela impose de créer au moins instance dès le départ, le singleton.

Or, beaucoup de services utilisent des ressources machine, la gestion des sons par exemple ou la création d'un email ou l'activation d'un appel téléphonique. Cela n'aurait ni sens ni intérêt d'instancier toute une liste de services de ce type alors même que l'application n'en fera peut-être jamais usage et que cela va gonfler inutilement sa consommation (mémoire, CPU, électrique).

On comprend alors mieux pourquoi la majeure partie des services exposés par MvvmCross est de type « classe » et non pas « instance ». La véritable raison est donc bien plus motivée par la possibilité d'avoir zéro instance que d'en avoir plusieurs.

L'enregistrement d'un service, le concept

Je dis de type « classe » ou « instance » car c'est de cette façon que MvvmCross va enregistrer un service et faire la différence entre ceux qui peuvent exister en zéro ou plusieurs exemplaires et ceux qui existent dès le départ, et au maximum, en un seul exemplaire.

Quand je dis « classe », je dois dire plus exactement « interface » puisque l'IoC de MvvmCross se fonde sur l'utilisation d'interfaces.

En effet, rappelons-nous que tout cela n'est pas seulement un libre-service dans lequel on vient piocher mais qu'il s'agit avant tout de créer un découplage fort entre fournisseurs et consommateurs de services en utilisant les concepts propres à l'IoC. Les interfaces sont la clé de voute de ce découplage, le reste n'est qu'enrobage.

De fait, l'enregistrement d'un service s'effectue d'abord en passant le type de son interface à la méthode chargée de cette fonction.

En second lieu seulement on passera soit l'instance du singleton, soit le type de la classe d'implémentation.

L'IoC, comme je le disais bien plus haut, peut être vu comme un registre, un dictionnaire dont la clé est le type de l'interface. On revient aux principes de l'IoC exposés précédemment, par exemple au localisateur de service qui utilise des clés pour masquer les noms des classes réelles.

Pour enregistrer un service unique il faut avoir accès au premier de ceux-ci : le gestionnaire d'IoC.

MvvmCross s'initialise en créant le gestionnaire d'IoC d'une façon qui elle-même autorise ce service à être proposé en différentes implémentations (cross-plateforme donc).

Mais Qui de la poule ou de l'œuf existe en premier me demanderez-vous ?

L'œuf.

Non, je plaisante, tout le monde sait que c'est la poule.

MvvmCross utilise lors de son initialisation un autre procédé (un marquage via un attribut) respectant le concept de l'IoC qui lui permet de chercher dans l'espace de noms de l'application où se trouve le service d'IoC.

Il ne peut y en avoir qu'un seul et une exception est levée si d'aventure deux se trouvaient en présence ou si aucun ne pouvait être trouvé. Stuart Lodge a voulu ici se protéger contre tout changement ou évolution de son propre framework et s'est appliqué à lui-même les principes qu'il a adoptés par ailleurs et qu'il propose via la gestion des IoC dans MvvmCross. Dans les conditions normales d'utilisation aucune des deux situations ne peut se présenter bien entendu. MvvmCross est fourni avec un seul service d'IoC qui fait partie du framework.

Donc il existe un premier service qui est le gestionnaire des services.

Comme tout service il faut pouvoir y accéder, ici impossible de faire appel à l'IoC puisque c'est justement lui auquel on veut accéder…

C'est au travers de méthodes d'extension que le service d'IoC va être disponible. Mais je vais y revenir.

Comme je l'ai évoqué, MvvmCross enregistre lui-même de nombreux services qui sont directement accessibles dans toutes les applications. J'ai parlé du service de log (`Trace`), de celui émettant des évènements pour gérer le cycle de vie de l'application, mais aussi du sélectionneur d'image, du composeur de numéro de téléphone, etc.

Toutefois une application peut décider d'exposer d'autres services que les ViewModels pourront exploiter comme, par exemple, un service de remontée des erreurs de fonctionnement. D'autres types de services peuvent aussi être fournis pour remplacer des briques du framework. On peut trouver dans les exemples fournis avec MvvmCross quelques utilisations de ce type.

L'enregistrement d'un service, le code

Maintenant que savons ce que sont les services, pourquoi il existe deux types d'enregistrement et comment accéder à l'IoC, il ne reste plus qu'à voir quel code permet d'effectuer dans la pratique ces enregistrements.

