www.e-naxos.com

Formation – Audit – Conseil – Développement
XAML (Windows Store, WPF, Silverlight, Windows Phone), C#
Cross-plateforme Windows / Android / iOS
UX Design

ALL DOT.BLOG
TOME 1

C#

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

Olivier Dahan
odahan@gmail.com

Table des matières

Les nouveautés de C# 3.0

C# est un langage en perpétuel mouvement. Changeant sans rien remettre en cause, s'améliorant au-delà de ce que bon nombre de développeurs pouvait même imaginer. Cette métamorphose pousse C# vers un langage fonctionnel supportant un style de plus en plus déclaratif. Bien entendu les avancées du langage font intégralement partie du bouillonnement général d'idées qui est celui des équipes Microsoft depuis ce que j'appellerais « l'ère .NET ».

En effet, depuis le lancement de la plateforme .NET (le Framework et C#), ce sont régulièrement des idées plus innovantes les unes que les autres que nous propose Microsoft. Qu'il s'agisse de WPF, WCF, WF, de Silverlight, ou bien de l'Entity Framework et donc aussi de LINQ (et je raccourci volontairement la liste à laquelle on pourrait ajouter Microsoft Ajax, Astoria, …), chacune de ces évolutions pourrait à elle seule passer pour une révolution géniale chez un autre éditeur… Ne nous lassons pas par habitude ou tellement le rythme des nouveautés est soutenu et gardons intact notre capacité d'émerveillement ! Le Framework 3.5 regorge d'idées nouvelles, C# 3.0 n'est finalement qu'une partie de cette immense galaxie.

Je reparlerai d'ailleurs de l'Entity Framework dans un autre article et bien sûr de LINQ (même si ce dernier n'est pas uniquement lié aux données de type SQL), mais avant d'aborder ces avancées il est essentiel de bien comprendre les nouveautés de C# 3.0 tellement les évolutions du Framework sont intimement liées à celles du langage.

Inférence des types locaux

Sous ce terme un peu barbare (*local type inference* en anglais) se cache une fonction puissante, celle du mot clé `var`.

`Var` existe sous d'autres langages (JavaScript, Delphi) mais avec un comportement bien différent. Sous Delphi il s'agit du seul moyen de déclarer une variable, entre l'entête de méthode et son corps, sous JavaScript la variable déclarée par `var` n'est pas typée et peut contenir une chaîne de caractères comme un entier par exemple.

Sous C#, `var` s'utilise partout où une variable peut être déclarée (continuité du style) et s'emploie à la place du type car celui-ci sera deviné automatiquement en fonction de celui de l'expression (inférence du type). Une fois le type attribué il ne sera pas modifiable et la variable se comporte exactement comme si elle avait été déclarée de façon traditionnelle. Il ne faut d'ailleurs par confondre ce fonctionnement avec les *variants* qu'on retrouve sous Delphi ou d'autres langages comme Visual Basic. Les variants possèdent un type dynamique fixé à l'exécution alors qu'une déclaration `var` de C# n'est qu'un raccourci qui donnera naissance *à la compilation* à une déclaration de type tout à fait classique.

Un exemple nous fera comprendre l'intérêt et la syntaxe de ce nouveau mot clé…

Utilité

D'abord parlons de l'utilité, on retient mieux ce dont on comprend le sens et les avantages. Dans les langages objet à typage fort toute variable doit se déclarer avec son type. Mais comme toute variable objet doit être instanciée avant d'être utilisée on est très souvent obligé d'appeler le constructeur à la suite de la déclaration de la variable. De fait, on obtient une écriture du type de ceci :

```
MonTypeObjet unObjet = new MonTypeObjet() ;
```

Écrire deux fois « `MonTypeObjet` » dans une ligne si petite de code semble à la longue très contraignant. Certes la notation est rigoureuse mais bien peu efficace.

Regardons la même déclaration utilisant `var` :

```
var unObject = new MonTypeObjet() ;
```

Du point de vu fonctionnel la variable « `unObjet` » sera identique et bien entendu toujours fortement typée. Il s'agit donc d'un artifice de notation qui rend le code plus lisible. D'ailleurs il suffit de regarder le code IL généré à la compilation, il ressemble exactement à une déclaration de variable « classique », c'est-à-dire que le code IL contient le type inféré comme si celui-ci avait été saisi directement dans le code source.

Exemples

Une fois l'intérêt mis en évidence, voici d'autres exemples d'utilisation :

```
var a = 3 ;                  // a sera de type int
var b = "Salut !" ;          // b sera de type string
var q = 52.8 ;                     // q sera un double
var z = q / a ;                    // z aussi
var m = b.Length() ;         // m sera un int
decimal d ; var f = d ;      // f sera un decimal
var o = default(string) ;    // o sera un string
var t = null ;               // Interdit !
                                   // le type ne peut pas être inféré…
```

On notera le mot clé `default(<type>)` qui retourne la valeur nulle par défaut pour le type considéré.

Une porte sur les types anonymes

Arrivé à ce stade on pourrait se dire que, finalement, var n'a été introduit que pour pallier la paresse des développeurs… Même si cela n'est pas faux (mais un bon développeur doit être paresseux !), ce n'est pas que cela, `var` est aussi le seul moyen de déclarer des variables avec un **type anonyme**, autre nouveauté que nous verrons plus loin.

A noter, `var` ne peut être utilisé qu'à l'intérieur d'une portée locale. Cela signifie qu'on peut déclarer une variable locale avec `var` mais pas un paramètre de méthode ou un type de retour de méthode par exemple.

Les expressions Lambda

Encore une terminologie savante qui peut effrayer certaines personnes. En réalité, et nous allons le voir, il n'y a rien d'inquiétant dans cette nouvelle syntaxe qui ne fait que poursuivre le chemin tracé par C# 2.0 et ses méthodes anonymes en simplifiant et en généralisant à l'extrême l'utilisation et la notation de ces dernières.

Rappel sur les méthodes anonymes

Pour rappel, une méthode anonyme n'est qu'une méthode… sans nom, c'est-à-dire un morceau de code représentant le corps d'une méthode qu'on peut placer là où un pointeur de code (un *delegate*) est attendu, et ce, dans le respect de la déclaration du delegate (C# reste un langage très fortement typé malgré les apparences syntaxiques trompeuses de certaines de ses évolutions !).

Tout d'abord voyons un exemple de code utilisant une méthode anonyme à la façon de C# 2.0 :

```
public class DemoDelegate
        {
            delegate T Func<T>(T a, T b);

            static T Agreger<T>(List<T> l, Func<T> f)
            {
                T result = default(T);
                bool premierPassage = true;
                foreach (T value in l)
                {
                    if (premierPassage)
                    {
                        result = value;
                        premierPassage = false;
                    }
                    else
                    {
                        result = f(result, value);
                    }
                }
                return result;
            }

            public static void LancerDemo()
            {
                int somme;
                List<int> lesEntiers = new List<int> {1,2,3,4,5,6,7,8,9};
                somme = DemoDelegate.Agreger(
                        lesEntiers,
                        delegate(int a, int b) { return a + b; }
                    );
                Console.WriteLine("La somme = {0}", somme);
            }
        }

        static void Main(string[] args)
        {
            DemoDelegate.LancerDemo();
        }
```

Dans cet exemple nous définissons une classe `DemoDelegate` qui expose une méthode générique `Agreger` permettant d'appliquer une fonction sur une liste dont les éléments

peuvent être de tout type. Liste et fonction à appliquer étant passées en paramètre lors de l'appel de la méthode.

La méthode (statique) `LancerDemo` créée une liste d'entiers ainsi qu'une variable `somme`. Ensuite elle assigne à cette dernière le résultat de l'appel à `DemoDelegate.Agreger` en lui passant en paramètre la liste d'entiers ainsi qu'une méthode anonyme (en surligné gris dans l'exemple ci-dessus) définissant le traitement à effectuer.

On notera que cette méthode anonyme répond au prototype du delegate `Func` aussi déclaré dans `DemoDelegate`. Cette déclaration permet d'assurer un typage fort du code passé. En place et lieu d'une méthode anonyme de C# 2.0 nous aurions été obligés, sous C# 1.0, de créer une méthode de calcul portant un nom et de passer ce dernier en argument de la méthode `Agreger` via un delegate.

On remarque ainsi que le passage de C# 1.0 à C# 2.0 nous a permis une économie syntaxique rendant le code plus léger, plus élégant et plus facilement réutilisable, et ce, grâce aux méthodes anonymes et aux génériques.

Peut-on aller plus loin dans le même esprit ?

Aller plus loin grâce aux expressions Lambda

La réponse de C# 3.0 est claire : oui, et cela s'appelle les expressions Lambda.

Regardons comment à l'aide des expressions Lambda nous pouvons réécrire le code de la méthode `LancerDemo` :

```
public static void LancerDemoCS3()
            {
                int somme;
                List<int> lesEntiers = new List<int> {1,2,3,4,5,6,7,8,9};
                somme = DemoDelegate.Agreger(lesEntiers,
                                    (int x, int y) => { return x + y; });
                Console.WriteLine("La somme = {0}", somme);
            }
```

Le code surligné en gris contient l'expression Lambda.

Comme on le voit il n'y a rien de bien compliqué, nous n'avons fait que supprimer le mot clé `delegate` et avons introduit le symbole `=>` entre la déclaration des paramètres de la méthode anonyme et l'écriture de son code.

On peut lire l'expression ci-dessus de la façon suivante : « *étant donné les paramètres x et y de type entier, retourner la somme de x et y.* »

La simplification peut aller plus loin car les expressions lambda supportent le *typage implicite* des paramètres. Ainsi l'expression peut être simplifiée comme suit :

```
(x, y) => { return x + y; }
```

Le type des paramètres `x` et `y` peut être omis puisque C# sait que les paramètres doivent correspondre au prototype `Func`. Certes ce delegate est totalement déclaré avec des types génériques… Et c'est en réalité par l'appel de `Agreger` et grâce au premier paramètre (la liste

de valeurs) que C# 3.0 peut inférer les types. Il n'y a donc aucune magie et surtout, tout reste très fortement typé sans faire aucune concession.

L'inférence des types est effectuée à la compilation, bien entendu, et non à l'exécution.

Comment prononcer le nouveau symbole ?

Il n'y a semble-t-il pas de nom particulier pour le signe `=>` des expressions Lambda. On peut proposer de le lire comme « Tel Que » lorsque l'expression est un prédicat ou « Deviens » lorsqu'il s'agit d'une projection.

> *Pour rappel, un prédicat est une expression booléenne généralement utilisée pour créer un filtre et une projection est une expression retournant un type différent de son unique paramètre.*

Deux écritures possibles du corps

La syntaxe d'une expression Lambda supporte deux façons de définir le corps de la méthode anonyme, soit avec des *brackets* comme nous l'avons vu dans l'exemple ci-dessus, soit sous la forme d'une simple instruction `return` en omettant ce mot-clé. Ainsi l'expression de notre exemple deviendra encore plus simplement :

```
somme = DemoDelegate.Agreger(lesEntiers, (x, y) => x + y );
```

Nous avons supprimé les brackets et même le point-virgule final puisque nous ne sommes plus dans un bloc de code mais plutôt dans l'écriture d'une simple instruction et que la syntaxe à cet endroit n'autorise pas de point-virgule après l'instruction (l'expression Lambda occupe en effet la place du second paramètre de `Agreger`, les paramètres sont seulement suivis par des virgules sauf le dernier, ce qui est le cas de l'expression ici).

Simplifions encore

Prenons maintenant un autre exemple réclamant un delegate ne possédant qu'un seul paramètre :

```
public delegate T Func2<T>(T x);

public static T Agreger2<T>(List<T> l, Func2<T> f)
{
  T result = default(T);
  foreach (T value in l) result = f(value);
  return result;
}

public static void LancerDemoCS3v4()
{
  Single somme=0f;
  List<Single> lesSimples =
        new List<Single> { 1.5f, 2.6f, 3.7f, 4.8f, 5.9f,
                           6.0f, 7.1f, 8.2f, 9.3f };
  somme = DemoDelegate.Agreger2(lesSimples, (x) => somme += x);
  Console.WriteLine("La somme = {0}", somme);
 }
```

Dans la version ci-dessus nous avons créé un nouveau delegate qui ne prend qu'un seul paramètre (`Func2`). La méthode `Agreger2` a été modifiée pour refléter cette modification, son code est devenu d'ailleurs plus simple.

Mais ce qui nous intéresse est l'utilisation de la méthode Lambda (sur fond gris).

En dehors du fait qu'elle n'utilise plus qu'un seul paramètre, conformément au nouveau delegate, on s'aperçoit qu'elle peut se permettre d'utiliser la variable locale `somme` à l'intérieur même de sa définition. En effet, `somme` est une locale de la méthode contenant l'expression et sa portée ainsi que sa durée de vie sont étendues à l'instance de la méthode anonyme définit par l'expression Lambda.

Simplifions encore… Lorsqu'il n'y a qu'un seul paramètre comme dans le dernier exemple on peut omettre les parenthèses qui l'entourent. Ainsi, l'expression pourra directement s'écrire :

```
x => somme += x
```

Rappel important

Les expressions lambda, tout comme les méthodes anonymes dont elles sont un prolongement et une simplification syntaxique, ne servent à saisir que des petits bouts de code et non des pages entières ! On les utilise principalement pour créer des filtres ou autres fonctions de ce type ne réclamant que quelques instructions au maximum. Si le corps d'une expression Lambda, tout comme une méthode anonyme C# 2.0, dépasse cette limite il faut alors déclarer une méthode et utiliser un delegate « classique ». Que ceux qui seraient tentés d'écrire trois pages de code dans une expression Lambda soient prévenus, cela est très fortement déconseillé !

Un autre exemple

Cette mise au point indispensable effectuée, voyons comment une expression Lambda peut simplifier grandement un code de type filtrage de liste (donc utiliser l'expression en tant que prédicat).

```
public class DemoPredicat
{
  public static void AfficheListe<T>(T[] items, Func<T, bool> leFiltre)
  {
    foreach (T item in items) if (leFiltre(item)) Console.WriteLine(item);
  }

  public static void lancerDemo()
  {
   string[] villes = { "Paris", "Berlin", "Londres", "New-york",
                       "Barcelone", "Milan" };

   Console.WriteLine("Les villes sans 'e' dans leur nom sont:");
   AfficheListe(villes, s => !s.Contains('e'));

   Console.WriteLine("Les villes ayant 'i' dans leur nom sont:");
   AfficheListe(villes, s => s.Contains('i'));
  }
}
```

Dans le code ci-dessus que remarque-t-on ?

D'abords nous n'avons pas déclaré de delegate. Nous avons utilisé une déclaration existante dans le Framework. Ce genre de prototype étant très courant, notamment pour les prédicats, le Framework le contient déjà.

Ensuite nous voyons très clairement à quel point les expressions Lambda rendent le code concis et clair. La liste des villes est filtrée et affichée deux fois, les villes ne possédant pas de « e » dans leur nom puis celles contenant un « i » dans ce même nom. D'ailleurs la même fonction `AfficherListe` pourrait être utilisée sans modification pour afficher une liste d'entiers ou de dates filtrés puisqu'elle n'utilise que des types génériques. Quant à l'appel de cette méthode, il contient directement le filtrage à effectuer, de façon simple, clair et lisible, sans artifice ni delegate superflu !

Les expressions Lambda, c'est exactement ça : plus de simplicité et d'élégance pour un code plus lisible, plus flexible et plus puissant.

On notera que le Framework définit l'ensemble suivant de delegates utilisables de la même façon que `Func` dans notre exemple qu'on retrouve en première entrée de la liste :

- `public delegate T Func< T >();`
- `public delegate T Func< A0, T >( A0 arg0 );`
- `public delegate T Func<A0, A1, T> ( A0 arg0, A1 arg1 );`
- `public delegate T Func<A0, A1, A2, T >( A0 arg0, A1 arg1, A2 arg2 );`
- `public delegate T Func<A0, A1, A3, T> ( A0 arg0, A1 arg1, A2 arg2, A3 arg3 );`

Il n'y a bien entendu aucune obligation d'utiliser ces types définis dans `System.Linq` (ajouté automatiquement aux projets sous VS 2008). Vous pouvez utiliser vos propres types. Il n'y a qu'un seul cas dans lequel il faut respecter les définitions de delagate présentées ci-dessus : lorsqu'on veut transformer une expression en **arbre d'expression**.

Les arbres d'expression

Il faut bien prendre conscience que ces ajouts au langage ont été faits certes pour leur puissance intrinsèque mais aussi et surtout pour faciliter l'implémentation de Linq… Or Linq, dont nous parlerons en détail dans un prochain article, impose certaines exigences comme le fait de pouvoir transformer une expression en un arbre facilement navigable. Les arbres d'expression sont analysés à l'exécution et peuvent même être créés à ce moment. Cela est utilisable de plusieurs façons, Linq, lui, s'en sert notamment pour transformer les requêtes Linq C# en syntaxe SQL (Linq to ADO.NET) conforme à la base cible. Cette dernière dépendant de la connexion et de la base cible, de son langage, des champs, Linq a besoin d'interpréter les arbres à ce moment et non à la compilation.

La différence principale entre une expression Lambda comme celles que nous avons vues dans cet article et un arbre expression se situe uniquement dans la représentation de la méthode anonyme. Une expression Lambda binaire est compilée et se présente sous la forme de code IL, alors qu'un arbre expression est une représentation mémoire dynamique (modifiable notamment) donc une représentation runtime.

Seules les expressions Lambda possédant un corps peuvent être transformées en arbre expression. Nous avons vu dans cet article que dans des cas très simples les expressions pouvaient s'écrire comme une instruction, ce sont les expressions sous cette forme qui sont exclue de la transformation en arbre. Nous aborderons les arbres d'expression dans un prochain article, le sujet réclamant de s'y attarder plus longuement.

Les méthodes d'extension

On peut les voir comme une émanation de la design pattern *Decorator*. On les trouvait déjà sous d'autres formes dans d'autres langage, par exemple Borland les a implémentées dans Delphi 8 pour .NET principalement pour ajouter artificiellement les méthodes de TObject à System.Object de .NET et faire passer le second pour le premier, fonctionnellement, aux yeux de la VCL.NET.

Il s'agit donc de pouvoir ajouter des méthodes à une classe sans modifier la dite classe… Magique ? Oui et non. Cela peut paraître très rusé mais risque vite d'être ingérable si on imagine un code utilisant en plus des interfaces et de l'héritage cette technique qui fait sortir des méthodes du chapeau du magicien et non des classes elles-mêmes… Vous voilà prévenus, cela peut être utile, mais c'est à utiliser avec une grande modération !

Microsoft a implémenté cette possibilité dans C# 3.0 pour simplifier la syntaxe de Linq, la rendre plus lisible et plus concise.

C# 3.0 fait en sorte que les méthodes accrochées à une classe ne puissent accéder qu'à ces membres publics. De fait le procédé ne permet en aucune sorte de violer le principe d'encapsulation des objets. Cela a l'avantage d'être propre et d'éviter certaines dérives.

Les class helpers doivent être définis dans des classes statiques avec des méthodes statiques.

Je n'ai hélas trouvé aucune utilisation simple et pertinente des class helpers, rien qui ne puisse être réglé bien plus proprement par l'héritage ou le support d'une interface. Vous l'avez compris, je n'aime pas trop cette « amélioration » du langage. Mais personne ne m'oblige à m'en servir non plus, alors tout va bien !

C# étant un langage très puissant il ne faut pas non plus regarder sa syntaxe par le très réducteur gros bout de la lorgnette… Comme simple possibilité, les class helpers ne sont pas indispensables, toutefois lorsqu'on associe class helpers et généricité, on peut arriver à trouver des utilisations intelligentes et élégantes, c'est d'ailleurs dans un tel esprit que Linq s'en sert. A vous de trouver des utilisations au moins aussi pertinentes.

Sans trop entrer dans de tels détails (je reste pour l'instant, et faute de recul, réservé sur le sujet des class helpers au sein d'un code bien écrit et maintenable) je vous livre un exemple pour que vous puissiez en comprendre le mécanisme :

```
public struct Article
{
  public int Code;
  public string Désignation;
```

```
  public override string ToString()
  { return Code + ", " + Désignation; }
  public Article(int code, string designation)
  { Code = code; Désignation = designation; }
}

public struct Client
{
  public int Code;
  public string Société;
  public override string ToString()
  { return Code + ", " + Société; }
  public Client(int code, string société)
  { Code = code; Société = société; }
}

public static class DemoHelpers
{
  public static void LanceDemo()
  {
    Article a = new Article(101, "Zune 20 Go");
    Client c = new Client(5800, "E-Naxos");
    a.Affiche(); // appel « magique » à Affiche
    c.Affiche();
  }
}

// déclaré non imbriqué dans une autre classe
public static class Afficheur
{
  public static void Affiche(this object o)
  { Console.WriteLine(o.ToString()); }
}
```

Dans l'exemple ci-dessus deux structures sont déclarées, `Article` et `Client`, chacune ayant ses spécificités et ne partageant rien en commun. Ailleurs dans le code est déclarée la classe statique `Afficheur` qui possède la méthode `Affiche` (statique aussi). Les paramètres de `Affiche`, et l'utilisation de `this`, en font automatiquement un class helper.

C'est ce qui permet d'appeler `Affiche` depuis des instances de `Article` ou de `Client`. Si la déclaration de la méthode `Affiche` avait indiqué `Article` à la place de `object`, seule la classe `Article` aurait pu utiliser `Affiche` qui ne serait donc plus visible depuis les instances de `Client`.

Les expressions d'initialisation des objets

Un code source travaille sur des variables qu'il faut déclarer et initialiser. La syntaxe dédiée à ces opérations élémentaires est importante puisque, revenant très souvent sous les doigts du développeur, toute lourdeur sera ressentie comme pénible avec le temps.

