Portable Class Library avec Visual Studio 2012

Avec la multiplication des supports (PC, tablettes, Smartphones, etc.), et des technologies de l'écosystème Microsoft (WPF, Silverlight, Windows Phone, Windows Azure, Xbox 360, etc.), la compatibilité des composants entre les technologies devient un sujet incontournable pour obtenir des développements de qualité. Il s'agit là de ne pas avoir à dupliquer des composants et donc du code pour chaque plateforme utilisée. Microsoft adresse ce sujet dans sa nouvelle version de Visual Studio 2012 avec la « Portable Class Library ».

Avertissement :
ce n'est pas du Windows 8, mais du pur Visual Studio 2012

Lorsque Silverlight est apparu, beaucoup de développeurs ont été confrontés à la même problématique : comment partager et mutualiser les développements communs entre des applications basées sur le framework .NET standard et celles basées sur le framework .NET Silverlight. Car, par défaut, il n'est pas possible de mélanger des librairies .NET standard et Silverlight et de les utiliser dans un même projet. Si on essaie cela, une erreur apparaît [Fig.1].

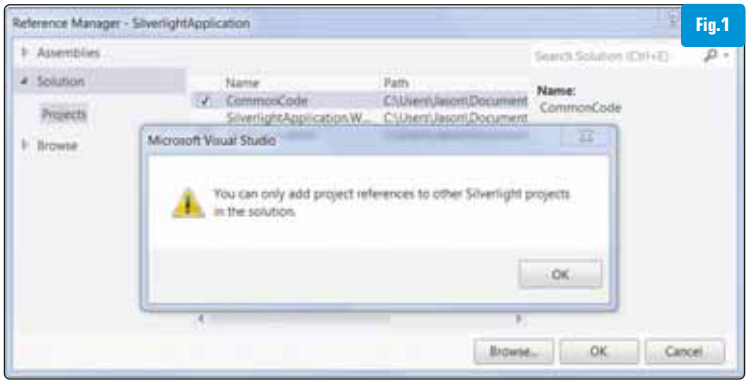

Fig.1

> Solution de contournement : Add as Link

A l'époque, la solution la plus répandue et aussi préconisée par Microsoft était plutôt une solution de contournement qu'une solution pérenne. Il s'agissait de dupliquer les librairies (au moins une par plateforme) sans dupliquer les fichiers sources : pour cela, il fallait rajouter les fichiers sources existants via la fonctionnalité « Add as Link » intégrée dans Visual Studio dans la boîte de dialogue « Add Existing Item » [Fig.2]. Après avoir compilé, on constate que le fichier n'a pas été copié vers le projet, mais à la place une référence y a été ajoutée.

Les fichiers sources ont donc été partagés, toutefois sans duplication, mais pas les librairies : il existe bien une version de librairie qui doit être compilée pour chaque plateforme visée.

Cette solution n'était bien évidemment pas satisfaisante, de plus la multiplication des plateformes dans les développements modernes rend cette approche laborieuse et inapplicable.