Le premier cas est celui du service unique. Comme le service de gestion du cycle de vie des applications. Il est enregistré comme suit dans le code de MvvmCross :

```
RegisterServiceInstance<IMvxLifetime>(new MvxWindowsPhoneLifetimeMonitor());
```

`RegisterServiceInstance` permet d'enregistrer l'instance unique d'un service unique. L'enregistrement commence par le type de l'interface (la clé pour le dictionnaire de l'IoC), ici `IMvxLifeTime` puis est complété par un paramètre acceptant une instance de tout objet (du moment qu'il supporte l'interface déclarée).

Le second cas est celui d'un service « optionnel », ce terme convenant mieux que celui de « multi instance » puisqu'il souligne le fait que le service peut ne jamais être instancié. Prenons l'exemple du service de sélection d'image, MvvmCross l'enregistre de la façon suivante :

```
RegisterServiceType<IMvxPictureChooserTask, MvxPictureChooserTask>();
```

`RegisterServiceType` permet cette fois-ci d'enregistrer un type et non une instance. On trouve comme précédemment en premier lieu l'interface qui est exposée, ici `IMvxPictureChooserTask`. Mais au lieu de supporter un paramètre permettant de passer l'instance du service, la méthode générique accepte un second type, celui du service (implémentation concrète), ici `MvxPictureChooserTask`. Aucun paramètre n'est passé, la déclaration générique des deux types, l'interface et la classe de service, suffisent à la méthode pour faire son travail.

La consommation des services

La consommation des services sous MvvmCross s'effectue en demandant à l'IoC de retourner une instance de celui-ci après lui avoir fourni la clé, c'est-à-dire le type de l'interface, du service.

Les méthodes permettant d'accéder aux services ne sont pas directement héritées, elles sont implémentées sous la forme de méthodes d'extension pour le type `IMvxServiceConsumer`. Le principe reste le même que pour l'enregistrement des services.

Pourquoi ce choix ?

Plusieurs possibilités étaient envisageables pour que le code de l'application puisse accéder aux services. Un moyen simple aurait été de proposer une classe statique regroupant les méthodes d'accès aux services quels qu'ils soient.

C'est finalement ce qui est fait, mais au lieu d'exposer une telle classe statique, il existe des méthodes d'extension qui lui donne accès, créant ainsi une indirection plus sophistiquée.

Toute classe qui désire consommer un service doit le demander en supportant l'interface générique `IMvxServiceConsumer<TService>`. L'héritage multiple des interfaces existant en C#, il est donc possible pour une classe consommatrice de services d'hériter autant de fois que nécessaire de `IMvxServiceConsumer` en indiquant à chaque fois le type de service consommé.

Ce choix possède toutefois un inconvénient majeur : puisque la classe consommatrice « supporte » des interfaces, elle doit les implémenter, la délégation d'implémentation d'interface n'existant pas en C#.

C'est une situation bien embêtante…

La façon de régler le problème est intéressante : il suffit que l'interface en question soit vide et de créer des méthodes d'extension qui ne sont accessibles qu'aux types supportant `IMvxServiceConsumer`…

Parmi les exemples de code fournis, on peut trouver un code de ce type :

```
public class BaseViewModel
        : MvxViewModel
        , IMvxServiceConsumer<IMvxPhoneCallTask>
        , IMvxServiceConsumer<IMvxWebBrowserTask>
        , IMvxServiceConsumer<IMvxComposeEmailTask>
              , IMvxServiceConsumer<IMvxShareTask>
        , IMvxServiceConsumer<IErrorReporter>
```

La classe `BaseViewModel`, la classe de base pour tous les ViewModels de l'application « Conférence », hérite d'abord de `MvxViewModel`, puisque c'est un ViewModel, puis elle hérite « faussement » de `IMvxServiceConsumer` autant de fois que nécessaire pour déclarer les services qu'elle désire consommer comme `IMvxPhoneCallTask` ou `IMvxComposeEmailTask`, services de base de MvvmCross, ou `IErrorReporter` qui est un service propre à cette application.

Dès lors, cette classe peut utiliser les méthodes d'extension permettant d'accéder à l'IoC. Ce qu'elle fait en exposant à ses héritiers des accès simplifiés. Voici un court extrait du code de la classe exemple :

```
        protected void ReportError(string text)
        {
            this.GetService<IErrorReporter>().ReportError(text);
        }
```

```
        protected void MakePhoneCall(string name, string number)
        {
            var task = this.GetService<IMvxPhoneCallTask>();
            task.MakePhoneCall(name, number);
        }
```

La méthode `ReportError(string text)` sera disponible à tous les ViewModels de l'application (qui hériteront de cette classe), méthode qui appelle `GetService<IErrorReporter>()` pour obtenir le service de gestion des erreurs propre à cette application. `GetService` est disponible via les méthodes d'extension pour les types supportant `IMvxServiceConsumer<TService>`.