Les initialisations rapides de C# 1.x

C# 1.0 proposait déjà quelques facilités syntaxiques, pour rappel :

```
      string s = "Bonjour" ;
      single x = 10.0f ;
      Synthé synthé = new Synthé("Prophet",5,"Sequential Circuit") ;
```

Lorsqu'on instancie un type par valeur ou par référence on peut appeler l'un de ses constructeurs permettant, en une seule opération, d'initialiser les principaux états de l'objet. C'est le cas du dernier exemple ci-dessus.

Si cette approche est particulièrement efficace elle oblige à prévoir (et à coder) de nombreuses surcharges du constructeur dans chaque classe. On remarque avec *Intellisense* que Microsoft a utilisé cette façon de faire en de nombreuses occasions ce qui facilite grandement le codage. Certaines classes possède jusqu'à dix ou vingt constructeurs différents… Autant de code à écrire et à maintenir malgré tout.

Les initialisations avec la syntaxe de base

Si on en revient à la syntaxe de base pour créer et initialiser un objet et que nous reprenons notre dernier exemple, le code s'écrirait comme suit :

```
        Synthé synthé ;
        synthé = new Synthé() ;
        synthé.Modèle= "Prophet" ;
        synthé.Version = 5 ;
        synthé.Fabriquant = "Sequential Circuit" ;
```

On notera la lourdeur du style, et l'augmentation du nombre de bogues potentiels comparativement à la syntaxe raccourcie vue plus haut.

C# 3.0, le meilleur des deux mondes

C# 3.0 apporte le meilleur des deux syntaxes, celle de l'appel à un initialiseur, compacte, et celle plus classique de l'accès à chaque membre, plus complète et ne réclamant pas l'écriture d'une série d'initialiseurs spécialisés pour chaque cas de figure.

Reprenons l'exemple de notre bon vieux synthétiseur…

```
Synthé synthé = new Synthé
 {Modèle="Prophet",Version=5,Fabriquant="Sequential Circuit",
  AnnéeDeSortie = 1978 } ;
```

On remarque immédiatement les avantages, par exemple nous avons pu initialiser l'année de sortie alors qu'elle n'a pas été prévue dans le constructeur / initialiseur de cette classe. De fait aucun initialiseur n'a été codé dans la classe d'ailleurs. On remarque ensuite qu'on appelle le constructeur par défaut en omettant les parenthèses, il est directement suivi du bloc d'initialisation entre brackets.

La technique est séduisante et permet de gagner en concision, toutefois elle n'est pas parfaite. Il ne faut donc pas voir cette solution comme la fin des initialiseurs spécifiques mais plutôt comme un complément pratique, parfois tout à fait suffisant, parfois ne pouvant répondre à tous les besoins d'un vrai initialiseur.

Par exemple seules les propriétés publiques sont accessibles, il est donc impossible par cette syntaxe d'initialiser un état interne à partir des paramètres d'initialisation, ce qu'un constructeur permet de faire. Enfin, puisqu'on accède aux propriétés publiques et que très souvent les modifications de celles-ci déclenchent des comportements spécifiques (mise à jour de l'affichage par exemple), la nouvelle syntaxe peut s'avérer pénalisante là où un constructeur est capable de changer tous les états internes avant d'accomplir les actions ad hoc.

Il faut comprendre en effet que la nouvelle syntaxe des initialiseurs d'objets n'est qu'un artifice syntaxique, il n'y a pas eu de changement du langage lui-même et le code IL produit par la nouvelle syntaxe est rigoureusement identique à la syntaxe de base pour créer et initialiser un objet (déclaration de la variable, appel du constructeur puis initialisation de chaque propriété, voir l'exemple plus haut).

L'appel aux constructeurs

La nouvelle syntaxe est en revanche assez souple pour permettre d'appeler un autre constructeur que celui par défaut, et dans un tel cas elle procure bien un avantage stylistique. Toujours en partant du même exemple :

```
        Synthé synthé = new Synthé("Prophet",5,"Sequential Circuit")
                                    { AnnéeDeSortie = 1978 } ;
```

Ici nous supposons qu'il existe un constructeur prenant en compte le modèle, la version et le fabriquant. Toutefois il n'en existe aucune version permettant aussi d'initialiser l'année de sortie, comme cette propriété publique existe il est possible de mixer l'appel au constructeur le plus proche de notre besoin et d'ajouter à la suite l'initialisation du ou des champs qui nous intéressent (`AnnéeDeSortie` ici).

Les expressions d'initialisation des objets ne sont pas un ajout décisif au langage mais habilement utilisées elles complètent la syntaxe existante pour produire un code toujours plus clair, lisible et plus facilement maintenable. On notera que C# fait toujours les choses avec beaucoup de précautions puisque l'appel à une expression d'initialisation créée une variable cachée en mémoire jusqu'à ce que toutes les initialisations soient terminées et uniquement à ce moment là la variable est renseignée (la variable `synthé` dans notre exemple). Ainsi on retrouve les avantages d'un constructeur, tout est passé ou rien n'est passé, mais à aucun moment un morceau de code ne pourra accéder à une variable « à démi » initialisée.

Un peu de magie…

Nous avons défini la classe `Synthé` pour l'exemple plus haut, nous allons la réutiliser pour créer une classe `MiniStudio` qui définit un synthétiseur principal et un autre, secondaire :

```
public class MiniStudio
{
  private Synthé synthéPrincipal = new Synthé();
  private Synthé synthéSecondaire = new Synthé();
  public Synthé SynthéPrincipal { get { return synthéPrincipal; } }
  public Synthé SynthéSecondaire { get { return synthéSecondaire; } }
}
```

Comme on le voit ci-dessus, les deux synthés sont définis comme des champs privés de la classe auxquels on accède via des propriétés en lecture seule.

Nous pouvons dès lors instancier et initialiser un `MiniStudio` de la façon suivante en utilisant la nouvelle syntaxe :

```
var studio = new MiniStudio
             {
              SynthéPrincipal = { Modèle = "Prophet", Version = 5 },
```

```
                SynthéSecondaire = { Modèle = "Wave", Version = 2,
                                     Fabriquant="PPG", AnnéeDeSortie = 1981 }
            };
```

I l est donc tout à fait possible d'une part d'utiliser la nouvelle syntaxe pour initialiser des instances imbriquées (les instances de `Synthé` dans l'instance de `MiniStudio`), mais surtout on peut voir que les propriétés `SynthéPrincipal` et `SynthéSecondaire` sont accessible en écriture alors qu'elles sont en lecture seule ! Violation de l'encapsulation, magie noire ?

Bien sur que non. Et heureusement… En réalité nous n'écrivons pas une nouvelle valeur pour les pointeurs d'objet que sont les propriétés `SynthéPrincipal` et `SynthéSecondaire`, nous ne faisons qu'accéder aux instances créées par la classe `MiniStudio` et aux propriétés publiques de ces instances…

Toutefois, si nous avons fait précéder les brackets par `new` le compilateur aurait rejeté la syntaxe puisque ici il n'est pas possible de passer une nouvelle instance aux synthés (les propriétés sont bien en lecture seule).

De fait la ligne suivante serait rejetée à la compilation :

```
var studio = new MiniStudio
             {
              SynthéPrincipal = new { Modèle = "Prophet", Version = 5 },
              SynthéSecondaire = { Modèle = "Wave", Version = 2,
                                   Fabriquant="PPG", AnnéeDeSortie = 1981 }
             };
```

L'erreur rapportée est *"Error 1, Property or indexer 'SynthéPrincipal' cannot be assigned to -- it is read only"*

La nouvelle syntaxe est utilisable aussi pour initialiser des collections tant qu'elle supporte l'interface `System.Collection.Generic, ICollection<T>.` Dans ce cas les items seront initialisés et `ICollection<T>.Add(T)` sera appelé automatiquement pour les ajouter à la liste.

Il est ainsi possible d'avoir des initialisations imbriquées (comme l'exemple du mini studio) au sein d'initialisation de collections et inversement ainsi que toute combinaison qu'on peut imaginer. Bien plus qu'un simple artifice, on se rend compte que les expressions d'initialisation d'objets ne sont en réalité pas si anecdotique que ça, même si comparée aux autres nouveautés de C# 3.0 elles semblent moins essentielles.

Produire un code clair, lisible et maintenable est certes moins éblouissant de prime abord que de montrer des expressions Lambda, mais au bout du compte c'est peut-être ce qui est le plus important en production…

Les types anonymes

Partons maintenant pour la cinquième dimension, *twilight zone* !, et abordons ce qui semble un contresens dans un langage très fortement typé : les types.. anonymes.

Imaginons une instance d'une classe qui n'a jamais été définie mais qui peut malgré tout posséder des propriétés (que personne n'a créées) auxquelles ont peut accéder !

Regardons le code suivant :

```
var truc = new { Couleur = Color.Red, Forme = "Carré" };
var bidule = new { Marque = "Intel", Type = "Xeon", Coeurs = 4 };

Console.WriteLine("la couleur du truc est " + truc.Couleur +
                  " et sa forme est un " + truc.Forme);
Console.WriteLine("le processeur est un " + bidule.Type + " de chez " +
                  bidule.Marque + " avec " + bidule.Coeurs + " coeurs.");
```

Ce code produira la sortie suivante :

```
la couleur du truc est Color [Red] et sa forme est un Carré
le processeur est un Xeon de chez Intel avec 4 coeurs.
```

C# créé bien des classes (un type) pour chacune des variables (`truc` et `bidule` dans l'exemple ci-dessus). Si on affiche le type (par `GetType()`) de ces dernières on obtient :

```
le type de truc est <>f__AnonymousType0`2[System.Drawing.Color,System.String]
le type de bidule est
<>f__AnonymousType1`3[System.String,System.String,System.Int32]
```

Plus fort, si nous créons un objet `trucbis` définit de la même façon que l'objet `truc` et que nous inspectons les types, nous trouverons le même type que l'objet `truc`, ce qui les rend compatibles pour d'éventuelles manipulations groupées ! Les propriétés doivent être définit dans le même ordre, si nous inversions `Couleur` et `Forme`, un nouveau type sera créé par le compilateur.

Normalement sous C# nous ne sommes pas habitué à ce que l'ordre des membres dans une classe ait une importance, mais il faut bien concevoir que tous les nouveaux éléments syntaxiques de C# 3.0 servent en réalité à l'implémentation de Linq, et Linq a besoin de pouvoir créer de types qui n'existent pas, par exemple créer un objet qui représente chaque ligne du résultat d'un SELECT, et il a besoin de faire la différence dans l'ordre des champs puisque dans un SELECT l'ordre peut avoir une importance.

Conclusion

Les nouveaux éléments syntaxiques de C# 3.0 ne s'arrêtent pas là puisqu'il y a aussi tout ce qui concerne Linq et le requêtage des données. Mais avant d'aborder Linq dans un prochain article il était essentiel de fixer les choses sur les nouvelles syntaxes introduites justement pour servir Linq.

Une fois les types anonymes compris, les expressions Lambda digérées et tout le reste, il vous semblera plus facile d'aborder la syntaxe et surtout l'utilisation de Linq qui fait une utilisation débridée de ces nouveautés !

Espérant vous avoir éclairé utilement, je vous souhaite un happy coding !

C# 4.0 : les nouveautés du langage

Visual Studio 2010 beta 2 est maintenant accessible au public et il devient donc possible de vous parler des nouveautés sans risque de violer le NDA qui courrait jusqu'à lors pour les MVP et autres early testers de ce produit.

Les évolutions du langage commencent à se tasser et la mouture 4.0 est assez loin des annonces fracassantes qu'on a pu connaître avec l'arrivée des génériques ou des classes statiques et autres nullables de C# 2.0, ni même avec LINQ ou les expressions Lambda de C# 3.0.

Pour la version 4 du langage on compte pour l'instant peu d'ajouts (le produit ne sortira qu'en 2010 et que d'autres features pourraient éventuellement apparaître). On peut regrouper les 3 principales nouveautés ainsi :

- Les types dynamiques (dynamic lookup)
- Les paramètres nommés et les paramètres optionnels
- Covariance et contravariance

Paramètres optionnels

Il est en réalité bien étrange qu'il ait fallu attendre 4 versions majeures de C# pour voir cette syntaxe de Delphi refaire surface tellement son utilité est évidente.
De quoi s'agit-il ? Vous avez tous écrits du code C# du genre :

```
1:  MaMethode(typeA param1, typeB param2, typeC param3) …;
2:  MaMethode(typeA param1, typeB param2) { MaMethode(param1, param2, null) }
3:  MaMethode(typeA param1) { MaMethode(param1, null) }
4:  MaMethode() { MaMethode(null) }
```

Et encore cela n'est qu'un exemple bien court. Des librairies entières ont été écrites en C# sur ce modèle afin de permettre l'appel à une même méthode avec un nombre de paramètres variable. Le Framework lui-même est écrit comme cela.
Bien sûr il existe "params" qui autorise dans une certaine mesure une écriture plus concise, mais dans une certaine mesure seulement. Dans l'exemple ci-dessus le remplacement des valeurs manquantes par des nulls est une simplification. Dans la réalité les paramètres ne sont pas tous des objets ou des nullables. Dans ces cas-là il faut spécifier des valeurs bien précises aux différents paramètres omis. Chaque valeur par défaut se nichant dans le corps de chacune des versions de la méthode, pour retrouver l'ensemble de ceux-ci il faut donc lire toutes les variantes et reconstituer de tête la liste. Pas très pratique.
Avec C# 4.0 cette pratique verbeuse et inefficace prend fin. Ouf !
Il est donc possible d'écrire une seule version de la méthode comme cela :

```
1:  MaMethode(bool param1=false, int param2=25, MonEnum param3 =
MonEnum.ValeurA)  …
```

Grâce à cette concision l'appel à "MaMethode(true)" sera équivalente à "MaMethode(true, 25, MonEnum.ValeurA)". Le premier paramètre est fixé par l'appelant (c'est un exemple), mais les deux autres étant oubliés ils se voient attribuer automatiquement leur valeur par défaut.
Pas de surcharges inutiles de la méthode, toutes les valeurs par défaut sont accessibles dans une seule déclaration. Il reste encore quelques bonnes idées dans Delphi que Anders pourraient reprendre comme les indexeurs nommés ou les if sans nécessité de parenthèses systématiques. On a le droit de rêver :-)

Comme pour se faire pardonner d'avoir attendu 4 versions pour ressortir les paramètres par défaut de leur carton, C# 4.0 nous offre un petit supplément :

Les paramètres nommés

Les paramètres optionnels c'est sympa et pratique, mais il est vrai que même sous Delphi il restait impossible d'écrire du code tel quel "MaMethode(true,,MonEnum.ValeurA)". En effet, tout paramètre doit recevoir une valeur et les paramètres "sautés" ne peuvent être remplacés par des virgules ce qui rendrait le code totalement illisible. C# 4.0 n'autorise pas plus ce genre de syntaxe, mais il offre la possibilité de ne préciser que quelques uns des paramètres optionnels en donnant leur nom.
La technique est proche de celle utilisée dans les initialiseurs de classe qui permettent d'appeler un constructeur éventuellement sans paramètre et d'initialiser certaines propriétés de l'instance en les nommant. Ici c'est entre les parenthèses de la méthode que cela se jouera. Pour suivre notre exemple précédent, si on veut ne fixer que la valeur de "param3" il suffit d'écrire :

```
1:  MaMethode(param3 : MonEnum.ValeurZ);
```

de même ces syntaxes seront aussi valides :

```
1:  MaMethode(true,param3:MonEnum.ValeurX);
2:  MaMethode(param3:MonEnum.ValeurY,param1:false);
```

En effet, l'ordre n'est plus figé puisque les noms lèvent toute ambigüité. Quant aux paramètres omis, ils seront remplacés par leur valeur par défaut.

Voici donc une amélioration syntaxique qui devrait simplifier beaucoup le code de nombreuses librairies, à commencer par le Framework lui-même !

Dynamique rime avec Polémique

Autre nouveauté de C# 4.0, les types dynamiques. Aie aie aie…

Dynamique. C'est un mot qui fait jeune, sautillant, léger. Hélas. Car cela ne laisse pas présager du danger que représente cette extension syntaxique ! La polémique commence ici et, vous l'aurez compris, je ne suis pas un fan de cette nouveauté :-)

Techniquement et en deux mots cela permet d'écrire "MaVariable.MethodeMachin()" sans être sûr que l'instance pointée par MaVariable supporte la méthode MethodeMachin(). Et ça passe la compilation sans broncher. Si çà pète à l'exécution, il ne faudra pas venir se plaindre. Le danger du nouveau type "dynamic" est bien là. Raison de mes réticences…
Si on essaye d'être plus positif il y a bien sûr des motivations réelles à l'implémentation des dynamiques. Par exemple le support par .NET des langages totalement dynamiques comme Python et Ruby (les dynamique de C# 4 s'appuient d'ailleurs sur le DLR), même si ces langages sont plus des gadgets amusants que l'avenir du développement (avis personnel).
Les dynamiques simplifient aussi l'accès aux objets COM depuis IDispatch, mais COM n'est pas forcément non plus l'avenir de .NET (autre avis personnel).

Les deux autres emplois des dynamiques qui peuvent justifier leur existence sont l'accès simplifié à des types .NET au travers de la réflexion (pratique mais pas indispensable) ou bien des objets possédant une structure non figée comme les DOM HTML (pratique mais à la base de pas mal de code spaghetti).

Bref, les dynamiques ça peut être utile dans la pratique, mais ce n'est pas vraiment une nouvelle feature améliorant C# (comme les autres ajouts jusqu'à maintenant). Le danger de supporter un tel type est-il compensé par les quelques avantages qu'il procure ? C'est là que dynamique rime avec polémique !
Pour moi la réponse est non, mais je suis certain que ceux qui doivent jongler avec du COM ou des DOM Html penseront le contraire.
J'arrête de faire le grognon pour vous montrer un peu mieux la syntaxe. Car malgré tout le dynamisme n'est pas une invitation au chaos. Enfin si. Mais un chaos localisé. C'est à dire que l'appel à une méthode non existante reste impossible partout, sauf pour un objet déclaré avec le nouveau type "dynamic" :

```
1:  dynamic x;
2:  x = Machin.ObtientObjetDynamique();
3:  x.MethodeA(85); // compile dans tous les cas
4:
5:  dynamic z = 6; // conversion implicite
6:  int i = z; // sorte de unboxing automatique
```

Bien entendu le “dynamisme” est total : cela fonctionne sur les appels de méthodes autant que sur les propriétés, les délégués, les indexeurs, etc.

Le compilateur va avoir pour charge de collecter le maximum d’information sur l’objet dynamique utilisé (comment il est utilisé, ses méthodes appelées…), charge au runtime du Framework de faire le lien avec la classe de l’instance qui se présentera à l’exécution. C’est du late binding avec tout ce qui va avec notamment l’impossibilité de contrôler le code à la compilation.

A vous de voir, mais personnellement je déconseille fortement l’utilisation des dynamiques qui sont comme un gros interrupteur ajouté en façade de C# “Langage Fortement Typé On/Off”. Restez dans le mode “On” et ne passez jamais en mode “Off” !

Covariance et Contravariance ou le retour de l’Octothorpe

J’adore le jargon de notre métier. “*Comment passer pour un hasbeen en deux secondes à la machine à café*” est une mise en situation comique que j’utilise souvent, certainement influencé par mon passé dans différentes grosses SSII parisiennes et par la série Caméra Café de M6…
Ici vous aurez l’air stupide lorsque quelqu’un lancera “Alors t’en penses quoi de la contravariance de C#4.0 ?”… L’ingé le plus brillant qui n’a pas lu les blogs intéressants la veille sera dans l’obligation de plonger le nez dans son café et de battre en retraire piteusement, prétextant un truc urgent à finir…
Covariance et contravariance sont des termes académiques intimidants. Un peu comme si on appelait C# “C Octothorpe”. On aurait le droit. Octothorpe est l’un des noms du symbole #. Mais franchement cela serait moins sympathique que “do dièse” (C# est la notation de do dièse en américain, à condition de prononcer le # comme “sharp” et non “square” ou “octothorpe”).

Un support presque parfait sous C# 1 à 3

Un peu comme monsieur Jourdain faisait de la prose sans le savoir, la plupart d’entre nous a utilisé au moins la covariance en C# car il s’agit de quelque chose d’assez naturel en programmation objet et que C# le supporte pour la majorité des types. D’ailleurs la covariance existe depuis le Framework 2.0 mais pour certains cas (couverts par C# 4.0) il aurait fallu émettre directement du code IL pour s’en servir.

C# 4.0 n’ajoute donc aucune nouvelle fonctionnalité ou concept à ce niveau, en revanche il comble une lacune des versions 1 à 3 qui ne supportaient pas la covariance et la contravariance pour les délégués et les interfaces dans le cadre de leur utilisation avec les génériques. Un cas bien particulier mais devant lequel on finissait pas tomber à un moment ou un autre.

Un besoin simple

C# 4.0 nous assure simplement que les choses vont fonctionner comme on pourrait s'y attendre, ce qui n'était donc pas toujours le cas jusqu'à lors.

Les occasions sont rares où interfaces et délégués ne se comportent pas comme prévu sous C#, très rares. Mais cela peut arriver. Avec C# 4.0 ce sont ces situations rares qui sont supprimées. De fait on pourrait se dire qu'il n'y a rien à dire sur cette nouveauté de C# 4.0 puisqu'on utilisait la covariance et la contravariance sans s'en soucier et que la bonne nouvelle c'est qu'on va pouvoir continuer à faire la même chose !

Mais s'arrêter là dans les explications serait un peu frustrant.

Un exemple pour mieux comprendre

Supposons les deux classes suivantes :

```
1:  class Animal{ }
2:  class Chien: Animal{ }
```

La seconde classe dérive de la première. Imaginons que nous écrivions maintenant un délégué définissant une méthode retournant une instance d'un type arbitraire :

```
1:  delegate T MaFonction<T>();
```

Pour retourner une instance de la classe Chien nous pouvons écrire :

```
1:  MaFonction<Chien> unChien = () => new Chien();
```

Vous noterez l'utilisation d'une expression Lambda pour définir le délégué. Il s'agit juste d'utiliser la syntaxe la plus concise. On pourrait tout aussi bien définir d'abord une fonction retournant un Chien, lui donner un nom, puis affecter ce dernier à la variable "unChien" comme dans le code ci-dessous :

```
1:  public Chien GetChien()
2:  {
3:      return new Chien();
4:  }
5:
6:  MaFonction<Chien> unChien = GetChien; // sans les () bien sur !
```

Partant de là, il est parfaitement naturel de se dire que le code suivant est valide :

```
1:  MaFonction<Animal> animal = unChien;
```

En effet, la classe Chien dérivant de Animal, il semble légitime de vouloir utiliser le délégué de cette façon. Hélas, jusqu'à C# 3.0 le code ci-dessus ne compile pas.