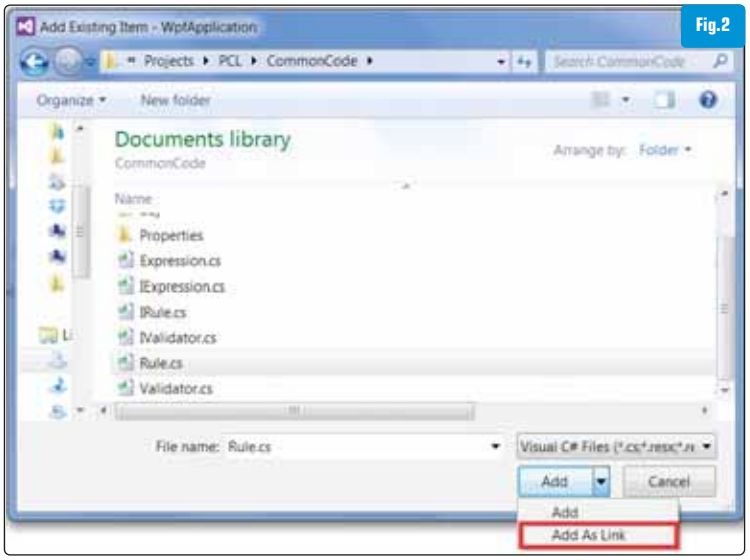

Fig.2

> Solution via NuGet package : Portable Library Tools

Microsoft, ne pouvant laisser cette situation en l'état, s'est attaqué à ce problème et a sorti une extension à Visual Studio 2010 SP1 permettant d'écrire des librairies utilisables par plusieurs plateformes via les « Portable Library Tools » actuellement en version 2 RC disponible ici : http://visualstudiogallery.msdn.microsoft.com/b 0e0b5e9-e138-410b-ad10-00cb3caf4981/.

Encore plus facile, ce package peut aussi être téléchargé et installé directement via NuGet depuis l' « Extension Manager » de Visual Studio 2010 [Fig.3]. Une fois ce package installé, il est possible de créer un nouveau projet de type « Portable Class Library ». Ce projet permet de générer des librairies pouvant être utilisées directement par plusieurs plateformes qui doivent toutefois être choisies au préalable. La première version de cette extension a été mise en ligne en janvier 2011 et mise à jour régulièrement pour pouvoir prendre en compte :

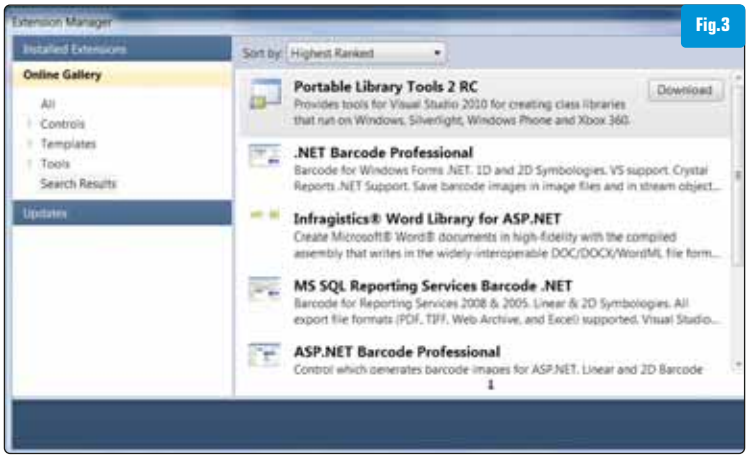

Fig.3

- Les nouvelles plateformes comme Modern UI (Windows 8), Windows Phone 7.5, Silverlight 5, etc.
- Les nouvelles versions du framework .NET comme la version .NET 4.5
- Ou bien encore pouvoir supporter de nouvelles fonctionnalités comme Code Contract

> Solution intégrée à Visual Studio 2012 : Portable Class Library (PCL)

Désormais, le type de projet « Portable Class Library (PCL) » est directement intégré à Visual Studio 2012 et c'est une bonne nouvelle, car les exigences en termes de qualité, maintenabilité et fonctionnalités se sont accrues.

La PCL peut s'avérer très utile pour partager des algorithmes (par exemple pour la validation des règles métier), des interfaces ou des objets de données entre plusieurs applications. Comme déjà évoqué, ces applications peuvent être basées sur des technologies différentes. Un autre cas d'utilisation intéressant est le stockage du View-Model dans une librairie PCL quand on souhaite implémenter le pattern MVVM [Fig.4]. Le View-Model peut alors être partagé dans toutes vos applications (Windows, Web, Mobile, etc.).

L'utilisation de la PCL permet donc la consistance et l'encapsulation du code indépendamment de l'endroit où il sera utilisé. Ceci réduit les coûts de maintenance et augmente la productivité de l'équipe.