La même chose est faite pour le service MvvmCross qui permet d'initier un appel téléphonique.

Très souvent les applications se contentent d'utiliser `GetService` dont le comportement dépend du type de service (unique ou optionnel). Si le service est unique (enregistré par son instance) c'est l'instance unique qui est retournée systématiquement, si le service est optionnel (enregistré par son type), c'est à chaque fois une nouvelle instance qui est renvoyée.

Il est donc nécessaire de porter une attention particulière à l'utilisation des services optionnels pour éviter le foisonnement des instances en mémoire…

Il existe deux autres méthodes d'extension pour les consommateurs de services :

- `Bool IsServiceAvailable<TService>()`, qui permet de savoir si un service est supporté ou non.
- `bool TryGetService<TService>`, qui permet d'éviter la levée d'une exception en cas d'impossibilité d'accès au service.

On notera que puisqu'il existe en réalité une instance statique fournissant les services de l'IoC « cachés » derrière les méthodes d'extension il est donc théoriquement possible de se passer de la déclaration des interfaces des services consommés et d'accéder directement à cette instance statique.

Et c'est en effet possible, depuis n'importe quel code il est par exemple possible d'écrire :

```
var c = MvxServiceProviderExtensions.GetService<IMvxTrace>();
            c.Trace(MvxTraceLevel.Error, "test", "coucou");
```

Il suffit de déclarer le bon "`using`" pour accéder au code des méthodes d'extension. Parmi celles-ci se trouve des variantes qui ne sont pas des méthodes d'extensions, mais de simples accès à la fameuse instance statique. Mieux, les méthodes d'extension ne font en réalité qu'appeler ces méthodes simples. Ces dernières ne faisant, *in fine*, qu'utiliser l'instance statique `MvxServiceProvider.Instance` qu'on pourrait d'ailleurs utiliser directement en se passant totalement de l'intermédiaire des méthodes d'extension.

Néanmoins je ne conseille pas d'accéder à l'IoC de cette façon, en l'absence de documentation officielle complète et dans l'attente de la sortie de « vNext », les seules indications données par les exemples supposent de déclarer les interfaces puis d'accéder à l'IoC via les méthodes d'extension.

On peut se demander pourquoi une telle indirection existe alors qu'elle ne semble pas très utile… Attendons « le coup de ménage » et les évolutions de « vNext » avant de nous prononcer d'autant que beaucoup de services proposés comme tels aujourd'hui sont, dans « vNext », proposés sous la forme de plugins, une approche plus modulaire (mais respectant toujours l'IoC présenté ici).

vNext

MvvmCross « vNext » est en cours de développement, on peut en trouver une version intermédiaire sur Git Hub. Beaucoup de simplifications sont apportées au code, notamment dans la quantité de code empruntée à OpenNETCF.IoC qui passe à presque zéro en raison de l'utilisation du système de PCL, les nouvelles librairies de code portable de VS 2012.

L'ensemble de MvvmCross vNext, au moins la librairie de base, n'est plus qu'un seul projet, portable.

Les principes de l'IoC présentés ici, et la façon de MvvmCross propose cette fonction aux applications ne changent pas.

Toutefois le code sous-jacent est devenu plus clair et beaucoup plus simple.

La dernière version Git Hub date de trois ou quatre mois. Stuart continue bien sûr à travailler dessus et beaucoup d'autres choses auront bougé lorsqu'il fera le prochain commit…

C'est pourquoi cet article s'arrête là pour l'instant car il serait stupide d'entrer dans les détails de la version actuelle alors même que vNext arrive prochainement.

Conclusion

MvvmCross a beaucoup d'intérêt, c'est une librairie qui permet de régler en partie l'épineux problème de la portabilité des applications entre les différentes plateformes, qu'il s'agisse de celles de Microsoft ou celles d'Apple et Google.

Nous prenons le projet à ses débuts, la version prochaine en cours de finition sera déjà plus mature, même MvvmCross est d'ores et déjà utilisable.

Je vais attendre la publication de vNext pour écrire un article complet, comme je l'ai fait pour MVVM Light ou Jounce.

Encore un peu de patience donc !