La Covariance

La covariance n'est en fait que la possibilité de faire ce que montre le dernier exemple de code. C# 4.0 introduit les moyens d'y arriver en introduisant une nouvelle syntaxe. Cette dernière consiste tout simplement à utiliser le mot clé "*out*" dans la déclaration du délégué:

```
1:  delegate T MaFonction<out T>();
```

Le mot clé "*out*" est déjà utilisé en C# pour marquer les paramètres de sortie dans les méthodes. Mais il s'agit bien ici d'une utilisation radicalement différente. Pourquoi "*out*" ? Pour marquer le fait que le paramètre sera utilisé en "sortie" de la méthode.
La covariance des délégués sous C# 4.0 permet ainsi de passer un sous-type du type attendu à tout délégué qui produit en sortie (out) le type en question.

Si vous pensez que tout cela est bien compliqué, alors attendez deux secondes que je vous parle de contravariance !

La Contravariance

Si la covariance concerne les délégués et les interfaces utilisés avec les types génériques dans le sens de la sortie (*out*), et s'il s'agit de pouvoir utiliser un sous-type du type déclaré, ce qui est très logique en POO, la contravariance règle un problème inverse : autoriser le passage d'un super-type non pas en sortie mais en entrée d'une méthode.

Un exemple de contravariance

Pas de panique ! un petit exemple va tenter de clarifier cette nuance :

```
1:  delegate void Action1<in T>(T a);
2:
3:  Action1<Animal> monAction = (animal) => { Console.WriteLine(animal);
};
4:  Action1<Chien> chien1 = monAction;
```

Bon, ok. Paniquez. !!!

Ici un délégué est défini comme une méthode ayant un paramètre de type arbitraire. Le mot clé "*in*" remplace "*out*" de la covariance car le paramètre concerné est fourni en entrée de la méthode (*in*).

La plupart des gens trouve que la contravariance est moins intuitive que la covariance, et une majorité de développeurs trouve tout cela bien complexe. Si c'est votre cas vous êtes juste dans la norme, donc pas de complexe :-)

La contravariance se définit avec le mot clé "*in*" simplement parce que le type concerné est utilisé comme paramètre d'entrée. Encore une fois cela n'a rien à voir avec le sens de "in" dans les paramètres d'entrée des méthodes. Tout comme "out" le mot clé "in" est utilisé ici dans un contexte particulier, au niveau de la déclaration d'un type générique dans un délégué.
Avec la contravariance il est donc possible de passer un super-type du type déclaré. Cela semble contraire aux habitudes de la POO (passer un sous-type d'un type attendu est naturel mais pas l'inverse). En réalité la contradiction n'est que superficielle. Dans le code ci-dessus on s'aperçoit qu'en réalité "monAction" fonctionne avec n'importe quelle instance de "Animal", un Chien étant un Animal, l'assignation est parfaitement légitime !

M'sieur j'ai pas tout compris !

Tout cela n'est pas forcément limpide du premier coup, il faut l'avouer.

En réalité la nouvelle syntaxe a peu de chance de se retrouver dans du code "de tous les jours". En revanche cela permet à C# de supporter des concepts de programmation fonctionnelle propres à F# qui, comme par hasard, est aussi fourni de base avec .NET 4.0 et Visual Studio 2010. Covariance et contravariance seront utilisées dans certaines librairies et certainement dans le Framework lui-même pour que, justement, les délégués et les interfaces ainsi définis puissent être utilisés comme on s'y attend. La plupart des développeurs ne s'en rendront donc jamais compte certainement… En revanche ceux qui doivent écrire des librairies réutilisables auront tout intérêt à coder en pensant à cette possibilité pour simplifier l'utilisation de leur code.

Et les interfaces ?

Le principe est le même. Et comme je le disais la plupart des utilisations se feront dans des librairies de code, comme le Framework l'est lui-même. Ainsi, le Framework 4.0 définit déjà de nombreuses interfaces supportant covariance et contravariance. IEnumerable<T> permet la covariance de T, IComparer<T> supporte la contravariance de T, etc. Dans la plupart des cas vous n'aurez donc pas à vous souciez de tout cela.

Lien

La documentation est pour l'instant assez peu fournie, et pour cause, tout cela est en bêta ne l'oublions pas. Toutefois la sortie de VS2010 et de .NET 4.0 est prévue pour Mars 2010 et le travail de documentation a déjà commencé sur MSDN. Vous pouvez ainsi vous référer à la série d'articles sur MSDN : Covariance and Contravariance.

Conclusion

Les nouveautés de C# 4.0, qui peuvent toujours changer dans l'absolu puisque le produit est encore en bêta, ne sont pas à proprement parler des évolutions fortes du langage. On voit bien que les 3 premières versions ont épuisé le stock de grandes nouveautés hyper sexy comme les génériques ou Linq qui ont modifié en profondeur le langage et décuplé ses possibilités.

C# 4.0 s'annonce comme une version mature et stable, un palier est atteint. les nouveautés apparaissent ainsi plus techniques, plus "internes" et concernent moins le développeur dans son travail quotidien.

Une certaine convergence avec F# et le DLR pousse le langage dans une direction qui ouvre la polémique. Je suis le premier a resté dubitatif sur l'utilité d'une telle évolution surtout que la sortie de F# accompagnera celle de C# 4.0 et que les passionnés qui veulent à tout prix coder dans ce style pourront le faire à l'aide d'un langage dédié. Mélanger les genre ne me semble pas un avantage pour C#.

C# est aujourd'hui mature et il est peut-être temps d'arrêter d'y toucher…
L'ensemble .NET est d'ailleurs lui-même arrivé à un état de complétude qu'aucun framework propriétaire et cohérent n'avait certainement jamais atteint.
.NET a tout remis à plat et à repousser les limites sur tous les fronts.
On peut presque affirmer que .NET est aujourd'hui "complet". Même si la plateforme va encore évoluer dans l'avenir. Mais tous les grands blocs sont présent, des communications à la séparation code / IHM, des workflows aux interfaces graphiques et multitouch, de LINQ au Compact Framework.

Quand un système arrive à un haut niveau de stabilité, le prochain est déjà là, sous notre nez mais on ne le sait pas. Le palier atteint par .NET 4.0 marque une étape importante. Cet ensemble a coûté cher, très cher à développer. Il s'installe pour plusieurs années c'est une évidence (et une chance !). Mais on peut jouer aux devinettes : quelle sera la prochaine grande plateforme qui remplacera .NET, quel langage remplacera C# au firmament des langages stars pour les développeurs dans 10 ans ?

Bien malin celui qui le devinera, mais il est clair que tout palier de ce type marque le sommet d'une technologie. De quelle taille est le plateau à ce sommet ? Personne ne peut le prédire, mais avec assurance on peut affirmer qu'après avoir grimpé un sommet, il faut le redescendre. Quelle sera la prochaine montagne à conquérir ? Il y aura-t-il un jour un .NET 10 ou 22 ou bien quelque chose d'autre, de Microsoft ou d'un autre éditeur, l'aura-t-il supplanté ?

C'est en tout cas une réalité qui comme l'observation des espaces infinis qu'on devine dans les clichés de Hubble laisse songeur…

Quoi de neuf dans C# 5 ?

Contre vents et marées, ce fantastique langage qu'est C# continue son éternelle mutation, comme un papillon qui n'en finirait pas de renaitre de son cocon, toujours plus beau à chaque fois. Dernièrement j'ai beaucoup parlé de WinRT et Windows 8, et j'en reparlerai tout l'été pour préparer la rentrée ! Mais lorsque tout cela sera enfin sur le marché la version 5 de C# le sera aussi et il serait bien dommage de l'oublier. Quelles nouvelles parures arbore notre papillon dans cette mouture ?

C#5 Une évolution logique

Selon comment vous regarderez C#5 vous le trouverez révolutionnaire ou bien simplement dans la suite des améliorations déjà immenses des versions précédentes.

C# 5 est "tout simplement" une suite logique en adéquation avec les besoins des développeurs.

D'un côté peu de nouveautés aussi faramineuses qu'à pu l'être Linq par exemple, et de l'autre des avancées absolument nécessaires pour être en phase avec les exigences des applications modernes.

Des petites choses bien utiles...

Informations de l'appelant

Parmi ces petites choses bien utiles on trouve les informations de la méthode appelante. C'est simple, ce n'est pas le truc qui scotche, mais cela permet par exemple d'écrire en quelques lignes son propre logger sans être dépendant d'une grosse librairie externe comme Log4Net ou d'autres.

On connaissait déjà les paramètres optionnels introduits par C# 4 et qui permettent d'écrire un code de ce type :

```
1: public void MaMethode(int a = 123, string b = "Coucou") { ... }
2: MaMethode(753);  // compilé en MaMethode(753, "Coucou")
3: MaMethode();     // compilé en MaMethode(123, "Coucou")
```

Sous C# 4 la valeur des paramètres était forcément une constante. C# 5 introduit la possibilité d'utiliser un attribut qui ira chercher la valeur au runtime parmi les informations de la méthode appelante (appelante et non appelée ce qui fait toute la différence ici).

De fait, il devient possible d'écrire un code comme celui-ci :

La méthode de log

```
1: public static void Trace(string message,
2:                          [CallerFilePath] string sourceFile = "",
3:                          [CallerMemberName] string memberName = "")
4: {
5:     var msg = String.Format("{0}: {1}.{2}: {3}",
6:                   DateTime.Now.ToString("yyyy-mm-dd HH:MM:ss.fff"),
7:                   Path.GetFileNameWithoutExtension(sourceFile),
8:                   memberName, message);
9:   MyLogger.Log(msg);
```

Ici rien de très spécial, sauf la présence des attributs dans la déclaration des paramètres optionnels. Pour faire court le code suppose une infrastructure hypothétique "MyLogger" qui elle stocke le message (ou l'envoie sur le web ou ce que vous voulez).

Grâce à cette astuce il est très facile de logger des messages dans son code en utilisant son propre code "Trace" :

```
1: // Fichier CheckUser.cs
2: public void CheckUserAccount(string userName)
3: {
4: // compilé en Trace("Entrée dans la méthode", "CheckUser.cs", "CheckUserAccount")
5:   Trace("Entrée dans la méthode");
6:   // ...
7:   Trace("Sortie de la méthode");
8: }
```

A première vue ce n'est pas révolutionnaire en effet. Pratique en revanche. A seconde vue ce n'est toujours pas révolutionnaire mais ça peut s'utiliser de façon plus pratique...

Prenons le cas de l'interface INotifyPropertyChanged qui demande à passer le nom de la propriété dans l'évènement. Il existe tout un tas de "ruses" dans certaines librairies pour soit

contrôler le nom passé, soit tenter comme Jounce d'éviter de le taper. Toutes ces tentatives sont essentielles car une simple erreur d'orthographe ou tout bêtement un refactoring du nom d'une propriété peut casser toute la belle logique d'un databinding...
En y réfléchissant bien, les nouveaux attributs de paramètres optionnels peuvent être utilisés pour régler définitivement ce problème récurrent, de façon efficace, simple et uniquement en utilisant le langage et ses possibilités :

```
   1: public class ViewModelBase : INotifyPropertyChanged {
   2:   protected void Set<T>(ref T field, T value, [CallerMemberName] string
propertyName = "")
   3:  {
   4:     if (!Object.Equals(field, value))
   5:     {
   6:       field = value;
   7:       OnPropertyChanged(propertyName);
   8:     }
   9:   }
  10:   // ...
  11: }
  12:
  13: public class Ecran1WM : ViewModelBase
  14: {
  15:   private int largeur;
  16:   public int Largeur
  17:   {
  18:     get { return largeur; }
  19:     set { Set(ref largeur, value); } //Le compilateur remplira avec "Largeur"
  20:   }
  21: }
```

Pas si simpliste que ça donc cet ajout de C#5 !

Variables de boucle et expression Lambda

Ici aussi il ne s'agit pas forcément d'une révolution. Quoi que...

Vous le savez peut-être (je dis bien peut-être car le sujet est loin d'être compris par tout le monde, même des développeurs confirmés) il ne faut pas utiliser les variables de boucle dans des expressions Lambda par exemple.

En effet, le fameux problème de "closure" fait que la variable encapsulée est celle de la boucle et qu'en général cela ne correspond absolument pas à l'effet escompté.

Je ne referai un pas speech sur les closures puisque la bonne nouvelle c'est qu'en réalité C# 5 fonctionne comme on s'y attendait sans plus avoir à se poser de question bizarre...

Un petit exemple pour ceux qui ont du mal à situer le problème. Le code suivant ne fait pas ce qu'on attend de lui :

```
 1: var nombres = GetNombres(1, 2, 3, 4, 5);
 2: foreach (var n in nombres)
 3:  {
 4:   Console.WriteLine(n(10));
 5:  }
 6:
 7: // Sortie réelle : 15 15 15 15 15
 8:
 9:  public static List<Func<int, int>> GetNombres(params int[] addends)
10:  {
11:   var funcs = new List<Func<int, int>>();
12:   foreach (int addend in addends)funcs.Add(i => i + addend);
13:   return funcs;
14:  }
```

Bref c'est pas très clair les closures pour plein de gens.

Au lieu d'avoir à créer une variable locale qui elle peut être capturée par la closure et éviter la catastrophe du code ci-dessus, C# 5 comprend la situation et fournira cette fois-ci le résultat attendu...

Cela supprimera des bugs bien sournois pas encore découverts et qui, au gré d'une recompilation en C# 5 disparaitront tous seuls sans que personne ne sache qu'ils ont pourtant été là !

... Aux grandes choses très utiles !

La programmation Asynchrone, l'épouvantail à développeur...
Ah, la programmation asynchrone... Il suffit d'en parler pour que le silence se fasse autour de la machine à café et que chacun trouve une excuse pour s'éclipser ! Il est vrai que le sujet à de quoi imposer le silence : soit on est un expert, soit il est préférable de ne rien dire de peur de dire une bêtise. D'ailleurs asynchrone c'est du multitâche ou ce n'en est pas ? Clac-clac font les dents dans le silence des vapeurs de café (ou les volutes de cigarettes en se

caillant sur le trottoir, mais là c'est le froid plus que la peur qui fait jouer des castagnettes aux quenottes !).

.NET et C# proposent des tas de moyens de faire de la programmation asynchrone et du multitâche, mais ces méthodes ne passionnaient guère de monde jusqu'à ce que les progrès du hardware ne passent plus par les GHz mais par le nombre de cœurs du CPU et jusqu'à ce que les services Web (et me Cloud) se démocratisent Et là, panique ! Trop peu de compétence pour un sujet si délicat.
Tout le monde a eu, à un moment ou un autre, "peur" du multitâche et de l'asynchrone. Mutex, Lock, Thread, ThreadPool, ThreadStart, WaitCallBack, Monitor.Enter, Monitor.Exit, TryEnter, Pulse et Wait, BeginGetResponse, EndGetResponse, objets immutables et j'en passe !!!

De quoi avoir le tournis je l'avoue.

Asynchrone ? Multitâche ?

Les deux choses sont très différentes et sont souvent confondues à tort.

Bref c'est un peu le bouillon. C# 5 s'intéresse à l'asynchrone, le multitâche avait plutôt été traité dans C# 4.

Multitâche
Le multitâche consiste à faire tourner plusieurs tâches en même temps. Ni plus ni moins. Il peut être utilisé en conjonction de la programmation asynchrone ou non, il n'y a pas de lien direct entre les deux techniques.

C# 5 ne propose rien de particulier concernant le multitâche proprement dit puisque cela a plutôt été l'une des avancées de C# 4 avec PLINQ et la classe Parallel. Alors passons à la suite...

Asynchrone
L'asynchrone est d'une autre nature. Il s'agit de faire exécuter une tâche (généralement sur une autre machine ou un périphérique) sans être sûr de quand arrivera la réponse (s'il y en a une) le tout sans bloquer le logiciel et son UI (ce sont ces considérations optionnelles qui peuvent amener à utiliser des techniques issu de la programmation multitâche, sans rapport direct avec l'asynchronisme ou faire penser que l'asynchronisme est du multitâche, vous suivez toujours ? !).

Le cas le plus fréquent aujourd'hui est la gestion des données. Qu'il s'agisse de véritables services Web ou d'équivalences techniques, les données sont de moins en moins accédées de façon directe.

L'écriture synchrone est facile. Le programme s'écrit au fil des lignes schématisant le temps qui coule de haut en bas dans le sens de lecture du code. Il est aisé d'entreprendre des actions, d'attendre leur réponse, de tester des valeurs, de passer à la suite. C'est la programmation "d'avant".

Avec un service Web, une requête SQL, Entity framework, etc, le temps n'est plus linéaire au sein du programme puisqu'en réalité d'autres machines (d'autres cœurs, d'autres ordinateurs plus "loin", d'autres périphériques) devront, chacun à leur rythme et en fonction de leur charge traiter un bout du problème et retourner une réponse. Tout ce qui prend du temps peut être rendu asynchrone pour rendre la main le plus vite possible, clé de la réactivité des OS modernes.

L'asynchronisme pose ainsi de gros problèmes d'écriture. Comment faire en sorte que le programme ne soit pas bloqué sur la ligne x, en attende de la réponse à la question "envoyée ailleurs" sans pour autant passer à la ligne x+1 qui doit elle attendre que les résultats soient là pour avoir un sens ? Le propre de l'asynchronisme est de ne pas être bloquant, et c'est bien là que ça... bloque ! Car comment continuer à travailler sur des données qu'on a demandé si elles ne sont pas encore là...

Grâce aux méthodes anonymes de C# il était plus ou moins facile de résoudre le problème en programmant l'action a effectuer sur la réponse au sein d'un Callback. Charge au développeur de gérer les conséquences de tout cela : que faire pendant qu'on attend quelque chose ? rien ? passer à autre chose ? Que faire quand on sera interrompu, "plus tard", par la réponse qui enfin arrivera ?

Tout cela peut se résoudre en appliquant des guides lines précises et en maitrisant, notamment sous Silverlight, WPF et demain WinRT, les notions de databinding, les méthodes de travail de type MVVM, et bien entendu les bases mêmes à la fois du multitâches et de la programmation asynchrone. Car dans la pratique c'est en faisant un savant mélange de toutes ces choses qu'on arrive à écrire un programme fluide et réactif.

Simplifions un peu

Toutefois, si l'utilisation des méthodes anonymes a rendu les traitements asynchrones plus faciles à orchestrer, elles n'ont pas résolu tous les problèmes. Loin s'en faut.

Lorsqu'une application Silverlight doit par exemple demander une liste d'items sur laquelle elle doit effectuer un autre traitement asynchrone (le tout via RIA Services par exemple), les

callbacks s'imbriquent les uns dans les autres pour devenir illisibles. S'il faut en même temps prendre en charge la gestion d'éventuelles erreurs, leur log, etc, le code peut devenir rapidement imbuvable, donc in-maintenable.

La programmation asynchrone se généralisant il fallait trouver un moyen de simplifier tout ça.

Async et Await
C# 5 vient à la rescousse avec deux nouvelles instructions. Async et Await.
Et c'est plus simple encore que cela en a l'air au regard de la complexité du sujet.

Async est utilisé pour marquer une méthode. Cette marque est faite à l'intention du compilateur pour lui dire "à l'intérieur de cette méthode je veux écrire mon code comme si tout était synchrone". C'est le compilateur (et la plateforme) qui vont se charger du reste. Pour la petite histoire, cette bonne idée a été reprise de F# d'ailleurs.

Il est important de marquer une méthode avec Async car cela est utilisé par les appelants qui doivent savoir qu'ils auront certainement la main avant que le travail de la méthode appelée ne soit terminé. L'impact dépasse donc la méthode marquée pour atteindre tout code qui en fera l'usage.

Await est simplement utilisé comme une sorte de préfixe devant une instruction asynchrone pour dire au compilateur de se débrouiller pour que tout ce passe "comme si" l'appel était bloquant. On revient à une écriture synchrone du code. C'est donc une grande simplification. Mais attention l'astuce est plus complexe, j'y reviendrai certainement, car les appels ne sont pas réellement bloquants... la méthode prendra souvent fin et reviendra à l'appelant avant que le job des tâches asynchrones ne soit terminé. Ce sont bien des callbacks et non des points bloquants...
En fait, C# 5 va bien écrire les callbacks à notre place. Mais comme il le fait pour nous le code qu'on écrit redevient clair et limpide. Il ne faut pas se laisser abuser par cette apparence donc.

Voici un exemple de code utilisant Async et Await :

```
1: public async void ShowReferencedContent(string filename)
2: {
3:   var url = await BeginReadFromFile(filename);
4:   var contentOfUrl = await BeginHttpGetFromUrl(url);
5:   MessageBox.Show(contentOfUrl);
6: }
```

Dans la méthode ci-dessus on remarque deux choses : la méthode elle-même est marquée avec le mot clé “async” comme expliqué plus haut, et les lignes 3 et 4 utilise “await” en préfixe du code exécuté. Ces deux lignes de code font des appels à des méthodes asynchrones. Normalement le programme passerait à la ligne 4 avant que le résultat de la ligne 3 ne soit connu. Et forcément ça marcherait moins bien...

Mais ici, inutile d’écrire des méthodes anonymes passées en paramètres de méthodes asynchrones acceptant des callbacks (rien que l’écrire c’est compliqué !). C#5 se chargera de tout. Le code “semble” synchrone, mais il reste asynchrone seule l’imbrication des callbacks est écrite à notre place.