Voici un tableau avec les fonctionnalités supportées par technologie en utilisant la PCL :

Fonctionnalité	.NET Framework	Modern UI style	Silverlight	Windows Phone	Xbox 360				
	4	4.0.3	4.5		4	5	7	7.5	
Core BCL	X	X	X	X	X	X	X	X	X
Core XML	X	X	X	X	X	X	X	X	X
LINQ	X	X	X	X	X	X	X	X	
IQueryable	X	X	X	X	X	X		X	
Dynamic Keyword			X	X	X	X			
Core WCF	X	X	X	X	X	X	X	X	
Core Networking	X	X	X	X	X	X	X	X	
MEF	X	X	X	X	X	X			
Data Contract Serialization	X	X	X	X	X	X	X	X	
XML Serialization	X	X	X	X	X	X	X	X	
Json Serialization	X	X	X	X	X	X	X	X	
View models	X	X	X	X	X	X			
Data annotations	X	X	X	X	X				
XLINQ	X	X	X	X	X	X	X	X	
System.Numerics			X	X	X	X	X	X	

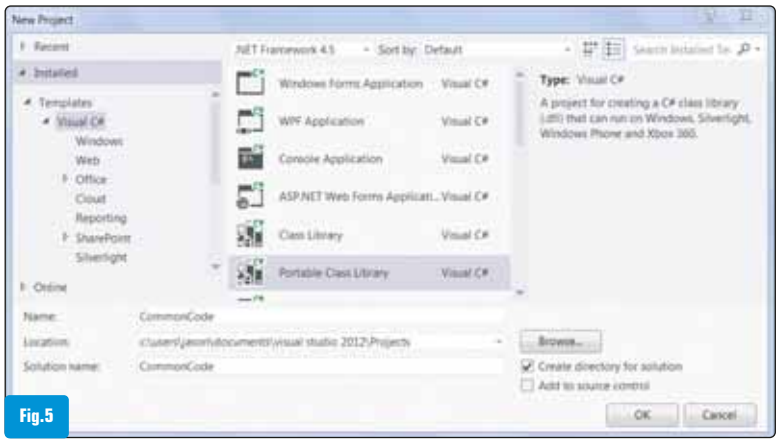

Fig.5

Il faut noter que la PCL permet seulement l'utilisation d'un ensemble de fonctionnalités communes aux différentes technologies : on est donc contraint de cibler le plus petit dénominateur commun. Ceci est compréhensible, puisque toutes les fonctionnalités ne sont pas forcément disponibles dans chaque technologie. Dans la suite de cet article, nous allons tenter de démystifier la PCL par le biais d'un exemple simple qui couvrira un maximum de plateformes.

Création du projet :

Intéressons-nous maintenant à l'utilisation de la PCL avec Visual Studio 2012. La première chose à faire est de créer un nouveau projet de type PCL [Fig.5].

Ensuite, vous devez sélectionner les frameworks cibles avec lesquels votre librairie PCL doit être compatible. Cette étape est très importante, car plus vous allez en ajouter, moins vous aurez de fonctionnalités accessibles : si vous souhaitez développer un composant qui doit être utilisé par des projets qui utilisent les framework .NET 3.5 et .NET 4.5 par exemple, vous n'aurez pas accès aux fonctionnalités liées à « Code Contract ».

Notez qu'une fois compilée, la librairie PCL qui en résulte peut être référencée sans aucune modification ni configuration.

Une autre information intéressante est que vous avez la possibilité de modifier (ajouter et supprimer) la liste des technologies supportées après la génération du projet. Une bonne pratique consiste donc à inclure, dans un premier temps, uniquement les frameworks nécessaires et à ajouter les autres au fur et à mesure que les besoins se présentent.

Targeting Packs :

Vous avez également la possibilité d'étendre le nombre de frameworks supportés en téléchargeant d'autres packs de frameworks pour Visual Studio 2012. Ils peuvent contenir des mises à jour pour Visual Studio et doivent être installés avec les runtimes correspondants.

Voici les technologies supportées et les packs actuellement disponibles au téléchargement (http://msdn.microsoft.com/en-us/hh487283.aspx) :

- .NET Framework 4.5 (inclus dans Visual Studio 2012)
- .NET Framework 4.0.3, 4.0.2, 4.0.1
- .NET Framework 4 (inclus dans Visual Studio 2012)
- .NET Framework 2.0/3.0/3.5 SP1
- Windows Azure (via Windows Azure SDK)
- Xbox 360 (via XNA Game Studio 4.0)
- Silverlight 5 (inclus dans Visual Studio 2012)
- Windows Phone 7.5 (inclus dans Visual Studio 2012)
- Modern UI style / Windows 8 (inclus dans Visual Studio 2012)

Implémentation :

Prenons un exemple concret dans Visual Studio 2012 pour implémenter des fonctionnalités d'un validateur de règles métiers basées sur des expressions.