Vous allez penser que c’est très bien, mais que se passe-t-il si on souhaite que la méthode retourne une réponse à ses appelants ?

Comme une méthode “async” pourra se terminer avant que son travail ne soit réellement fini, il va falloir trouver un moyen d’indiquer à l’appelant qu’en plus elle retourne une valeur qui ne viendra que “plus tard”. On utilise alors une notation un peu différente pour son entête.

Par exemple, si la méthode doit retourner un “string”, son entête ne sera pas “public async string maméthode()” mais elle utilisera la classe générique **Task** pour retourner un Task<string>.
Une instance de **Task** représente un “bout de travail” qui peut éventuellement retourner une valeur. L’appelant peut examiner l’objet Task pour connaître son état et sa valeur de retour.
Un tel code ressemblera à cela :

```
1: public static async Task<string> GetReferencedContent(string filename)
2: {
3:    string url = await BeginReadFromFile(filename);
4:    string contentOfUrl = await BeginHttpGetFromUrl(url);
5:    return contentOfUrl;
6: }
```

On note la particularité suivante : la méthode retourne un string alors que le type est Task<string>. Ici aussi c’est le compilateur qui prend en charge la transformation du string en Task<string>.

Désormais un appelant peut utiliser la méthode comme bon lui semble : dans un mode “à la synchrone” avec await, ou bien en attendant “manuellement” le résultat, en sondant régulièrement le Task pour connaître son état...

Ceux qui ont déjà utilisé les méthodes asynchrones de .NET 4 noteront que les paires de méthodes "Begin / End” (comme WebRequest.BeginGetResponse / WebRequest.EndGetResponse), si elles existent toujours sous .NET 4.5 ne sont pas utilisables avec “await” (les Beginxxx nécessitent un appel de méthode explicite à l’intérieur du callback pour obtenir la réponse notamment). A la place, .NET 4.5 fournit de nouvelles méthodes qui retournent un Task. Ainsi, au lieu par exemple d’appeler WebRequest.BeginGetResponse, on utilisera WebRequest.GetResponseAsync.

Un petit exemple pour clarifier :

```
 1: private static async Task<string> GetContent(string url)
 2: {
 3:   WebRequest wr = WebRequest.Create(url);
 4:   var response = await wr.GetResponseAsync();
 5:   using (var stm = response.GetResponseStream())
 6:   {
 7:     using (var reader = new StreamReader(stm))
 8:     {
 9:       var content = await reader.ReadToEndAsync();
10:       return content;
11:     }
12:   }
13: }
```

Conclusion

C# a tellement évolué depuis sa première version qu’on se demande comment il est possible de trouver encore matière à faire de nouvelles versions... Mais c’est sans compter sur les besoins mêmes du développement qui évoluent.

Jusqu’à C# la plupart des langages disparaissaient lorsqu’ils devenaient inadaptés. C# a connu des modifications d’une profondeur rarement atteinte par aucun langage professionnel.

C# était à sa sortie une sorte de Java avec des éléments de syntaxe de Delphi (comme les propriétés). Delphi et C# ayant le même père on sentait la proximité tout comme l’influence de Java qui avait valu quelques soucis à Microsoft à l’époque avant d’être abandonné. Puis,

en une dizaine d'années, au fil des versions, C# est devenu un langage totalement différent de ses sources d'inspiration. Intégrant rapidement des techniques empruntées à d'autres langages, ajoutant ses propres bonnes idées ou piochant dans des langages essayistes tels que F#.

Avec C# 5, fonctionnant avec WinRT, WPF et Silverlight, nous allons disposer d'un langage encore plus proche de nos besoins quotidiens pour produire des logiciels de plus en plus sophistiqués, réactifs, fluides, designés, s'adaptant à différents form factors.

Loin de se spécialiser (et perdre de sa souplesse) ou de trop se généraliser (et de se noyer dans trop d'options), C# impose son style unique qui en fait un allié de premier plan pour programmer des logiciels modernes tout en nous permettant de maitriser la complexité croissante du code à produire.

Certaines modes voudraient faire venir au premier plan pour développer de vraies applications de vieux langages utilitaires interprétés dont l'indigence laisse pantois. Ne vous laissez pas convaincre par ces leurres, servez-vous de vos connaissances, rentabilisez vos diplômes et demandez-vous si un informaticien professionnel qui ne sait pas faire la différence en javaScript et C# 5 a encore le droit de se proclamer professionnel...

C# - Les optimisations du compilateur dans le cas du tail-calling

Les optimisations du compilateur C# ne sont pas un sujet de discussion très courant, en tout cas on voit très nettement que le fait d'avoir quitté l'environnement Win32 pour l'environnement managé de .NET a fait changer certaines obsessions des codeurs... Ce n'est pas pour autant que le sujet a moins d'intérêt ! Nous allons voir cela au travers d'une optimisation particulière appelée "tail-calling" (appel de queue, mais je n'ai pas trouvé de traduction française, si quelqu'un la connaît, qu'il laisse un commentaire au billet).

Principe

On appelle "tail-calling" un mécanisme d'optimisation du compilateur qui permet d'économiser les instructions exécutées en même temps que des accès à la pile. Les circonstances dans lesquelles le compilateur peut utiliser cette optimisation sont celles où une méthode se termine par l'appel d'une autre, d'où le nom de tail-calling (appel de queue).

Prenons l'exemple suivant :

```
static public void Main()
{
    Go();
}

static public void Go()
{
    Première();
    Seconde();
    Troisième();
}

static public void Troisième()
{
}
```

Dans cet exemple le compilateur peut transformer l'appel de Troisième() en un appel de queue (tail-calling). Pour mieux comprendre regardons l'état de la pile au moment de l'exécution de *Seconde()* :Seconde()-Go()-Main()

Quand Troisième() est exécutée il devient possible, au lieu d'allouer un nouvel emplacement sur la pile pour cette méthode, de simplement remplacer l'entrée de Go() par Troisième(). La pile ressemble alors à Troisième()-Main().

Quand Troisième() se terminera elle passera l'exécution à Main() au lieu de transférer le trait àSeconde() qui immédiatement devra le transférer à Main().
C'est une optimisation assez simple qui, cumulée tout au long d'une application, et ajoutée aux autres optimisations, permet de rendre le code exécutable plus efficace.

Quand l'optimisation est-elle effectuée ?

La question est alors de savoir quand le compilateur décide d'appliquer l'optimisation de tail-calling. Mais dans un premier temps il faut se demander de quel compilateur nous parlons.... Il y existe en effet deux compilateurs dans .NET, le premier prend le code source pour le compiler en IL alors que le second, le JIT, utilisera ce code IL pour créer le code natif. La compilation en IL peut éventuellement placer certains indices qui permettront de mieux repérer les cas où le tail-calling est applicable mais c'est bien entendu dans le JIT que cette optimisation s'effectue.

Il existe de nombreuses règles permettant au JIT de décider s'il peut ou non effectuer l'optimisation. Voici un exemple de règles qui font qu'il n'est pas possible d'utiliser le tail-calling (par force cette liste peut varier d'une implémentation à l'autre du JIT) :

- L'appelant ne retourne pas directement après l'appel;
- Il y a une incompatibilité des arguments passés sur la pile entre l'appelant et l'appelé ce qui imposerait une modification des arguments pour appliquer le tail-calling;
- L'appelant et l'appelé n'ont pas le même type de retour (données de type différents, void);
- L'appel est transformé en inline, l'inlining étant plus efficace que le tail-calling et ouvrant la voie à d'autres optimisations;
- La sécurité interdit ponctuellement d'utiliser l'optimisation;
- Le compilateur, le profiler, la configuration ont coupé les optimisations du JIT.
- Pour voir la liste complète des règles, jetez un oeil à ce post.

Intérêt de connaitre cette optimisation ?

Normalement les optimisations du JIT ne sont pas un sujet intéressant au premier chef le développeur. D'abord parce qu'un environnement managé comme .NET fait qu'à la limite ce sont les optimisations du code IL qui regarde directement le développeur et beaucoup moins la compilation native qui peut varier d'une plateforme à l'autre pour une même application. Ensuite il n'est pas forcément judicieux de se reposer sur les optimisations du JIT puisque, justement, ce dernier peut être différent sans que l'application ne le sache.

Qui s'intéresse à l'optimisation du tail-calling alors ? Si vous écrivez un profiler c'est une information intéressante, mais on n'écrit pas un tel outil tous les jours... Mais l'information est intéressante lorsque vous déboguez une application car vous pouvez vous trouver face à une pile d'appel qui vous semble "bizarre" ou "défaillante" car il lui manque l'une des méthodes appelées !

Et c'est là que savoir qu'il ne faut pas chercher dans cette direction pour trouver le bug peut vous faire gagner beaucoup de temps... Savoir reconnaître l'optimisation de tail-calling évite ainsi de s'arracher les cheveux dans une session de debug un peu compliquée si on tombe en plus sur un pile d'appel optimisée. Un bon debug consiste à ne pas chercher là où ça ne sert à rien (ou à chercher là où c'est utile, mais c'est parfois plus difficile à déterminer !), alors rappelez-vous du tail-calling !

Déboguer simplement : les points d'arrêt par code

Voici une astuce toute simple comme je les aime mais qui rend bien des services !

Lorsqu'on débogue une application on utilise fréquemment les points d'arrêt notamment lorsqu'on soupçonne un problème dans une partie de code précise. Tous les développeurs connaissent les points d'arrêt et savent s'en servir. Il est déjà plus rare de voir un développeur se servir des points d'arrêt conditionnels, pourtant un simple clic-droit sur le rond rouge dans la marge (symbolisant le point d'arrêt) permet de fixer une condition liée au code ou au nombre de passages par exemple. Il existe d'autres possibilités d'une grande richesse et si vous ne les connaissez pas, au moins une fois pour voir, faites un clic droit sur

un point d'arrêt et jouez un peu avec les options, vous comprendrez alors comment vous auriez pu gagner des minutes ou des heures précieuses si vous aviez connu cette astuce plus tôt !

Mais ce n'est pas des points d'arrêt conditionnels que je voulais vous parler aujourd'hui mais d'une autre astuce encore moins connue / utilisée : les points d'arrêt par code.

En effet, le debugger de Visual Studio peut, en partie, être contrôlé par le code lui-même en utilisant la classe Debugger de l'espace de noms System.Diagnostics.
Les méthodes statiques de cette classe permettent par exemple de savoir si le debugger est lancé ou non, voire de lancer s'il n'est pas actif. La méthode Break() quant à elle permet simplement de faire un point d'arrêt et c'est elle qui nous intéresse ici.

Plutôt que d'attendre qu'une exception soit levée, de revenir dans le code, de placer un point d'arrêt et de relancer l'application en espérant que "ça plante" de la même façon, il est plus facile de prévoir d'emblée le point d'arrêt là où il existe un risque d'y avoir un problème, notamment en phase de mise au point d'un code. Un simple Debugger.Break(), dès qu'il sera rencontré lors de l'exécution, aura le même effet qu'un point d'arrêt inconditionnel placé dans Visual Studio. Bien entendu, le break peut être programmer selon un test (valeur non valide d'une propriété par exemple). Dans un tel cas dès que l'application rencontrera le break elle déclenchera le passage en mode debug sur le "point d'arrêt" ainsi défini. Le développeur peut se dégager l'esprit pour d'autres tâches de test, dès que le break sera rencontré VS passera en debug immédiatement sans risque de "louper" le problème ou de le voir trop tard et de ne plus avoir accès à certaines variables.
Un petit exemple :

```
class Program
{
  static void Main(string[] args)
  {
     var list = new List<Book>
     {   new Book {Title = "Livre 1", Year = 1981},
         new Book {Title = "Livre 2", Year = 2007},
         new Book {Title = "Livre 3", Year = 2040} };

foreach (var book in list) { Console.WriteLine(book.Title+" "+book.Year); }
}
class Book
{
  private int year;
  public string Title { get; set; }
  public int Year
```

```
 { get { return year; }
   set {
         if (value > 1980 && value < DateTime.Now.Year) year = value;
         else Debugger.Break();
    // throw new Exception("L'année " + value + " n'est pas autorisée");
       }
  }
}
```

Lors de l'initialisation de la collection "list" dans le Main(), l'année du troisième livre (2040) déclenchera le break. On pourra alors directement inspecter le code et savoir pourquoi ce "logiciel" plante dès son lancement... On voit qu'ici j'ai mis en commentaire l'exception qui sera lancée par la version "finale" du code. A sa place j'ai introduit l'appel à Break(). Rien à surveiller. Si le problème vient de là (ce qui est le cas ici) VS passera tout seul en debug...

Améliorer le debug sous VS avec les proxy de classes

Visual Studio est certainelement l'IDE le plus complet qu'on puisse rêver et au-delà de tout ce qu'il offre "*out of the box*" il est possible de lui ajouter de nombreux add-ins (gratuits ou payants) permettant de l'adapter encore plus à ses propres besoins. Ainsi vous connaissez certainement les "gros" add-ins comme Resharper dont j'ai parlé ici quelque fois ou GhostDoc qui écrit tout seul la doc des classes. Vous connaissez peut-être les add-ins de debogage permettant d'ajouter vos propres visualisateurs personnalisés pour le debug. Mais vous êtes certainement moins nombreux à connaître les proxy de classes pour le debug (**Debugger Type Proxy**).
A quoi cela sert-il ?

Tout d'abord cela n'a d'intérêt qu'en mode debug. Vous savez que lorsque vous placez un point d'arrêt dans votre code le debugger de VS vous permet d'inspecter le contenu des variables. C'est la base même d'un bon debugger.

La classe à deboguer

Supposons la classe Company suivante décrivant une société stockée dans notre application. On y trouve trois propriétés, le nom de la société Company, l'ID de la société dans la base de données et la date de dernière facturation LastInvoice :

```
public class Customer
        {
            private string company;
            public string Company
            {
                get { return company; }
                set { company = value; }
```

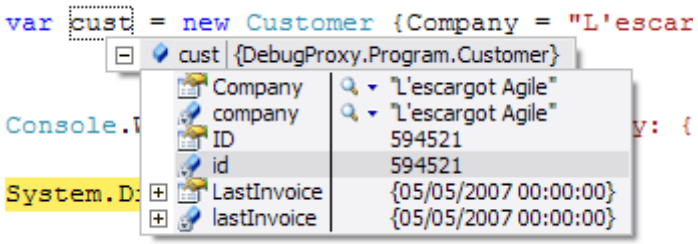
```
        }

        private int id;
        public int ID
        {
            get { return id; }
            set { id = value; }
        }

        private DateTime lastInvoice;
        public DateTime LastInvoice
        {
            get { return lastInvoice; }
            set { lastInvoice = value; }
        }
    }
```

Le debug "de base"

Supposons maintenant que nous placions un point d'arrêt dans notre application pour examiner le contenu d'une variable de type Customer, voici que nous verrons :



Le debugger affiche toutes les valeurs connues pour cette classe, les propriétés publiques comme les champs privés. Il n'y a que 3 propriétés et 3 champs dans notre classe, imaginez le fatras lorsqu'on affiche le contenu d'une instance créée depuis une classe plus riche ! Surtout qu'ici, pour tester notre application, ce dont nous avons besoin immédiatement ce sont juste deux informations claires : le nom et l'ID de la société et surtout le nombre de jours écoulés depuis la dernière facture. Retrouver l'info parmi toutes celles affichées, voire faire un calcul à la main pour celle qui nous manque, c'est transformer une session de debug qui s'annonçait plutôt bien en un véritable parcours du combattant chez les forces spéciales !

Hélas, dans la classe Customer ces informations sont soit **éparpillées** (nom de société et ID) soit**inexistantes** (ancienneté en jours de la dernière facture).

Il existe bien entendu la possibilité de créer un visualisateur personnalisé pour la classe Customer et de l'installer dans les plug-ins de Visual Studio. C'est une démarche simple mais elle réclame de créer une DLL et de la déployer sur la machine. Cette solution est parfaite en de nombreuses occasions et elle possède de gros avantages (réutilisation, facilement distribuable à plusieurs développeurs, possibilité de créer un "fiche" complète pour afficher l'information etc).
Mais il existe une autre voie, **encore plus simple et plus directe** : les **proxy de types pour le debugger**.

Un proxy de type pour simplifier

A ce stade du debug de notre application nous avons vraiment besoin du nombre de jours écoulés depuis la dernière facture et d'un moyen simple de voir immédiatement l'identification de la société. Nous ne voulons pas créer un visualisateur personnaliser, mais nous voulons tout de même une visualisation personnalisée...

Regardons le code de la classe suivante :

```
public class CustomerProxy
        {
            private Customer cust;

            public CustomerProxy(Customer cust)
            {
                this.cust = cust;
            }

            public string FullID
            {
                get { return cust.Company + " (" + cust.ID + ")"; }
            }

            public int DaysSinceLastInvoice
            {
                get { return (int)
                    (DateTime.Now - cust.LastInvoice).TotalDays; }
            }
        }
```

La classe CustomerProxy est très (très) simple : elle possède un constructeur prenant en paramètre une instance de la classe Customer puis elle expose deux propriétés en read only

: FullID qui retourne le nom de la société suivi de son ID entre parenthèses, et le nombre de jours écoulés depuis la dernière facture.
Nota: Ce code de démo ne contient aucun test... dans la réalité vous y ajouterez des tests sur null pour éviter les exceptions si l'instance passée est nulle, bien entendu.

Vous allez me dire, c'est très joli, ça fait une classe de plus dans mon code, et comment je m'en sers ? Je ne vais pas modifier tout mon code pour créer des instances de cette classe cela serait délirant !

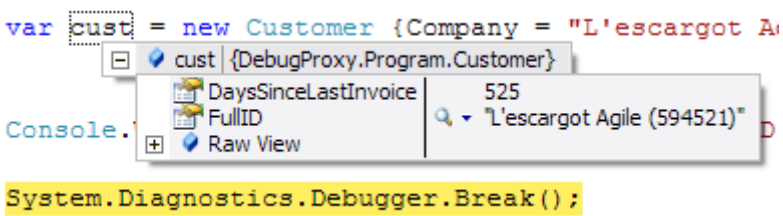
Vous avez parfaitement raison, nous n'allons pas créer d'instance de cette classe, nous n'allons pas même modifier le code de l'application en cours de debug (ce qui serait une grave erreur... modifier ce qu'on test fait perdre tout intérêt au test). Non, nous allons simplement indiquer au framework .NET qu'il utilise notre proxy lorsque VS passe en debug... Un attribut à ajouter à la classe originale Customer, c'est tout :

```
#if (DEBUG)
        [System.Diagnostics.DebuggerTypeProxy(typeof(CustomerProxy))]
#endif
        public class Customer
        {
            //...
        }
```

Vous remarquerez que pour faire plus "propre" j'ai entouré la déclaration de l'attribut dans un #if DEBUG, cela n'est absolument **pas obligatoire**, j'ai fait la même chose autour du code de la classe proxy. De ce fait ce code ne sera pas introduit dans l'application en mode Release. Je vous conseille cette approche malgré tout.
Et c'est fini !

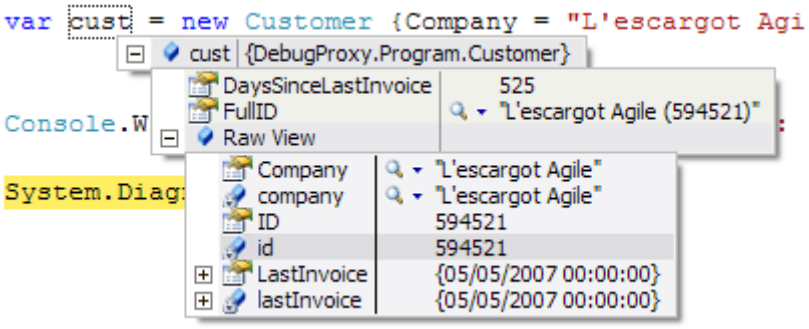
Le proxy en marche

Désormais, lorsque nous sommes en debug que nous voulons voir le contenu d'une instance de la classeCustomer voici ce que Visual Studio nous affiche :



Vous remarquez immédiatement que le contenu de l'instance de la classe Customer n'est plus affiché directement mais en place et lieu nous voyons les deux seules propriétés

"*artificielles*" qui nous intéressent : le nom de société avec son ID, et le nombre de jours écoulés depuis la dernière facture. Fantastique non ? !
"Et si je veux voir le contenu de Customer malgré tout ?" ... Je m'attendais à cette question... Regardez sur l'image ci-dessus, oui, là, le petit "plus" dans un carré, "Raw View"... Cliquez sur le plus et vous aurez accès au même affichage de l'instance de Customer qu'en début d'article (sans le proxy) :



Conclusion

Si ça ce n'est pas de la productivité et de la customisation aux petits oignons alors je suis à court d'arguments !

Contourner le problème de l'appel d'une méthode virtuelle dans un constructeur

Ce problème est source de bogues bien sournois. J'ai déjà eu l'occasion de vous en parler dans un billet cet été (Appel d'un membre virtuel dans le constructeur ou "quand C# devient vicieux". A lire absolument...), la conclusion était qu'il ne faut tout simplement pas utiliser cette construction dangereuse. Je proposais alors de déplacer toutes les initialisations dans le constructeur de la classe parent, mais bien entendu cela n'est pas applicable tout le temps (sinon à quoi servirait l'héritage).
Dès lors comment proposer une méthode fiable et systématique pour contourner le problème proprement ?

Rappel

Je renvoie le lecteur au billet que j'évoque en introduction pour éviter de me répéter, le problème posé y est clairement démontré. Pour les paresseux du clic, voici en gros le résumé de la situation : L'appel d'une méthode virtuelle dans le constructeur d'une classe est fortement déconseillé. La raison : lorsque qu'une instance d'une classe dérivée est créée elle commence par appeler le constructeur de son parent (et ainsi de suite en cascade remontante). Si ce constructeur parent appelle une méthode virtuelle overridée dans la

classe enfant, le problème est que l'instance enfant elle-même n'est pas encore initialisée, l'initialisation se trouvant encore dans le code du constructeur parent. Et cela, comme vous pouvez l'imaginer, ça sent le bug !

Une première solution

La seule et unique solution propre est donc de s'interdire d'appeler des méthodes virtuelles dans un constructeur. Et je serai même plus extrémiste : il faut s'interdire d'appeler toute méthode dans le constructeur d'une classe, tout simplement parce qu'une méthode non virtuelle, allez savoir, peut, au gré des changements d'un code, devenir virtuelle un jour. Ce n'est pas quelque chose de souhaitable d'un pur point de vue méthodologique, mais nous savons tous qu'entre la théorie et la pratique il y a un monde...

Tout cela est bien joli mais s'il y a des appels à des méthodes c'est qu'elles servent à quelque chose, s'en passer totalement semble pour le coup tellement extrême qu'on se demande si ça vaut encore le coup de développer ! Bien entendu il existe une façon de contourner le problème : il suffit de créer une méthode publique "Init()" qui elle peut faire ce qu'elle veut. Charge à celui qui créé l'instance d'appeler dans la foulée cette dernière pour compléter l'initialisation de l'objet.
Le code suivant montre une telle construction :

```
// Classe Parent
   public class Parent2
   {
       public Parent2(int valeur)
       {
           // MethodeVirtuelle();
       }

       public virtual void Init()
       {
           MethodeVirtuelle();
       }

       public virtual void MethodeVirtuelle()
       { }
   }

   // Classe dérivée
   public class Enfant2 : Parent2
   {
       private int val;
```

```
        public Enfant2(int valeur)
            : base(valeur)
        {
            val = valeur;
        }

        public override void MethodeVirtuelle()
        {
            Console.WriteLine("Classe Enfant2. champ val = " + val);
        }
    }
```

La méthode virtuelle est appelée dans Init() et le constructeur de la classe de base n'appelle plus aucune méthode.
C'est bien. Mais cela complique un peu l'utilisation des classes. En effet, désormais, pour créer une instance de la classe Enfant2 il faut procéder comme suit :

```
// Méthode 2 : avec init séparé
var enfant2 = new Enfant2(10);
enfant2.Init(); // affichera 10
```

Et là, même si nous avons réglé un problème de conception essentiel, côté pratique nous sommes loin du compte ! Le pire c'est bien entendu que nous obligeons les utilisateurs de la classe Enfant2 à "penser à appeler Init()". Ce n'est pas tant l'appel à Init() qui est gênant que le fait qu'il faut penser à le faire ... Et nous savons tous que plus il y a de détails de ce genre à se souvenir pour faire marcher un code, plus le risque de bug augmente.
Conceptuellement, c'est propre, au niveau design c'est à fuir...

Faut-il donc choisir entre peste et choléra sans aucun espoir de se sortir de cette triste alternative ? Non. Nous pouvons faire un peu mieux et rendre tout cela transparent notamment en transférant à la classe enfant la responsabilité de s'initialiser correctement sans que l'utilisateur de cette classe ne soit obligé de penser à quoi que ce soit.

La méthode de la Factory

Il faut absolument utiliser la méthode de l'init séparé, cela est incontournable. Mais il faut tout aussi fermement éviter de rendre l'utilisation de la classe source de bugs. Voici nos contraintes, il va falloir faire avec.

La solution consiste à modifier légèrement l'approche. Nous allons fournir une méthode de classe (méthode statique) permettant de créer des instances de la classe Enfant2, charge à

cette méthode appartenant à Enfant2 de faire l'intialisation correctement. Et pour éviter toute "bavure" nous allons cacher le constructeur de Enfant2. Dès lors nous aurons mis en place une Factory (très simple) capable de fabriquer des instances de Enfant2 correctement initialisées, en une seule opération et dans le respect du non appel des méthodes virtuelles dans les constructeurs... ouf !

C'est cette solution que montre le code qui suit (Parent3 et Enfant3 étant les nouvelles classes) :

```
// Classe Parent
public class Parent3
{
    public Parent3(int valeur)
    {
        // MethodeVirtuelle();
    }

    public virtual void Init()
    {
        MethodeVirtuelle();
    }

    public virtual void MethodeVirtuelle()
    { }
}

// Classe dérivée
public class Enfant3 : Parent3
{
    private int val;

    public static Enfant3 CreerInstance(int valeur)
    {
        var enfant3 = new Enfant3(valeur);
        enfant3.Init();
        return enfant3;
    }

    protected Enfant3(int valeur)
        : base(valeur)
    {
        val = valeur;
```

```
        }

        public override void MethodeVirtuelle()
        {
            Console.WriteLine("Classe Enfant3. champ val = " + val);
        }
    }
```

La création d'une instance de Enfant3 s'effectue dès lors comme suit :

```
var enfant3 = Enfant3.CreerInstance(10);
```

C'est simple, sans risque d'erreur (impossible de créer une instance autrement), et nous respectons l'interdiction des appels virtuels dans le constructeur sans nous priver des méthodes virtuelles lors de l'initialisation d'un objet. De plus la responsabilité de la totalité de l'action est transférée à la classe enfant ce qui centralise toute la connaissance de cette dernière en un seul point.

Dans une grosse librairie de code on peut se permettre de déconnecter la Factory des classes en proposant directement une ou plusieurs abstraites qui sont les seuls points d'accès pour créer des instances. Toutefois je vous conseille de laisser malgré tout les Factory "locales" dans chaque classe. Cela évite d'éparpiller le code et si un jour une classe enfant est modifiée au point que son initialisation le soit aussi, il n'y aura pas à penser à faire des vérifications dans le code de la Factory séparée. De fait une Factory centrale ne peut être vue que comme un moyen de regrouper les Factories locales, sans pour autant se passer de ces dernières ou en modifier le rôle.

Conclusion

Peut-on aller encore plus loin ? Peut-on procéder d'une autre façon pour satisfaire toutes les exigences de la situation ? Je ne doute pas qu'une autre voie puisse exister, pourquoi pas plus élégante. Encore faut-il la découvrir. C'est comme en montagne, une fois qu'une voie a été découverte pour atteindre le sommet plus facilement ça semble une évidence, mais combien ont dû renoncer au sommet avant que le découvreur de la fameuse voie ne trace le chemin ?

Saurez-vous être ce premier de cordée génial et découvrir une voie alternative à la solution de la Factory ?

Simplifier les gestionnaires d'événement grâce aux expressions Lambda

Les expressions Lambda ont été introduites dans C# 3.0. Utilisées correctement, tout comme LINQ to Object, elles permettent une grande simplification du code entraînant dans un cercle vertueux une meilleure lisibilité de ce dernier favorisant maintenabilité et fiabilité de ce même code. Il n'y a donc aucune raison de rester "frileux" vis à vis de cette nouveauté syntaxique comme encore trop de développeurs que je rencontre dans mes formations ou ailleurs.

Pour démontrer cette souplesse, voici un petit exemple qui valide un document XML en fonction d'un schéma.

La méthode Validate() de la classe XDocument attend en paramètre un delegate de typeValidationEventHandler. Dans un code très court il n'est pas forcément judicieux de déclarer une méthode juste pour passer son nom en paramètre à Validate(). Les "*sous procédures*" ou procédures imbriquées de Pascal n'existent pas en C# et l'obligation de déclarer à chaque fois des méthodes private pour décomposer la moindre action est une lourdeur syntaxique de ce langage dont on aimerait se passer parfois (une méthode même private est visible par toutes les autres méthodes de la classe ce qui n'est pas toujours souhaitable. Seules les méthodes imbriquées de Pascal sont des "méthodes privées de méthodes").
En fin de billet vous trouverez d'ailleurs le lien vers un autre billet dans lequel j'explique comment utiliser les expressions Lambda en les nommant pour retrouver la souplesse des procédures imbriquées. Même si "l'astuce" présentée ici peut y ressembler dans l'esprit, dans la pratique les choses sont assez différentes puisque nous n'utiliserons pas une expression nommée.

Revenons à l'exemple, il s'agit de valider un document XML à partir d'un schéma. Et de le faire de la façon la plus simple possible, c'est à dire en évitant l'écriture d'un delegate, donc en utilisant directement une expression Lambda en paramètre de Validate().
Supposons que nous ayons déjà le schéma (variable schema) et le document XML (variable doc), l'appel àValidate pourra donc s'écrire :

```
doc.Validate(schema,(obj, e) => errors += e.Message +
Environment.NewLine);
```

"errors" est déclarée comme une chaîne de caractères. A chaque éventuelle erreur détectée par la validation l'expression concatène le message d'erreur à cette dernière. En fin de validation il suffit d'affichererrors pour avoir la liste des erreurs. Mais cela n'est qu'un exemple d'utilisation de l'expression Lambda. Ce qui compte c'est bien entendu de

comprendre l'utilisation de cette dernière en place et lieu d'un delegate de tout type, ici ValidationEventHandler d'où les paramètres (obj,e) puisque ce delegate est déclaré de cette façon (un objet et un argument spécifique à cet handler).
C'est simple, nul besoin de déclarer un gestionnaire d'événement donc une méthode avec un nom et une visibilité. On évite que cette méthode soit réutilisée dans le code de la classe considérée (si elle existe on est tenté de la réutiliser mais sa stratégie d'écriture n'est pas forcément adaptée à une telle utilisation d'où possible bug), etc. Que des avantages donc.

Pour plus d'information vous pouvez lire mon article sur les nouveautés syntaxiques de C# 3.0 ainsi que mon billet montrant comment retrouver en C# le bénéfice des procédures imbriquées de Pascal grâce aux expressions Lambda.

Astuce : recenser rapidement l'utilisation d'une classe dans une grosse solution

Comment recenser toutes les utilisations d'une classe précise dans une grosse solution pleine de projets ?

Certains proposeront d'utiliser la fonction "*find usage*" de Resharper. Certes mais tout le monde n'a pas cet add-in. Et même si vous l'avez, vous n'êtes pas sûr que là où vous aurez à intervenir il sera toujours là...

D'autres proposeront le *Ctrl-F*. C'est pas mal mais ça trouvera aussi les bouts de texte qui citent la classe ou qui contiennent le nom de cette dernière. Les plus torturés proposeront alors d'utiliser une expression régulière. Techniquement c'est mieux mais concevoir une belle ER qui fasse bien le boulot, tout le monde ne sait pas forcément faire.

Non, moi je vous parle d'un **moyen ultra simple et absolument sûr** de trouver toutes les utilisations d'une classe dans des tas projets en quelques secondes sans trop se fatiguer.

... Vous séchez ? Alors voici la réponse : **l'attribut Obsolete**.

C'est tout bête, c'est une utilisation un peu détournée de la chose il faut l'avouer, mais il suffit d'ajouter devant la définition de la classe en question l'attribut

```
[Obsolete("blabla")]
public class TheClassARepérer ...
```

et l'affaire est jouée. Faites un Rebuild de la solution et dans les warnings vous aurez la liste de tous les endroits où la classe est utilisée. Un double-clic vous amènera directement dans le code en question.

Quand l'opération est terminée, il suffit de supprimer l'attribut. La manip est ultra légère, peu de chance d'introduire un bug, et si on onblit l'attribut ça se verra tout de suite dans les warnings.

Malin non ?

Un générateur de code C#/VB à partir d'un schéma XSD

Générer du code C# depuis un schéma XSD est un besoin de plus en plus fréquent, XML étant désormais omniprésent. On trouve dans le Framework l'outil "xsd.exe" qui permet une telle génération toutefois elle reste assez basique. C'est pour cela qu'on trouve aussi des outils tiers qui tentent, chacun à leur façon, d'améliorer l'ordinaire.

XSD2Code est un de ces outils tiers. Codé par Pascal Cabanel et releasé sur CodePlex, c'est sous la forme d'un add-in Visual Studio que se présente l'outil (le code source contient aussi une version Console).
Son originalité se trouve bien entendu dans les options de génération qui prennent en compteINotifyPropertyChanged ainsi que la création de List<T> ou ObservableCollection<T>. D'autres options comme la possibilité de générer le code pour C# ou VB.NET, le support des types nullable, etc, en font une alternative plutôt séduisante à "xsd.exe". La prise en compte des modifications de propriété (et la génération automatique du code correspondant) autorise par exemple le DataBinding sous WPF ou Silverlight...
Comme Pascal suit ce blog il pourra certainement m'éclairer sur le pourquoi d'un petit dysfonctionnement (j'ai aussi laissé un message dans le bug tracker du projet): lorsque l'add-in est installé, et après l'avoir activé dans le manager d'add-in je ne vois hélas pas l'entrée de menu apparaître sur le clic-droit dans l'explorateur de solution (sur un fichier xsd bien sûr). Heureusement, le code source étant fourni sur CodePlex et le projet intégrant une version console de l'outil j'ai pu tester la génération de code C# en ligne de commande. Il est vrai que l'intégration de l'outil dans l'IDE est un sacré plus que je suis triste ne n'avoir pu voir en action :-(Peut-être s'agit-il d'un problème lié au fait que ma version de VS est en US alors que mon Windows est en FR ? Cela trouble peut-être la séquence qui insère la commande dans le menu contextuel ?

Un petit détail à régler donc, mais dès que j'ai des nouvelles je vous en ferai part.

Le projet XSD2Code est fourni en deux versions, code source et setup près à installer. Pascal a même créé une petite vidéo montrant l'add-in en action.

Un outil qui, même en version console, remplace avantageusement "xsd.exe" et qui a donc toutes les raisons de se trouver dans votre boîte à outils !

Utiliser des clés composées dans les dictionnaires

Les dictionnaires sont des listes spécialisées permettant de relier deux objets, le premier étant considéré comme la clé, le second comme la valeur. Une fois clé et valeur associées au sein d'une entrée du dictionnaire ce dernier est capable de retourner rapidement la valeur de toute clé. Les dictionnaires peuvent être utilisés en de nombreuses circonstances, comme la conception de caches de données par exemple.

Un dictionnaire se créée à partir de la classe générique Dictionnary<Key,Value>. Comme on le remarque si la clé peut être de tout type elle reste monolithique, pas de clés composées donc, et encore moins de classes
telles Dictionnary<Key1,Key2,Value> ouDictionnary<Key1,Key2,Key3,Value> etc...

Or, il est assez fréquent qu'une clé soit composée (*multi-part key* ou *composed key*).

Comment utiliser les dictionnaires génériques dans un tel cas ?

La réponse est simple : ne confondons pas une seule clé et un seul objet objet clé ! En effet, si le dictionnaire n'accèpte qu'un seul objet pour la partie clé, *rien n'interdit que cet objet soit aussi complexe qu'on le désire*... Il peut donc s'agir d'instances d'une classe créée pour l'occasion, classe capable de maintenir une clé composée.
Vous allez me dire que ce n'est pas bien compliqué, et vous n'aurez qu'à moitié raison...

Créer une classe qui contient 2 propriétés n'est effectivement pas vraiment ardu. Prenons un exemple simple d'un dictionnaire associant des ressources à des utilisateurs. Imaginons que l'utilisateur soit repéré grâce à deux informations, son nom et une clé numérique (le hash d'un password par ex) et imaginons, pour simplifier, que la ressource associée soit une simple chaîne de caractères.

La classe qui jouera le rôle de clé du dictionnaire peut ainsi s'écrire en une poignée de lignes :

```
1:  public class LaClé
2:  {
3:    public string Name { get; set; }
4:    public int PassKey {get; set; }
5:  }
```

Oui, c'est vraiment simple. Mais il y a un hic !

En effet, cette classe ne gère pas l'égalité, elle n'est pas "comparable". De base, écrite comme ci-dessus, elle ne peut pas servir de clé à un dictionnaire...

Pour être utilisable dans un tel contexte il faut ajouter du code afin de **gérer la comparaison entre deux instances**. Il existe plusieurs façons de faire, l'une de celle que je préfère est l'implémentation de l'interface générique IEquatable<T>. On pourrait par exemple choisir une autre voie en implémentant dans la classe clé une autre classe implémentant IEqualityComparer<T>.

Toutefois dans un tel cas il faudrait préciser au dictionnaire lors de sa création qu'il lui faut utiliser ce comparateur là bien précis, cela est très pratique si on veut changer de comparateur à la volée, mais c'est un cas assez rare. En revanche si demain l'implémentation changeait dans notre logiciel et qu'une autre structure soit substituée au dictionnaire il y aurait de gros risque que l'application ne marche plus: les objets clés ne seraient toujours pas comparables deux à deux "automatiquement".

L'implémentation d'une classe utilisant IEqualityComparer<T> est ainsi une solution partielle en ce sens qu'elle réclame une action volontaire pour être active. De plus cette solution se limite aux cas où un comparateur de valeur peut être indiqué.

C'est pour cela que je vous conseille fortement d'implémenter directement dans la classe "clé" l'interfaceIEquatable<T>. Quelles que soient les utilisations de la classe dans votre application l'égalité fonctionnera toujours sans avoir à vous soucier de quoi que ce soit, et encore moins, et surtout, des éventuelles évolutions du code.
Comme par enchantement l'excellent Resharper (add-in pour VS totalement indispensable) sait générer automatiquement tout le code nécessaire, je n'ai donc pas eu grand chose à saisir pour le code final... Ceux qui ne disposent pas de cet outil merveilleux pourront bien entendu s'inspirer de l'implémentation proposée pour leur propre code.

Le code de notre classe "clé" se transforme ainsi en quelque chose d'un peu plus volumineux mais de totalement fonctionnel :

```
1:  public class ComposedKey : IEquatable<ComposedKey>
2:          {
3:              private string name;
4:              public string Name
5:              {
6:                  get { return name; }
7:                  set { name = value; }
8:              }
```

```
 9:
10:             private int passKey;
11:             public int PassKey
12:             {
13:                 get { return passKey; }
14:                 set { passKey = value; }
15:             }
16:
17:             public ComposedKey(string name, int passKey)
18:             {
19:                 this.name = name;
20:                 this.passKey = passKey;
21:             }
22:
23:             public override string ToString()
24:             {
25:                 return name + " " + passKey;
26:             }
27:
28:             public bool Equals(ComposedKey obj)
29:             {
30:                 if (ReferenceEquals(null, obj)) return false;
31:                 if (ReferenceEquals(this, obj)) return true;
32:                 return Equals(obj.name, name) && obj.passKey ==
passKey;
33:             }
34:
35:             public override bool Equals(object obj)
36:             {
37:                 if (ReferenceEquals(null, obj)) return false;
38:                 if (ReferenceEquals(this, obj)) return true;
39:                 if (obj.GetType() != typeof (ComposedKey)) return
false;
40:                 return Equals((ComposedKey) obj);
41:             }
42:
43:             public override int GetHashCode()
44:             {
45:                 unchecked
46:                 {
47:                     return ((name != null ? name.GetHashCode() :
0)*397) ^ passKey;
```

```
48:                 }
49:             }
50:
51:             public static bool operator ==(ComposedKey left,
ComposedKey right)
52:             {
53:                 return Equals(left, right);
54:             }
55:
56:             public static bool operator !=(ComposedKey left,
ComposedKey right)
57:             {
58:                 return !Equals(left, right);
59:             }
60:         }
```

Désormais il devient possible d'utiliser des instances de la classe **ComposedKey** comme clé d'un dictionnaire générique.
Dans un premier temps testons le comportement de l'égalité :

```
1:  // Test of IEquatable in ComposedKey
2:  var k1 = new ComposedKey("Olivier", 589);
3:  var k2 = new ComposedKey("Bill", 9744);
4:  var k3 = new ComposedKey("Olivier", 589);
5:
6:  Console.WriteLine("{0} =? {1} : {2}",k1,k2,(k1==k2));
7:  Console.WriteLine("{0} =? {1} : {2}",k1,k3,(k1==k3));
8:  Console.WriteLine("{0} =? {1} : {2}",k2,k1,(k2==k1));
9:  Console.WriteLine("{0} =? {1} : {2}",k2,k2,(k2==k2));
10: Console.WriteLine("{0} =? {1} : {2}",k2,k3,(k2==k3));
```

Ce code produira le résultat suivant à la console :

Olivier 589 =? Bill 9744 : False
Olivier 589 =? Olivier 589 : True
Bill 9744 =? Olivier 589 : False
Bill 9744 =? Bill 9744 : True
Bill 9744 =? Olivier 589 : False

Ces résultats sont conformes à notre attente. Nous pouvons dès lors utiliser la classe au sein d'un dictionnaire comme le montre le code suivant :

```
1:  // Build a dictionnary using the composed key
2:  var dict = new Dictionary<ComposedKey, string>()
3:                  {
4:                      {new ComposedKey("Olivier",145), "resource A"},
5:                      {new ComposedKey("Yoda", 854), "resource B"},
6:                      {new ComposedKey("Valérie", 9845), "resource C"},
7:                      {new ComposedKey("Obiwan", 326), "resource D"},
8:                  };
9:
10:  // Find associated resources by key
11:
12:  var fk1 = new ComposedKey("Yoda", 854);
13:  var s = dict.ContainsKey(fk1) ? dict[fk1] : "No Resource Found";
14:  // must return 'resource B'
15:  Console.WriteLine("Key '{0}' is associated with resource
'{1}'",fk1,s);
16:
17:  var fk2 = new ComposedKey("Yoda", 999);
18:  var s2 = dict.ContainsKey(fk2) ? dict[fk2] : "No Resource Found";
19:  // must return 'No Resource Found'
20:  Console.WriteLine("Key '{0}' is associated with resource '{1}'",
fk2, s2);
```

Code qui produira la sortie suivante :

Key 'Yoda 854' is associated with resource 'resource B'
Key 'Yoda 999' is associated with resource 'No Resource Found'
Et voilà ...

Rien de tout cela est compliqué mais comme on peut le voir il y a toujours une distance de la coupe aux lèvres, et couvrir cette distance c'est justement tout le savoir-faire du développeur !

Le retour des sous-procédures avec les expressions Lambda...