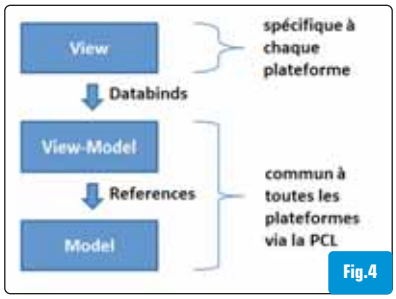

Fig.4

La classe « Expression » implémente l'interface « IExpression » qui contient la méthode « Execute() ».

Cette méthode retourne « true » si l'expres-

sion a abouti sans erreur, sinon elle retourne « false ». Dans cet exemple simple, elle retournera toujours « true ».

```
public interface IExpression
{
 bool Execute();
}
public class Expression : IExpression
{
 public bool Execute()
 {
   // Do something
   //return true if sucessful
   //return false if not sucessful
   return true;
 }
}
```

La classe « Rule » implémente l'interface « IRule » qui contient deux propriétés : « RuleId » du type usigned integer et « ExpressionList » du type IExpression (défini ci-dessus). Cette classe a pour objectif de stocker toutes les expressions pour une règle métier.

```
public interface IRule
{
 uint RuleId { get; }
 IList<IExpression> ExpressionList { get; }
}
public class Rule : IRule
{
 private uint _ruleId;
 private IList<IExpression> _expressionList = new List<IExpression>();

 public Rule(uint ruleId)
 {
   _ruleId = ruleId;
 }

 public uint RuleId
 {
   get { return _ruleId; }
 }

 public IList<IExpression> ExpressionList
 {
   get
   {
     return _expressionList;
   }
 }
}
```

La classe « Validator » implémente l'interface « IValidator » qui contient la méthode « ValidateRules(…) ». Cette méthode a comme paramètre d'entrée une liste de règles métiers du type « IList<IRule> » (défini ci-dessus). Elle retourne « true » si la validation de toutes les expression a abouti sans erreur, sinon elle retourne « false ».

```
public interface IValidator
{
 bool ValidateRules(IList<IRule> ruleList);
}
public class Validator : IValidator
{
 public bool ValidateRules(IList<IRule> ruleList)
 {
   bool result = true;

   foreach (var rule in ruleList)
   {
     foreach (var expression in rule.ExpressionList)
     {
       result &= expression.Execute();
     }
   }

   return result;
 }
}
```

Une fois le code compilé, vous pouvez utiliser cette librairie depuis n'importe quel framework sélectionné au début. Voici un exemple d'implémentation du validateur de règles métiers :

```
IExpression expression1 = new CommonCode.Expression();
IExpression expression2 = new CommonCode.Expression();

IRule rule = new Rule(1);
rule.ExpressionList.Add(expression1);
rule.ExpressionList.Add(expression2);

var ruleList = new List<IRule>();
ruleList.Add(rule);

IValidator validator = new Validator();
bool isValid = validator.ValidateRules(ruleList);
```

Cet exemple simple nous a permis de voir comment partager des interfaces, des classes et du code métier dans une librairie PCL unique.

Conclusion

Microsoft fait des progrès dans l'unification du développement multiplateforme en fournissant un moyen simple de partager du code via la Portable Class Library (PCL). De plus, même Mono 2.12 (http://www.mono-project.com), Mono for Android (http://xamarin.com/monoforandroid) et MonoTouch pour IPhone (http://xamarin.com/monotouch) supportent maintenant la PCL ce qui permet un partage de code encore plus large.

Jason **De Oliveira**
Practice Manager & Solutions Architect | MVP C#
.Net Rangers by Sogeti
Blog: http://jasondeoliveira.com

Fathi **Bellahcene**
Software Architect MVP C#.Net Rangers by Sogeti
Blog : http://blogs.codes-sources.com/fathi