La syntaxe de C# et son orientation "tout objet" ont définitivement tourné la page de la programmation procédurale. Ce n'est certes pas un mal, bien au contraire, mais au passage nous avons perdu une petite facilité de langage tel que Pascal qui autorisaient la déclaration de procédures à l'intérieur de procédures. Le manque n'est pas cruel mais tout de même... Il semble souvent bien lourd et assez artificiel d'être obligé de créer une méthode private ou

internal juste pour rendre un service à une seule méthode. De plus le morceau de code ainsi transformé en méthode, même private ou internal, ne sera encapsulé qu'au niveau de la classe et non de la méthode intéressée, d'où le risque de l'utiliser ailleurs (ce qui change sa statégie d"écriture). Le code sera aussi plus lourd en raison de la nécessité de passer en paramètre tout ce qui sera nécessaire à l'exécution de cette "sous méthode" alors qu'une procédure imbriquée peut référencer les variables de la procédure qui l'abrite.

Bref, l'affaire ne mérite certainement pas de grandes théories, mais il faut avouer que de temps en temps on aimerait bien pouvoir déclarer une petite méthode à l'intérieur d'une autre. Il s'agit là d'appliquer la logique des classes elle-mêmes : il est possible de déclarer une classe dans une autre lorsqu'elle ne sert exclusivement qu'à la première. Pourquoi ne pas retrouver cette possibilité au niveau des méthodes d'une classe ?

Les procédures imbriquées n'existent pas en C#. Cela vous semble une certitude. Avec C# 3.0 ce n'est plus aussi certain...

Les expressions Lambda utilisées comme des procédures imbriquées

Je n'entrerai pas ici dans le détail de la syntaxe des expressions Lambda que j'ai déjà présenté dans un long article sur les nouveautés de C# 3.0, article auquel je renvoie le lecteur s'il en ressent le besoin (Les nouveautés syntaxiques de C# 3.0 et Présentation des différentes facettes de LINQ)
Les procédures imbriquées ne sont rien d'autres que des procédures "normales" mais déclarées à l'intérieur d'autres procédures. En Pascal cela ne peut se faire qu'entre l'entête de la méthode principal et le corps de celle-ci :

```
Procedure blabla
 Procedure imbrique begin ... end;
begin // blabla
...
end; // blabla
```

Avec les expressions Lambda de C# 3.0 on retrouve une possibilité sensiblement identique avec plus de souplesse encore puisque la "sous procédure" peut être déclarée n'importe où dans le corps de la "procédure principale".

Exemple

```
static void Main(string[] args)
{
   // une "fonction" imbriquée (teste si un nombre est impair)
   Func<int, bool> isOdd = i => (i & 1) == 1;
   // une "procédure" imbriquée (formate et écrit un int à la console)
   Action<int> format = i => Console.WriteLine(i.ToString("000"));
```

```
  Console.WriteLine(isOdd(25));
  Console.WriteLine(isOdd(24));
    format(25);
    format(258);
    format(5);
}
```

La sortie sera :

```
True
False
025
258
005
```

Conclusion

La possibilité de déclarer des "sous procédures" est bien pratique, cela permet en général d'éviter les répétitions dans le corps d'une méthode, donc de diminuer le risque de bug et d'améliorer sensiblement la lecture du code. C# ne supportait pas cette possibilité syntaxique, mais en utilisant les expressions Lambda nous retrouvons la même fonctionnalité...

Appel d'un membre virtuel dans le constructeur ou "quand C# devient vicieux". A lire absolument...

En maintenant un code C# d'un client mon ami Resharper me dit d'un appel à une méthode dans le constructeur d'une classe "*virtual member call in constructor*". J'ai tellement pris le pli avec ce problème que je ne m'en souci plus guère dans mon propre code, j'évite soigneusement la situation...
Mais vous ? Avez-vous conscience de la gravité de ce problème ?

Sans Resharper il faut passer volontairement une analyse du code pour voir apparaître le message CA2214 "*xxx contient une chaîne d'appel aboutissant à un appel vers une méthode virtuelle définie par la classe.*". D'une part je doute fort que tout le monde comprenne du premier coup ce message ésotérique mais le pire c'est que je sais par expérience que la grande majorité des développeurs n'utilisent, hélas, que très rarement cette fonction... Et à la compilation du projet, aucune erreur, aucun avertissement ne sont indiqués !

Vous allez me dire "*ça ne doit pas être bien grave si le compilateur ne dit rien et que seul un FxCop relève un simple avertissement*". Je m'attendais à ce que vous me disiez cela... Et je

vais vous prouver dans quelques lignes que cette remarque candide est la porte ouverte à de gros ennuis...

Le grave problème des appels aux méthodes virtuelles dans les constructeurs

Ce problème est "grave" à plus d'un titre. Tout d'abord techniquement, comme le code qui suit va vous le montrer, votre programme aura un comportement que vous n'avez pas prévu et qui mène à des **bogues sournois**. Cela est en soi suffisant pour qualifier le problème de "grave".

Ensuite, moins on a conscience d'un problème potentiel et plus il est grave, par nature. Comme très peu de développeurs ont conscience du fait que ce comportement bien particulier de C# est une source potentielle d'énormes problèmes, sa gravité augmente d'autant.
Pour terminer et agraver la situation, le compilateur ne dit rien et seule une analyse du code (ou l'utilisation d'un outil comme Resharper qui l'indique visuellement dans l'éditeur de code) peut permettre de prendre connaissance du problème.

La chaîne ne s'arrête pas là (*tout ce qui peut aller mal ira encore pire* - Murphy), puisque même en passant l'analyseur de code le message sera noyé dans des dizaines, voire centaines d'avertissements et que, cerise sur le gateau, même si on prend la peine de lire l'avertissement, son intitulé est totalement nébuleux !

La preuve par le code

Maintenant que je vous ai bien alarmé, je vais enfoncé le clou par quelques lignes de code (qu'il est méchant) !

```
class Program
{
      static void Main(string[] args)
      {
         var derivé = new Derived();
      }
}

public class Base
{
  public Base()
  { Init(); }
    public virtual void Init()
   { Console.WriteLine("Base.Init"); }
}
```

```
public class Derived : Base
{
  private string s = "Non initialisée!";
  public Derived()
 { s = "variable initialisée"; }
   public override void Init()
  { Console.WriteLine("Derived.Init. var s = "+s); }
}
```

La question à deux eurocents est la suivante : Au lancement de la classe Program et de son Main, qu'est-ce qui va s'afficher à la console ?

La réponse est "Derived.Init. var s = Non initiliasée!".
L'action au ralenti avec panoramique 3D façon Matrix : Dans Main nous instancions la classe Derived. Cette classe est une spécialisation de la classe Base. Dans cette dernière il y a un constructeur qui appelle la méthode Init. Cette méthode est virtuelle et elle est surchargée dans la classe Derived.

Lorsque nous instancions Derived, de façon automatique le constructeur de Base se déclenche, ce qui provoque l'appel à Init. Donc à la version surchargée de Derived puisque *C# appelle toujours la méthode dérivée la plus proche du type en cours*.

D'où vient le problème ? ... Il vient du fait que le constructeur de Base, d'où provient l'appel à Init, n'est pas terminé (il le sera au retour de Init et une fois sa parenthèse de fin atteinte), du coup le constructeur de Derived n'a pas encore été appelé !

Si le code de Init ne repose sur aucune initialisation effectuée dans le constructeur de cette classe, tout va bien. Vous remarquerez d'ailleurs que le message affiché prend en compte la valeur de la variable s qui est initialisée dans sa déclaration et non pas une chaîne nulle. Ce qui prouve que les déclarations de variables initialisées sont, elles, bien exécutées, et avant le constructeur. Mais si le code de Init dépend de certaines initialisations effectuées dans le constructeur (initialisations simples comme dans l'exemple ci-dessus ou indirectes avec des appels de méthodes), alors là c'est la catastrophe : le constructeur deDerived n'a pas encore été appelé alors même que la version surchargée de Init dans Derived est exécutée par le constructeur de la classe mère !

La règle

Elle est simple : **ne jamais appeler de méthodes virtuelles dans le constructeur d'une classe** !
La règle CA2214 de l'analyseur de code :

"When a virtual method is called, the actual type that executes the method is not selected until run time. When a constructor calls a virtual method, it is possible that the constructor for the instance that invokes the method has not executed. "

"Quand une méthode virtuelle est appelée, le type actuel qui exécute la méthode n'est pas sélectionné jusqu'au runtime [ndt: c'est le principe des méthodes virtuelles, le "late binding"]. Quand un constructeur appelle une méthode virtuelle, il est possible que le constructeur de l'instance qui est invoquée n'ait pas encore été exécuté".
C'est "*possible*", ce n'est même pas sûr, donc il ne faut surtout pas écrire de code qui repose sur ce mécanisme...

L'aide de l'analyseur de code m'amuse beaucoup car dans sa section "How to fix violations" ("comment résoudre le problème"), il est dit tout simplement de ne jamais appeler de méthodes virtuelles dans les constructeurs... Avec ça débrouillez-vous !

La solution
Comme le dit laconiquement l'aide de l'analyseur : "faut pas le faire". Voilà la solution... En gros, si le cas se produit, comme dans notre exemple, la seule solution viable consiste à prendre le code de la méthodeInit et à le déplacer dans le constructeur, il est fait pour ça... La méthode Init n'existe plus bien entendu, et elle est n'est donc plus surchargée dans la classe fille.

Conclusion
J'espère que ce petit billet vous aura aidé à prendre conscience d'un problème généralement méconnu, une spécificité de C# qu'on ne retrouve ni sous C++ ni sous Delphi.

De l'intérêt d'overrider GetHashCode()

Les utilisateurs de Resharper ont la possibilité en quelques clics de générer un GetHashCode() et d'autres méthodes comme les opérateurs de comparaison pour toute classe en cours d'édition. Cela est extrêment pratique et utile à plus d'un titre. Encore faut-il avoir essayer la fonction de Resharper et s'en servir à bon escient... Mais pour les autres, rien ne vient vous rappeler **l'importance de telles fonctions**. Pourtant elles sont essentielles au bon fonctionnement de votre code !

GetHashCode()
Cette méthode est héritée de object et retourne une valeur numérique sensée être unique pour une instance. Cette unicité est toute relative et surtout sa répartition dans le champ des valeurs possibles est inconnue si vous ne surchargez pas GetHashCode() dans vos classes et structures ! Il est en effet essentiel que le code retourné soit en rapport direct avec le contenu de la classe / structure. Deux instances ayant des valeurs différentes doivent

retourner un hash code différent. Mieux, ce hash code doit être représentatif et générer le minimum de collisions...
Si vous utilsez un structure comme clé d'une Hashtable par exemple, vous risquez de rencontrer des **problèmes de performances** que vous aurez du mal à vous expliquer si vous n'avez pas conscience de ce que j'expose ici...

Je ne vous expliquerais pas ce qu'est un hash code ni une table Hashtable, mais pour résumer disons qu'il s'agit de créer des clés représentant des objets, clés qui doivent être "harmonieusement" réparties dans l'espace de la table pour éviter les collisions. Car en face des codes de hash, il y a la table qui en interne ne gère que quelques entrées réelles. S'il y a collision, elle chaîne les valeurs.

Moralité, au lieu d'avoir un accès 1->1 (une code hash correspond à une case du tableau réellement géré en mémoire) on obtient plutôt n -> 1, c'est à dire plusieurs valeurs de hash se partageant une même entrée, donc obligation de les chaîner, ce que fait la Hashtable de façon transparente mais pas sans conséquences !

Il découle de cette situation que lorsque vous programmez un accès à la table de hash, au lieu que l'algorithme (dans le cas idéal 1->1) tombe directement sur la cellule du tableau qui correspond à la clé (hash code), il est obligé de parcourir par chaînage avant toutes les entrées correspondantes... De là une dégration nette des performances alors qu'on a généralement choisi une Hashtable pour améliorer les performances (au lieu d'une simple liste qu'il faut balayer à chaque recherche). On a donc, sans trop le savoir, recréé une liste qui est balayée là où on devrait avoir des accès directs...

La solution : surcharger GetHashCode()

Il existe plusieurs stratégies pour générer un "bon" hash code. L'idée étant de répartir le plus harmonieusement les valeurs de sorties dans l'espace de la table pour éviter, justement, les collisions de clés. Ressortez vos cours d'informatique du placard, vous avez forcément traité le sujet à un moment ou un autre ! Pour les paresseux et ceux qui n'ont pas eu de tels cours, je ne me lancerais pas dans la théorie mais voici quelques exemples d'implémentations de GetHashCode() pour vous donner des idées :

La méthode "bourrin"

Quand on ne comprend pas forcément les justifications et raisonnements mathématiques d'un algorithme, le mieux est de faire simple, on risque tout autant de se tromper qu'en faisant compliqué, mais au moins c'est facile à mettre en œuvre et c'est facile à maintenir :-)

Imaginons une structure simple du genre :

```
public struct MyStruct
{
```

```
  public int Entier { get; set; }
  public string Chaine { get; set; }
  public DateTime LaDate { get; set; }
}
```

Ce qui différencie une instance d'une autre ce sont les valeurs des champs. Le plus simple est alors de construire une "clé" constituée de toutes les valeurs concaténées et séparées par un séparateur à choisir puis de laisser le framework calculer le hash code de cette chaîne. Toute différence dans l'une des valeurs formera une chaine-clé différente et par conséquence un hash code différent. Ce n'est pas super subtile, mais ça fonctionne. Regardons le code :

```
public string getKey()
{ return Entier + "|" + Chaine + "|" + LaDate.ToString("yyyyMMMddHHmmss");
} public overrideint GetHashCode() {return getKey().GetHashCode(); }
```

J'ai volontairement séparé la chose en deux parties en créant une méthode getKey pour pouvoir l'afficher.

La sortie (dans un foreach) de la clé d'un exemple de 5 valeurs avec leur hash code donne :

```
1|toto|2008juil.11171952 Code: -236695174
10|toto|2008juil.11171952 Code: -785275536
100|zaza|2008juil.01171952 Code: -684875783
0|kiki|2008sept.11171952 Code: 888726335
0|jojo|2008sept.11171952 Code: 1173518366
```

La méthode Resharper

Ce merveilleux outil se propose de générer pour vous la gestion des égalités et du GetHashCode, laissons-le faire et regardons le code qu'il propose (la structure a été au passage réécrite, les propriétés sont les mêmes mais elles utilisent des champs privés) : D'abord le code de hachage :

```
public override int GetHashCode()
{
  unchecked
  {
    int result = entier;
    result = (result*397) ^ (chaine != null ? chaine.GetHashCode() : 0);
    result = (result*397) ^ laDate.GetHashCode();
    return result;
  }
}
```

On voit ici que les choix algorithmiques pour générer la valeur sont un peu plus subtiles et qu'ils ne dépendent pas de la construction d'une chaîne pour la clé (ce qui est consommateur de temps et de ressource).

Profitons-en pour regarder comment le code gérant l'égalité a été généré (ainsi que le support de l'interface IEquatable<MyStruct> qui a été ajouté à la définition de la structure) - A noter, la génération de ce code est optionnel - :

```
public static bool operator ==(MyStruct left, MyStruct right)
{ return left.Equals(right); }

public static bool operator !=(MyStruct left, MyStruct right)
{ return !left.Equals(right); }
public bool Equals(MyStruct obj)
{ return obj.entier == entier && Equals(obj.chaine, chaine) && obj.laDate.Equals(laDate); }

public override bool Equals(object obj)
{
   if (obj.GetType() != typeof(MyStruct)) return false;
   return Equals((MyStruct)obj);
}
```

Bien que cela soit optionnel et n'ait pas de rapport direct avec GethashCode, on notera l'intérêt de la redéfinition de l'égalité et des opérateurs la gérant ainsi que le support de IEquatable. Une classe et encore plus une structure se doivent d'implémenter ce "minimum syndical" pour être sérieusement utilisables. Sinon gare aux bugs difficiles à découvrir (en cas d'utilisation d'une égalité même de façon indirecte) !

De même tout code correct se doit de surcharger ToString(), ici on pourrait simplement retourner le champ LaChaine en supposant qu'il s'agit d'un nom de personne ou de chose, d'une description. Tout autre retour est possible du moment que cela donne un résultat lisible. Ce qui est très pratique si vous créez une liste d'instances et que vous assignez cette liste à la propriété DataSource d'un listbox ou d'une combo... Pensez-y !

Conclusion

Créer des classes ou des structures, si on programme sous C# on en a l'habitude puisque aucun code ne peut exister hors de telles constructions. Mais "bien" construire ces classes et structures est une autre affaire. Le framework propose notamment beaucoup d'interfaces qui peuvent largement améliorer le comportement de votre code. Nous avons vu ici comment surcharger des méthodes héritées de object et leur importance, nous avons vu aussi l'interface IEquatable. IDisposable, INotityPropertyChanged, ISupportInitialize, et bien d'autres sont autant d'outils que vous pouvez (devez ?) implémenter pour obtenir un code qui s'intègre logiquement au framework et en tire tous les bénéfices.

Quizz C#. Vous croyez connaître le langage ? et bien regardez ce qui suit !

C# est un langage merveilleux, plein de charme... et de surprises !

En effet, s'écartant des chemins battus et intégrant de nombreuses extensions comme Linq aujourd'hui ou les méthodes anonymes hier, il est en perpétuelle évolution. Mais ces "extensions" transforment progressivement un langage déjà subtile à la base (plus qu'il n'y paraît) en une jungle où le chemin le plus court n'est pas celui que les explorateurs que nous sommes auraient envisagé...

Pour se le prouver, 6 quizz qui vous empêcheront de bronzer idiot ou de vous endormir au boulot ! (Les réponses se trouvent après la conclusion, ne trichez pas !).

Quizz 1

Etant données les déclarations suivantes :

```
class ClasseDeBase
{
   public virtual void FaitUnTruc(int x)
   { Console.WriteLine("Base.FaitUnTruc(int)"); }
}
class ClasseDérivée : ClasseDeBase
{
   public override void FaitUnTruc(int x)
  { Console.WriteLine("Dérivée.FaitUnTruc(int)"); }

  public void FaitUnTruc(object o)
 { Console.WriteLine("Dérivée.FaitUnTruc(object)"); }
}
```

Pouvez-vous prédire l'affiche du code suivant et expliquer la sortie réelle :

```
   ClasseDérivée d = new ClasseDérivée();
   int i = 10;
   d.FaitUnTruc(i);
```

Gratt' Gratt' Gratt'.....

Quizz 2

Pouvez-vous prédire l'affichage de cette séquence et expliquer l'affichage réel ?

```
double d1a = 1.00001;
double d2a = 0.00001;
Console.WriteLine((d1a - d2a) == 1.0);
double d1b = 1.000001;
double d2b = 0.000001;
Console.WriteLine((d1b - d2b) == 1.0);
double d1c = 1.0000001;
double d2c = 0.0000001;
Console.WriteLine((d1c - d2c) == 1.0);
```

Je vous laisse réfléchir ...

Quizz 3

Toujours le même jeu : prédire ce qui va être affiché et expliquer ce qui est affiché réellement... c'est forcément pas la même chose sinon le quizz n'existerait pas :-)

```
List<Travail> travaux = new List<Travail>();
Console.WriteLine("Init de la liste de delegates");
for (int i = 0; i < 10; i++)
{ travaux.Add(delegate { Console.WriteLine(i); }); }
Console.WriteLine("Activation de chaque delegate de la liste");
foreach (Travail travail in travaux) travail();
```

Les apparences sont trompeuses, méfiez-vous !

Quizz 4

Ce code compile-t-il ?

```
public enum EtatMoteur { Marche, Arrêt }

public static void DoQuizz()
{
   EtatMoteur etat = 0.0;
   Console.WriteLine(etat);
}
```

Quizz 5

Etant données les déclarations suivantes :

```
private static void Affiche(object o)
{ Console.WriteLine("affichage <object>"); }
private static void Affiche<T>(params T[] items)
{ Console.WriteLine("Affichage de <params T[]>"); }
```

Pouvez-vous prédire et expliquer la sortie de l'appel suivant :

```
Affiche("Qui va m'afficher ?");
```

Je sens que ça chauffe :-)

Quizz 6

Etant données les déclarations suivantes :

```
delegate void FaitLeBoulot(); private static FaitLeBoulot FabriqueLeDélégué()
{
  Random r = new Random();
  Console.WriteLine("Fabrique. r = "+r.GetHashCode());

  return delegate {
                    Console.WriteLine(r.Next());
                    Console.WriteLine("delegate. r = "+r.GetHashCode());
                  };
}
```

Quelle sera la sortie de la séquence suivante :

```
FaitLeBoulot action = FabriqueLeDélégué();
action();
action();
```

Conclusion

C# est un langage d'une grande souplesse et d'une grande richesse. Peut-être qu'à devenir trop subtile il peut en devenir dangereux, comme C++ était dangereux car trop permissif.

Avec C#, même de très bons développeurs peuvent restés interloqués par la sortie réelle d'une séquence qui n'a pourtant pas l'air si compliquée.

Ses avantages sont tels qu'il mérite l'effort de compréhension supplémentaire pour le maîtriser réellement, mais tout développeur C# (les moyens comme ceux qui pensent être très bons!) doit être mis au moins une fois dans sa vie face à un tel quizz afin de lui faire toucher la faiblesse de son savoir et les risques qu'il prend à coder sans vraiment connaître le langage.
Comme toujours l'intelligence est ce bel outil qui nous permet de mesurer à quel point nous ne savons rien.
Nous sommes plutôt habitués à envisager cette question sous un angle métaphysique, le développement nous surprend lorsque lui aussi nous place face à cette réalité...
Le projet ci-dessous contient les solutions, ne trichez pas !

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

namespace Quizz
{
    class Program
    {
        static void Main(string[] args)
        {


            Quizz("Quizz 1", Quizz1.DoQuizz);

            Quizz("Quizz 2", Quizz2.DoQuizz);

            Quizz("Quizz 3", Quizz3.DoQuizz);

            Quizz("Quizz 4", Quizz4.DoQuizz);

            Quizz("Quizz 5", Quizz5.DoQuizz);

            Quizz("Quizz 6", Quizz6.DoQuizz);

        }

        private delegate void TestIt();

        /// <summary>
        /// Launch a quizz with a title and a pause at end
        /// </summary>
        /// <param name="title">quizz title</param>
        /// <param name="test">the quizz start method</param>
        private static void Quizz(string title, TestIt test)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine("Quizz : " + title + "\n");
```

```
    Console.ForegroundColor = ConsoleColor.White;
    if (test != null) test();
    Console.WriteLine();
    Console.ForegroundColor = ConsoleColor.Gray;
    Console.Write("...<Return>...");
    Console.ReadLine();
}


#region quizz 1

// Question : Quel sera l'affichage et pourquoi ?

class ClasseDeBase
{
    public virtual void FaitUnTruc(int x)
    {
        Console.WriteLine("Base.FaitUnTruc(int)");
    }
}

class ClasseDérivée : ClasseDeBase
{
    public override void FaitUnTruc(int x)
    {
        Console.WriteLine("Dérivée.FaitUnTruc(int)");
    }

    public void FaitUnTruc(object o)
    {
        Console.WriteLine("Dérivée.FaitUnTruc(object)");
    }
}

public static class Quizz1
{
    public static void DoQuizz()
    {
        ClasseDérivée d = new ClasseDérivée();
        int i = 10;
        d.FaitUnTruc(i);
    }
}

// Réponse : affichage de "Dérivée.FaitUnTruc(object)"
// Au moment de choisir une surcharge, s'il existe des méthodes compatibles dans la classe dérivée alors
// les signatures déclarées dans la classe de base sont ignorée, même si elles sont surchargées dans
// la même classe dérivée.
// Ce qui explique que bien que la meilleure signature soit (int x) qui est la plus précise (ce qui est
// le choix normal du compilateur, il ne s'en sert pas car il ne la "voit" plus pour la raison évoquée...
```

```
#endregion

#region quizz 2

// Question : Prédire l'affichage. Et expliquer celui qui est réellement exécuté.

public static class Quizz2
{
    public static void DoQuizz()
    {

        double d1a = 1.00001;
        double d2a = 0.00001;
        Console.WriteLine((d1a - d2a) == 1.0);
        Console.WriteLine((d1a - d2a)); // 1.0

        double d1b = 1.000001;
        double d2b = 0.000001;
        Console.WriteLine((d1b - d2b) == 1.0);
        Console.WriteLine((d1b - d2b)); // 0.99999999999999989

        double d1c = 1.0000001;
        double d2c = 0.0000001;
        Console.WriteLine((d1c - d2c) == 1.0);
        Console.WriteLine((d1c - d2c)); // 1.0
    }
}

// Réponse : ahhh la représentation binaire des numériques, en programmation comme avec les SGBD
// cette erreur se rencontre souvent ! La sortie 2 vous dira "false" alors que la 1ere et la 3eme
// donne "true"... Mystère ? Non représentation binaire des doubles. Mettez un point d'arrêt et
// regardez le contenu des variables.. Ce n'est pas forcément 1.0 mais 0.99999... que vous verrez.
// le 1.0 du test peu être stocké avec précision, mais hélas les 1.0xxx01, ne le sont pas et tout
// dépend du nombre de décimale.

#endregion

#region quizz 3

// Question : Prédire l'affichage et expliquer la différence avec la réalité.

public static class Quizz3
{
    private delegate void Travail();

    public static void DoQuizz()
    {
        List<Travail> travaux = new List<Travail>();

        Console.WriteLine("Init de la liste de delegates");
        for (int i = 0; i < 10; i++)
```

```
        {
            travaux.Add(delegate { Console.WriteLine(i); });
        }

        Console.WriteLine("Activation de chaque delegate de la liste");
        foreach (Travail travail in travaux) travail();

    }
}

// Réponse : On frôle la mystique ici... Celle qui entour le principe de "variables capturées"
// (voir le quizz 6). Si nous regardons le code il n'y a qu'une variable "i" qui change à chaque
// itération. Mais la méthode anonyme "capture" au moment de sa création la variable elle-même et
// non pas sa valeur. En fin de boucle "i" vaut 10, et donc quand invoque les 10 items de la liste
// ensuite, nous avons 10 fois la valeur _actuelle_ de "i" (donc 10) à chaque affichage...

#endregion

#region quizz 4

// Question : ce code compile-t-il ? Pourquoi ?

public static class Quizz4
{
    public enum EtatMoteur { Marche, Arrêt }

    public static void DoQuizz()
    {
        EtatMoteur etat = 0.0;
        Console.WriteLine(etat);
    }
}

// Réponse : Bug de C# ... Cela ne devrait compiler car selon les spécifications du langage seule
// la valeur littérale "0" est compatible avec tous les types énumérés (créés par enum). Le décimal 0.0
// ne devrait pas être accepté par le compilateur. C# doit être corrigé ou bien les specs modifiées !
// (note: a tester sous Mono pour voir si leur implémentation fait attention à ce détail).
// (note: Resharper 3.x indique une erreur dans le code en expliquant que le double ne peut pas être
// casté en EtatMoteur. Merci Resharper !).

#endregion

#region quizz 5

// Question : deviner la sortie de ce programme et expliquer celle qui est réellement affichée.

public static class Quizz5
{
    public static void DoQuizz()
    {
        Affiche("Qui va m'afficher ?");
```

```
    }

    private static void Affiche(object o)
    {
        Console.WriteLine("affichage <object>");
    }

    private static void Affiche<T>(params T[] items)
    {
        Console.WriteLine("Affichage de <params T[]>");
    }
}

// Réponse : C'est "Affichage de <params T[]>" qui sera affiché à la console !
// Pourquoi donc le compilateur choisit-il de transformer la chaine de caractère passée en argument
// en un élément d'un tableau de 1 item qu'il doit fabriquer au lieu d'aller au plus simple avec
// la signature "object" ?
// ... parce que C# ne va pas au plus simple mais au plus précis ! Dans un cas il doit caster la chaîne
// en "object" ce qui est pour le moins très générique (rien de plus générique que la classe object
// dans le framework), soit il peut opter pour un tableau typé de chaînes. En effet, la signature
// utilisant le type générique T fait qu'avec une chaîne en argument elle devient un tableau de chaînes.
// Pour C# mieux faut un tableau de chaînes qui conserve le type de l'argument (chaîne) à une signature
// plus directe et plus simple mais qui oblige à cast faisant perdre de la spécificité au type de l'
// argument...

#endregion

#region quizz 6

// Question : qu'affiche ce code ? et pourquoi ?

public static class Quizz6
{
    public static void DoQuizz()
    {
        FaitLeBoulot action = FabriqueLeDélégué();
        action();
        action();
    }

    delegate void FaitLeBoulot();

    private static FaitLeBoulot FabriqueLeDélégué()
    {
        Random r = new Random();
        Console.WriteLine("Fabrique. r = "+r.GetHashCode());
        return delegate { Console.WriteLine(r.Next());
                          Console.WriteLine("delegate. r = "+r.GetHashCode());
        };
    }
```

```
        }

        // Réponse : La variable "r" est dans le scope de FaitLeBoulot. Peut-elle exister en dehors de ce scope ?
        // Normalement non... mais ici oui ! En raison de la "capture de variable" rendue indispensable par
        // les méthodes anonymes comme celle de l'exemple. De fait, la variable "r" est capturée par le delegate
        // et chaque appel à "action()" se sert de cette variable et non de la valeur qu'on pense avoir été
        // générée dans FabriqueLeDélégué (avec son r.Next()) !
        // Pour matérialiset cet état de fait j'ai ajouté l'affichage du hashcode de la variable "r".
        // L'appel à "FabriqueDélégué" est bien exécuté qu'une seule fois, on voit ainsi le premier hashcode,
        // ensuite le délégué stocké dans "action" est appelé deux fois, et on voit que hashcode est deux fois le
        // même, et qu'il est égal au premier : il y a bien une instance Random créée une fois dans l'appel à
        // "FabriqueLeDélégué" et cette instance a été "capturée" par la méthode anonyme et elle existe toujours
        // même en dehors de la portée de "FaireLeBoulot" !


        #endregion

    }
}
```

Le blues du "Set Of" de Delphi en C#

Il y a bien fort peu de chose qui puisse faire regretter un delphiste d'être passé à C#, et quand je parle de regrets, c'est un mot bien fort, disons plus modestement des manques agaçants.

Rien qui puisse faire réellement pencher la balance au point de revenir à Delphi, non, ce langage est bel et bien mort, assassiné par Borland/Inprise/Borland/CodeGear et son dernier big boss, tod nielsen qui ne mérite pas même les majuscules à son nom mais là n'est pas la question.

Donc il existe syntaxiquement trois choses qui peuvent agacer le delphiste, même, comme moi, quand cela fait des années maintenant que je ne pratique presque plus que C#.

La première qui revient à longueur de code, ce sont ces satanées parenthèses dans les "if". On n'arrête pas de revenir en arrière parce qu'on rajoute un test dans le "if" et qu'il faut remettre tout ça entre de nouvelles parenthèses qui repartent depuis le début. Certes on gagne l'économie du "then" pascalien, mais que ces parenthèses du "if" sont épouvantables et ralentissent la frappe bien plus qu'un "then" unique et ponctuel qu'on ne touche plus une fois écrit même si on rajoute "and machin=truc" ! A cela pas d'astuce ni truc. Et aucun espoir que les parenthèses de C# dans les "if" soient abandonnées un jour pour revenir au "then" ... Donc faut faire avec ! Le dogme "java/C++" est bien trop fort (quoi que C# possède le Goto de Delphi, ce qui n'est pas la meilleure idée du langage :-)).

La seconde tracasserie syntaxique est cette limitation totalement déroutante des indexeurs : un seul par classe et il ne porte pas de nom. this[], un point c'est tout. Je sais pas ce que notre bon Hejlsberg avait en tête, mais pourquoi diable n'a-t-il repris de Delphi qu'un seul indexeur et non pas la feature entière ? Il fait beaucoup de recherche, et le fait bien, mais je présume qu'il n'a jamais plus codé un "vraie" appli depuis des lustres... Car dans la vraie vie, il existe plein de situations où un objet est composé de plus d'une collection. Une voiture a 4 roues et 2 ou 4 portières. Pourquoi lorsque je modélise la classe Voiture je devrais donner plus d'importance aux roues qu'aux portières et choisir lesquelles auront le droit d'être l'unique indexeur de la classe ? Pourquoi tout simplement ne pas avoir Voiture.Roues[xx] et Voiture.Portières[yy] ? Mon incompréhension de ce choix très gênant dans plus d'un cas a été et reste des années après totale. Surtout après toutes les évolutions de C# sans que jamais cette erreur de conception ne soit corrigée. Pas suffisant pour faire oublier toute la puissance de C# et le bonheur qu'on a à travailler dans ce langage, mais quand même, ça agace.

Enfin, dans la même veine d'ailleurs, l'absence de "Set of" est cruelle. Pourquoi avec zappé cette feature de Delphi dans C# alors que bien d'autres ont été reprises (avec raison ou moins comme le Goto) ?

Mais là on peut trouver des astuces et certains (dont je fais partie) ont écrit ou essayer d'écrire des classes "SetOf" qui permettent de gérer des ensembles comme Delphi, mais que c'est lourd tout ce code au lieu d'écrire "variable machin : set of typeTruc" !

Autre astuce moins connue, et c'est pour ça que je vous la livre, est l'utilisation fine des Enums. En effet, tout comme sous Delphi, il est possible d'affecter des valeurs entières à chaque entrée d'une énumération. On peut donc déclarer "enum toto { item1 = 5, item2 = 28, item3 = 77 }".

Mais ce que l'on sait moins c'est que rien ne vous interdit d'utiliser des puissances de 2 explicitement car les Enums savent d'une part parser des chaînes séparées par des virgules et d'autre part savent reconnaître les valeurs entières cumulées.

Ainsi, avec enum Colors { rouge=1, vert=2, bleu=4, jaune=8 }; on peut déclarer :
Colors orange = (Colors)Enum.Parse(typeof(Colors),"rouge,jaune"); // étonnant non ?
La valeur de "orange" sera 9 qu'on pourra décomposer facilement, même par un simple Convert.ToInt64.

Pour ne pas réinventer la roue et éviter de faire des coquilles je reprends cet exemple tel que fourni dans la doc MS. Voici le code complet qui vous permettra, peut-être, d'oublier plus facilement "Set of" de Dephi...

Stay tuned!

Code de la doc MS :

```
using System;

public class ParseTest
{
   [FlagsAttribute]
   enum Colors { Red = 1, Green = 2, Blue = 4, Yellow = 8 };

   public static void Main()
   {
      Console.WriteLine("The entries of the Colors Enum are:");
      foreach (string colorName in Enum.GetNames(typeof(Colors)))
      {
         Console.WriteLine("{0}={1}", colorName,
                           Convert.ToInt32(Enum.Parse(typeof(Colors), colorName)));
      }
      Console.WriteLine();
      Colors myOrange = (Colors)Enum.Parse(typeof(Colors), "Red, Yellow");
      Console.WriteLine("The myOrange value {1} has the combined entries of {0}",
                  myOrange, Convert.ToInt64(myOrange));
   }
}

/*
This code example produces the following results:

The entries of the Colors Enum are:
Red=1
Green=2
Blue=4
Yellow=8

The myOrange value 9 has the combined entries of Red, Yellow

*/
```

Les class Helpers, enfer ou paradis ?

Class Helpers

Les class helpers sont une nouvelle feature du langage C# 3.0 (voir mon billet et mon article sur les nouveautés de C# 3.0).
Pour résumer il s'agit de "**décorer**" une classe existante avec de nouvelles méthodes sans modifier le code de cette classe, les méthodes en questions étant implémentées dans une autre classe.
Le principe lui-même n'est pas récent puisque Borland l'avait "inventé" pour Delphi 8.Net (la première version de Delphi sous .Net) afin de permettre l'ajout des méthodes de TObject à System.Object (qu'ils ne pouvaient pas modifier bien entendu) afin que la VCL puisse être facilement portée sous .Net. Borland avait déposé un brevet pour ce procédé (ils le prétendaient à l'époque en tout cas) et je m'étonne toujours que Microsoft ait pu implémenter exactement la même chose dans C# sans que cela ne fasse de vagues. Un mystère donc, mais là n'est pas la question.

Mauvaises raisons ?

Les deux seules implémentations de cet artifice syntaxique que je connaisse (je ne connais pas tout hélas, si vous en connaissez d'autres n'hésitez pas à le dire en commentaire que l'on puisse comparer) sont donc celle de Borland dans Delphi 8 pour simplifier le portage de la VCL sous .NET et celle de Microsoft dans C# pour faciliter l'intégration de Linq dans le langage.

Deux exemples, deux fois pour la même raison un peu spécieuse à mes yeux : simplifier le boulot du concepteur du langage pour supporter de nouvelles features. Deux fois une mauvaise raison à mes yeux, un peu trop puristes peut-être, qui pensent qu'un élément syntaxique doit se justifier d'une façon plus forte, plus théorique que simplement "pratique".

Résultat, je me suis toujours méfié des class helpers car leur danger est qu'un objet se trouve d'un seul coup affublé de méthodes "*sorties d'un chapeau*", c'est à dire qu'il semble exposer des méthodes publiques qui ne sont *nulles part dans son code*. J'ai horreur de ce genre de combines qui, à mon sens, favorise un code non maintenable. Si j'adore la magie, à voir ou à pratiquer, je la déteste lorsque je porte ma casquette d'informaticien... J'ai donc toujours conseillé la plus grande circonspection vis à vis de cet artifice syntaxique, que ce soit à l'époque (déjà lointaine.. le temps passe!) où je faisais du Delphi que maintenant sous C# qui vient d'ajouter cette fioriture à sa palette.
Jusqu'à aujourd'hui, faute d'avoir trouvé une utilisation intelligente des class helpers qui ne puissent être mise en œuvre plus "proprement", les class helpers étaient à mon sens plutôt à classer du côté enfer que paradis. Interface et héritage m'ont toujours semblé une solution préférable à ces méthodes fantômes.

Trouver une justification

Mais il n'y a que les imbéciles qui ne changent pas d'avis, n'est-ce pas... Et je cherche malgré tout toujours à trouver une utilité à un outil même si je le trouve inutile de prime abord, réflexe d'ingénieur qui aime trouver une place à toute chose certainement.
J'ai ainsi essayé plusieurs fois dans des projets de voir si les class helpers pouvaient rendre de vrais services avec une réelle justification, c'est à dire sans être un cache misère ni une façon paresseuse de modifier une classe sans la modifier tout en la modifiant...
Comme j'ai enfin trouvé quelques cas (rares certes) dans lesquels les class helpers me semblent avoir une justification pratique, je me suis dit que vous en toucher deux mots pourraient éventuellement faire avancer votre propre réflexion sur le sujet (même si j'ai bien conscience que je dois être assez seul à me torturer la cervelle pour trouver absolument une utilité aux class helpers :-)).

Bref, passons à la pratique.

Un cas pratique

Premier cas, les chaînes de caractères. Voilà un type de données vieux comme la programmation qui pourrait être un bon candidat à l'exploitation des possibilités des class helpers. En effet, hors de question de dériver la classe System.String et encore moins de pouvoir modifier le mot clé "string" de C#. Pourtant nous utilisons très souvent les mêmes fonctions "personnelles" sur les chaînes de caractères d'un même projet.

Par exemple, j'ai pour principe que les chaînes exposées par une classe (propriétés de type string donc) ne soient jamais à null. De ce fait, dans toutes les classes qui exposent un string, j'ai, dans le setter, la même séquence qui change l'éventuel null de *value* à string.Empty. C'est assez casse-pieds à répéter comme ça mécaniquement dans toutes les propriétés string de toutes les classes.
Et là, pas de possibilité de faire supporter une interface à System.String, ni de la dériver comme je le disais plus haut. C'est ici que les class helpers peuvent trouver une première justification pratique pour le développeur en dehors d'avoir facilité la vie à MS pour implémenter Linq.

Prenons le code suivant que nous plaçons dans une unité "Tools" de notre projet :

```
public static class Utilities
{
    public static string NotNullString(this string s)
     { return !string.IsNullOrEmpty(s) ? s.Trim() : string.Empty; }
...
 }
```

La classe *Utilities* est une classe statique qui contient tous les petits bouts de code utilisables dans tout le projet. Parmi ces méthodes, on voit ici l'implémentation du class helper

"NotNullString". D'après cette dernière, la méthode ne sera visible que sur les instances de la classe "string". Le code lui-même est d'une grande simplicité puisqu'il teste si la chaîne en question est vide ou non, et, selon le cas, retourne string.Empty ou bien un Trim() de la chaîne. J'aime bien faire aussi systématiquement un Trim() sur toutes les propriétés string, je trouve ça plus propre surtout si on doit tester des équalités et que les chaînes proviennent de saisies utilisateurs.

Dans la pratique il suffira maintenant n'importe où dans notre projet d'écrire la chose suivante pour être certain qu'à la sortie de l'affectation la chaîne résultante ne sera jamais nulle et qu'elle ne comportera jamais d'espaces en trop en début ou en fin :

```
public string BusinessID
{   get { return _BusinessID; }
    set {  if (value != _BusinessID)
          { _BusinessID = value.NotNullString().ToUpperInvariant();
            DoChange("BusinessID");
          }
        }
}
```

On voit ici une propriété string "BusinessID" qui, dans son setter, utilise désormais la nouvelle méthode fantome de la classe string... En fin de séquence nous sommes certains que _BusinessID est soit vide, soit contient un chaîne sans espace de tête ou de queue (et qu'elle est en majuscules, en plus dans cet exemple).

Voici donc une première utilisation "intelligente" des class helpers, la décoration d'une classe du framework (ou d'une lib dont on n'a pas le code source) pour lui ajouter un comportement, éventuellement complexe, dont on a souvent l'utilité dans un projet donné.

On pourrait penser ainsi à une fonction "ProperCase" qui passe automatiquement la casse d'une chaîne en mode "nom de famille", c'est à dire première lettre de chaque mot en majuscule, le reste en minuscule, ou à bien d'autres traitements qu'on aurait besoin d'appliquer souvent à des chaînes dans un projet.

Encore un autre cas

Dans la même veine, une application doit souvent manipuler des données issues d'une base de données et les modifier (en plus d'en insérer de nouvelles). On le sait moins (mais on s'aperçoit vite quand cela bug!) que le framework .NET possède, pour les dates par exemple, ses propres mini et maxi qui ne sont pas compatibles à toutes les bases de données,

notamment SQL Server. Si vous attribuer à une date la valeur DateTime.MinValue et que vous essayez d'insérer cette dernière dans un champ Date d'une base SQL Server vous obtiendrez une exception de la part du serveur : la date passée n'est pas dans la fourchette acceptée par le serveur.
Dommage... DateTime.MinValue est bien pratique...
On peut bien entendu fixer une constante dans son projet et l'utiliser en place et lieu de DateTime.MinValue. Voci un exemple pour MaxValue (le problème étant le même):

```
DateTime MaxiDate = new DateTime(3000, 1, 1, 0, 0, 0);
```

Il suffira donc d'utiliser MaxiDate à la place de DateTime.MaxValue. La date considérée comme "maxi" est ici arbitraire (comme on le ferait pour la date mini) et est choisie pour représenter une valeur acceptable à la fois pour l'application et pour la base de données. Ici on notera que je prépare le terrain pour le bug de l'an 3000. Moins stupide qu'un coboliste et le bug de l'an 2000, vous remarquerez que je me suis arrangé ne plus être joignable à la date du bug et que mes héritiers sont aussi à l'abris de poursuites judiciaires pour quelques siècles :-)

L'utilisation d'une constante n'a rien de "sale" ou de "moche", c'est un procédé classique en programmation, même la plus éthérée et la plus sophistiquée. Toutefois, Puisque DateTime existe, puisque DateTime est un type complexe (une classe) et non pas un simple emplacement mémoire (comme les types de base en Pascal par exemple), puisque cette classe expose déjà des méthodes, dont Min et MaxValue, il serait finalement plus "linéaire" et plus cohérent d'ajouter notre propre MaxValue à cette dernière en place et lieu d'une constante.

Encore un bon exemple d'utilisation des class helpers. Ici nous homogénéisons notre style de programmation en évitant le mélange entre méthodes de DateTime et utilisation d'une constante. De plus, en ajoutant une méthode spécifique à DateTime, celle-ci sera visible par Intellisense come membre de DateTime, ce qui ne serait pas le cas de la constante.

Dans la même classe Utilities nous trouverons ainsi :

```
public static DateTime SQLMaxValue(this DateTime d)
{ return new DateTime(3000, 1, 1, 0, 0, 0); }
```

Et nous voici avec la possibilité d'écrire : MaDate = DateTime.SQLMaxValue();

Conclusion

Enfer ou paradis ? Les class helpers, comme tout artifice syntaxique peuvent se ranger dans les deux catégories, ils ne sont qu'un outil à la disposition du développeur. Un simple marteau peut servir à bâtir une maison où vivre heureux ou bien à assassiner sauvagement

son voisin... Les objets inanimés n'ont pas de conscience, ou plutôt, si, ils en ont une : celle de celui qui les utilise. A chacun d'utiliser les outils que la technologie humaine met à sa disposition pour créer un enfer ou un paradis...

Techniquement, je n'enfoncerai pas les portes ouvertes en donnant l'impression d'avoir découvert une utilisation miraculeuse des class helpers. Cela serait stupide, puisque justement cette syntaxe a été créée par Borland puis MS pour justement décorer des classes dont on ne possède pas le source (pas d'ajout de méthode ni d'interface) et qui ne sont pas héritables. En ajoutant des class helpers à string ou DateTime nous ne faisons rien d'autre que d'utiliser les class helpers exactement en conformité avec ce pour quoi ils ont été créés.

L'intérêt de ce billet se situe alors dans deux objectifs : vous permettre de réfléchir à cette nouveauté de C#3.0 que vous ne connaissez peut-être pas ou mal et vous montrer comment, en pratique, cela peut rendre des services non négligeables.

Si l'un ou l'autre de ces objectifs a été atteint, vous tenez alors votre récompense d'avoir tout lu jusqu'ici, et moi d'avoir écrit ce billet :-)

Les class Helper : s'en servir pour gérer l'invocation des composants GUI en multithread

Les class helper dont j'ai déjà parlé ici peuvent servir à beaucoup de choses, si on se limite à des services assez génériques et qu'on ne s'en sert pas pour éclater le code d'une application qui deviendra alors très difficile à maintenir. C'est l'opinion que j'exprimais dans cet ancien billet et que je conserve toujours.

Dès lors trouver des utilisations pertinentes des class helpers n'est pas forcément chose aisée, pourtant il ne faudrait pas les diaboliser non plus et se priver des immenses services qu'ils peuvent rendent lorsqu'ils sont utilisés à bon escient.

Dans le blog de Richard on trouve un exemple assez intéressant à plus d'un titre. D'une part il permet de voir que les class helpers sont des alliés d'une grande efficacité dès qu'il s'agit de trouver une solution globale à un problème répétitif. Mais d'autre part cet exemple, par l'utilisation des génériques et des expressions lambda, a l'avantage de mettre en scène tout un ensemble de nouveautés syntaxiques de C# 3.0 en quelques lignes de code. Et les de ce genre sont toujours formateurs.

Pour ceux qui lisent l'anglais, allez directement sur le billet original en cliquant ici. Pour les autres, voici non pas un résumé mais une interprétation libre sur le même sujet :
Le problème à résoudre : l'invocation en multithread.

Lorsqu'un thread doit mettre à jour des composants détenus par le thread principal cela doit passer par un appel à Invoke car seul le thread principal peut mettre à jour les contrôles qu'il possède. Cette situation est courante. Par exemple un traitement en tâche de fond qui doit mettre à jour une barre de progression.

Bien entendu il ne s'agit pas de bricoler directement les composants d'une form depuis un thread secondaire, ce genre de programmation est à proscrire, mais même en créant dans la form une propriété publique accessible au thread, la modification de cette propriété se fera à l'intérieur du thread secondaire et non pas dans le thread principal...

Il faut alors détecter cette situation et trouver un moyen de faire la modification de façon "détournée", c'est à dire de telle façon à ce que ce soit le thread principal qui s'en charge. Les Windows Forms et les contrôles conçus pour cette librairie mettent à la disposition du développeur la méthode InvokeRequired qui permet justement de savoir si le contrôle nécessite l'indirection que j'évoquais plus haut ou bien s'il est possible de le modifier directement. Le premier cas correspond à une modification depuis un thread secondaire, le dernier à une modification du contrôle depuis le thread principal, cas le plus habituel.

La méthode classique
Sous .NET 1.1 le framework ne détectait pas le conflit et les applications mal conçues pouvaient planter aléatoirement si des modifications de contrôles étaient effectuées depuis des threads secondaires. Le framework 2.0 a ajouté plus de sécurité en détectant la situation qui déclenche une exception, ce qui est bien préférable aux dégâts aléatoires...
Donc, pour s'en sortir on doit écrire un code du genre de celui-ci :

```
[...]
NetworkChange.NetworkAddressChanged += new
NetworkAddressChangedEventHandler(NetworkChange_NetworkAddressChanged);
[...]
delegate void SetStatus(bool status);
void NetworkChange_NetworkAddressChanged(object sender, EventArgs e)
{
    bool isConnected = IsConnected();
    if (InvokeRequired)
      Invoke(new SetStatus(UpdateStatus), new object[] { isConnected });
    else
      UpdateStatus(isConnected);
}
```

```
void UpdateStatus(bool connected)
{
    if (connected)
      this.connectionPictureBox.ImageLocation = @"..\bullet.green.gif";
    else
      this.connectionPictureBox.ImageLocation = @"..\bullet.red.gif";
}
[...]
```

Cette solution classique impose la création d'un délégué et beaucoup de code pour passer d'une modification directe à une modification indirecte selon le cas. Bien entendu le code en question doit être dupliqué pour chaque contrôle pouvant être modifié par un thread secondaire... C'est assez lourd, convenons-en...

Pour la compréhension, le code ci-dessus change l'image d'une PictureBox pour indiquer l'état (vert ou rouge) d'une connexion à un réseau et le code appelant cette mise à jour de l'affichage peut émaner d'un thread secondaire.

Comme on le voit, la méthode est fastidieuse et va avoir tendance à rendre le code plus long, moins fiable (coder plus pour bugger plus...), et moins maintenable. C'est ici que l'idée d'utiliser un class helper prend tout son intérêt...

La solution via un class helper

La question qu'on peut se poser est ainsi "*n'existe-t-il pas un moyen générique de résoudre le problème ?*". De base pas vraiment. Mais avec l'intervention d'un class helper, si, c'est possible (© Hassan Céhef - joke pour les amateurs des "nuls"). Voici le class helper en question :

```
public static TResult Invoke<T, TResult>(this T controlToInvokeOn, Func<TResult> code)
where T : Control
{
  if (controlToInvokeOn.InvokeRequired)
  {
   return (TResult)controlToInvokeOn.Invoke(code);
  }
  else
  {
   return (TResult)code();
  }
}
```

Il s'agit d'ajouter à toutes les classes dérivées de Control (et à cette dernière aussi) la méthode "Invoke". Le class helper, tel que conçu ici, prend en charge le retour d'une valeur, ce qui est pratique si on désire lire la chaîne d'un textbox par exemple. Le premier paramètre "ne compte pas", il est le marqueur syntaxique des class helpers en quelque sorte. Le second paramètre qui apparaitra comme le seul et unique lors de l'utilisation de la méthode est de type Func<TResult>, il s'agit ici d'un prototype de méthode. Il sera donc possible de passer à Invoke directement un bout de code, voire une expression lambda, et de récupérer le résultat.

Un exemple d'utilisation : string value = this.Invoke(() => button1.Text);
Ici on va chercher la valeur de la propriété Text de "button1" via un appel à Invoke sur "this", supposée ici être la form. Le résultat est récupéré dans la variable "value". On note l'utilisation d'une expression lambda en paramètre de Invoke.

Mais si le code qu'on désire appeler ne retourne pas de résultat ? Le class helper, tel que défini ici, ne fonctionnera pas puisqu'il attend en paramètre du code retournant une valeur (une expression). Il est donc nécessaire de créer un overload de Invoke pour gérer ce cas particulier :

```
public static void Invoke(this Control controlToInvokeOn, Func code)
{
   if (controlToInvokeOn.InvokeRequired)
   {
      controlToInvokeOn.Invoke(code);
   }
   else
   {
      code();
   }
}
```

Avec cet overload la solution est complète et gère aussi bien le code retournant une valeur que le code "void".

On peut écrire alors: this.Invoke(() => progressBar1.Value = i);
Sachant que pour simplifier l'appel est ici effectué dans la form elle-même (this). L'appel à Invoke contient une expression lambda qui modifie la valeur d'une barre de progression. Mais peu importe les détails, c'est l'esprit qui compte.

Conclusion
Les class helpers peuvent fournir des solutions globales à des problèmes récurrents. Utilisés

dans un tel cadre ils prennent tout leur sens et au lieu de rendre le code plus complexe et moins maintenable, au contraire, il le simplifie et centralise sa logique.
L'utilisation des génériques, des prototypes de méthodes et des expressions lambda montrent aussi que les nouveautés syntaxiques de C#, loin d'être des gadgets dont on peut se passer forment la base d'un style de programmation totalement nouveau, plus efficace, plus sobre et plus ... générique. L'exemple étudié ici illustre parfaitement cette nouvelle façon de coder de C# 3.0 et les avantages qu'elle procure à ceux qui la maîtrise.

Les pièges de la classe Random

Générer des nombres aléatoires avec un ordinateur est déjà en soit ambigu : un PC est une machine déterministe (heureusement pour les développeurs et les utilisateurs !) ce qui lui interdit l'accès à la génération de suites aléatoires aux sens mathématique et statistique. Toutefois il s'agit d'un besoin courant et .NET propose bien entendu une réponse avec la classe Random.

L'ambigüité de Random

Random se présente comme une classe n'offrant aucune méthode statique. Dès lors le développeur comprend qu'il faut en créer une instance avant de l'utiliser. Oui mais cela n'a pas vraiment de sens et peut, de surcroit, être trompeur…

Quelques rappels indispensables

Posons d'abord que nous mettons hors de notre propos les utilisations "détournées" des faiblesses de Random. C'est-à-dire les applications (ou parties d'applications) qui se fondent sur le fait que Random retourne toujours une même suite pour une même "graine" (seed). Cela n'est pas spécifique à .NET, la majorité des langages et plateformes qui offrent un générateur de nombres aléatoires connaissent ce problème de suites "répétables", pour le comprendre il suffit de revenir à l'introduction de ce billet qui en explique le pourquoi…

Donc hors de ce cadre particulier qui exploite les faiblesses de Random, toute application consommatrice de nombres aléatoires suppose, au contraire, que le générateur est bien équilibré et que les suites retournées "ressemblent" suffisamment à de l'aléatoire pour être utilisées comme telles.

Rappelons que puisque ce n'est pas le cas, les applications réclamant de vraies suites aléatoires sont obligées de se reposer sur du hardware spécifique basé le plus souvent le bruit thermique ou des effets photo-électriques. Plus rares sont les montages utilisant des effets quantiques. Rappelons aussi que les langages ou plateformes comme .NET ne parlent pas de "générateur de nombres aléatoires" mais de générateurs de nombres "*pseudo-aléatoires*" ce qui traduit mieux leur réelle nature.

Enfin, pour être totalement précis, le générateur de nombres pseudo-aléatoires de .NET se fonde sur un algorithme bien spécifique, celui de générateur de nombres aléatoires soustractif de Donald E. Knuth tel que décrit dans “The art of computer programming, volume 2 : Seminumerical Algorithms”. Ouvrage passionnant (si si, il faut aimer les maths c’est tout) qu’on peut trouver chez Addison-Wesley.

Quel sens à l’utilisation de multiples instances ?

Revenons à nos moutons. Quel sens donner à l’utilisation de plusieurs instances de Random ? Une application doit-elle créer autant d’instances qu’elle a besoin de suites aléatoires ? Dans ce cas que peut-elle en attendre ?

De deux choses l’une : soit la qualité aléatoire de Random est insuffisante pour l’application considérée et ce n’est pas en multipliant les instances qu’on règlera le problème, soit elle est satisfaisante, et dans ce cas, aléatoire pour aléatoire la sortie d’une seule et unique instance de Random doit être aussi imprévisible que celle de 5 ou 10 autres instances…

Conséquemment, il semble ainsi déraisonnable d’utiliser plus d’une instance de Random dans une application.

La bonne pratique consiste ainsi à créer une variable statique de type Random, placée dans une classe de service accessible à toute l’application. L’instance peut être encapsulée dans une classe garantissant la concurrence d’accès si nécessaire. Si des dizaines de threads doivent maintenant accéder à une seule instance, cela peut créer un goulot d’étranglement et dans ce cas seulement il sera logique d’avoir autant d’instances que nécessaire (une par thread en l’occurrence, ni plus ni moins). La documentation de Random nous indique clairement qu’aucun membre d’une instance de Random n’est garantie être “thread safe”. Autant le savoir.

Pourquoi est-ce trompeur ?

L’utilisation de multiples instances de Random est trompeuse car le plus souvent on le fait en pensant disposer de séries aléatoires différentes. Dans l’idée ce n’est pas faux, mais dans la réalité il y a un problème si les instances sont créées très proches l’une de l’autre dans le temps.

Car la graine par défaut est dérivée de la valeur de l’horloge de la machine (de fait il est stupide de voir du code initialiser Random avec DateTime.Now.Milliseconds, puisque c’est ce que fait le constructeur par défaut…). Or, la graine, ainsi que l’'horloge, ont une résolution finie (on s’en douterait mais on n’y pense pas forcément). Il en découle que des instances de Random créées à très peu d’intervalle utiliseront en réalité une même valeur de graine ce qui impliquera des suites aléatoires rigoureusement identiques !

L'exemple de code suivant est issu de la documentation Microsoft et montre comment deux instances créées trop proches l'une de l'autre génèrent la même série :

```
byte[] bytes1 = new byte[100];
byte[] bytes2 = new byte[100];
Random rnd1 = new Random();
Random rnd2 = new Random();

rnd1.NextBytes(bytes1);
rnd2.NextBytes(bytes2);

Console.WriteLine("First Series:");
for (int ctr = bytes1.GetLowerBound(0);
     ctr <= bytes1.GetUpperBound(0);
     ctr++) {
   Console.Write("{0, 5}", bytes1[ctr]);
   if ((ctr + 1) % 10 == 0) Console.WriteLine();
}
Console.WriteLine();
Console.WriteLine("Second Series:");
for (int ctr = bytes2.GetLowerBound(0);
     ctr <= bytes2.GetUpperBound(0);
     ctr++) {
   Console.Write("{0, 5}", bytes2[ctr]);
   if ((ctr + 1) % 10 == 0) Console.WriteLine();
}
// The example displays the following output to the console:
//       First Series:
//          97  129  149   54   22  208  120  105   68  177
//         113  214   30  172   74  218  116  230   89   18
//          12  112  130  105  116  180  190  200  187  120
//           7  198  233  158   58   51   50  170   98   23
//          21    1  113   74  146  245   34  255   96   24
//         232  255   23    9  167  240  255   44  194   98
//          18  175  173  204  169  171  236  127  114   23
//         167  202  132   65  253   11  254   56  214  127
//         145  191  104  163  143    7  174  224  247   73
//          52    6  231  255    5  101   83  165  160  231
//
//       Second Series:
//          97  129  149   54   22  208  120  105   68  177
//         113  214   30  172   74  218  116  230   89   18
//          12  112  130  105  116  180  190  200  187  120
//           7  198  233  158   58   51   50  170   98   23
//          21    1  113   74  146  245   34  255   96   24
//         232  255   23    9  167  240  255   44  194   98
//          18  175  173  204  169  171  236  127  114   23
//         167  202  132   65  253   11  254   56  214  127
//         145  191  104  163  143    7  174  224  247   73
//          52    6  231  255    5  101   83  165  160  231
```

Conclusion

Bien se servir des outils mis à disposition par le Framework est essentiel. Les nombres aléatoires sont utilisés aussi bien pour tester un logiciel que pour créer des clés

(cryptographie) en passant par des données de mise au point de programme, etc. C'est au final un outil utilisé plus souvent qu'on ne le pense.

Il y aurait bien d'autres choses à dire sur le sujet, étudier finement la répartition statistique des nombres produits par Random, ou bien passer en revue les codes qui permettent de produire des suites normalisées (suivant une loi binomiale, exponentielle, gamma, normale, Pareto, Poisson, Weibull…). Ceux qui le désirent pourront creuser le sujet grâce à Binq !

A Dusty Net ! (*)
(*) anagramme de mon traditionnel "Stay Tuned !" choisi aléatoirement bien entendu :-)

Lire et écrire des fichiers XML Delphi TClientDataSet avec C#

Vous pouvez vous dire "quelle mouche le pique ?" A quoi bon en effet perdre son temps à bricoler des fichiers XML issus du TClientDataSet de Delphi puisque Delphi n'est plus qu'un souvenir (malgré quelques sursauts du côté d'Embarcadero à 4500 euros HT pour la version XE5, faut en vouloir !) et que le XML produit par le TClientDataSet est une curiosité ?

La chaleur de ce dimanche trop lourd peut-être ? Non, car des vieilleries en Delphi il en existe plein en circulation pour commencer (Delphi eut son heure de gloire il ne faut pas l'oublier, gloire méritée, même si la fin de l'histoire est un psychodrame), et que, comme je l'avais prédit il y a quelque années : "les delphistes seront (sont maintenant) les cobolistes des années 2010". On y est en 2010 (même en 2013 à la mise en page de ce PDF !), et effectivement tout comme les cobolistes des années 2000, une poignée de delphistes s'accroche à sa maigre connaissance en refusant obstinément d'évoluer (d'ailleurs vu les prix pratiqués par Embarcadero, la majorité des delphistes utilise toujours Delphi 5 ou 7, des produits qui ont dix ans voire plus…).

Utilité ?

Ce noyau dur d'irréductibles qui mourront en écrivant des begin/end produit encore des logiciels (peu il est vrai, les rares en poste font surtout de la maintenance de vieilles applis), qui parfois, sont utiles. On peut faire de mauvais choix pour ses outils mais fabriquer quelque chose d'utile, c'est presque paradoxal. Et il se peut que vous ayez à traiter de telles données et à vous taper tout le boulot, car un delphiste de 2013 refuse par essence toute espèce d'apprentissage et ce n'est pas lui qui vous fournira un fichier XML correctement formé…

Un exemple ? En dehors de nombreux softs de compta ou de gestion écrits avec Delphi, on trouve des choses comme Cumulus un logiciel de gestion de station météo. C'est écrit en Delphi, et je dirais même mieux "à la delphiste", c'est à dire que c'est un peu le boxon. Les

données sont éclatées en divers fichiers, certains sont des fichiers texte de type CSV, d'autres des fichiers INI (avec des champs qui sont parfois placés dans de mauvaises sections), et un fichier XML. Chouette ! Quand on l'ouvre : déception. Ce n'est pas un fichier XML bien formé, analysable facilement, c'est le fatras produit par le composant TClientDataSet. Je le reconnais au premier coup d'œil… Le contraire serait dommage pour quelqu'un qui a écrit trois livres sur Delphi lus et agréés par Borland malgré tout…
Comme un tel soft est utile dans sa partie connexion à la station et recueil des données, il serait idiot de réinventer la poudre. Mais comme l'organisation des données sent l'amateurisme à plein nez, difficile d'en faire quelque chose directement. Dans un tel cas, qui est valable pour ce soft et tous les softs Delphi de ce type, il va falloir concevoir un DAL C# capable de lire et de réécrire tous ces fichiers (exactement de la même façon car le code Delphi est généralement assez peu "blindé", donc ça pète si un octet n'est pas à sa place !).

Les fichiers XML issus du TClientDataSet utilisent ainsi une structure difficile à exploiter directement. En lecture quelques ruses permettent de trouver une parade, en écriture cela devient plus sportif, le moindre écart avec ce qui est attendu et le soft Delphi ne pourra pas relire le fichier.

Phase 1 : lire un XML TClientDataSet

La ruse est facile, .NET propose de longue date un composant assez proche, le DataSet. Seules "petites" différences : il produit un XML correctement formé, sait interpréter les schémas XSD et le "Set" de DataSet n'est pas usurpé puisque le DataSet est une mini base de données à lui tout seul capable de stocker plusieurs tables et leurs relations (le TClientDataSet ne travaille que sur une seule table). le "Set" (ensemble) se justifie parce que le TClientDataSet sait enregistrer un ensemble d'enregistrements (et non un ensemble de tables comme le Dataset .NET). Encore heureux, car un format qui ne sauvegarderait qu'un seul enregistrement et non un ensemble ne serait guère utile…

Dès lors, lire un fichier XML TClientDataSet peut s'effectuer directement de la façon suivante :

```
1: sXMLFileName = openFileDialog1.FileName;
2: aDS = new DataSet();
3: aDS.ReadXml(sXMLFileName);
4: dataGrid.DataSource = aDS;
5: dataGrid.DataMember = "ROW";
```

On suppose ici un dialogue d'ouverture de fichier retournant le nom du fichier XML à ouvrir. On créé le DataSet, on lit le fichier en mémoire, puis on connecte le DataSet à une DataGrid. On utilise le DataMember "ROW" qui plonge à l'intérieur du fichier XML sur la collection d'enregistrements.

C’est beau, ça marche. Si c’est juste pour exploiter le fichier en lecture, c’est donc très simple.

Sauf… sauf qu’il existe un petit problème de codage. Par exemple le fichier issus de Cumulus contient parfois des codes XML remplaçant certains caractères mais laisse certains autres non traduits (les accentuées par exemple).

Le fichier est d’ailleurs refusé par un logiciel comme XmlSpy en raison de l’incohérence entre le codage et le contenu. Il faut dire que le codage n’est pas indiqué dans l’entête du fichier… Dans XmlSpy il suffit de dire que le mode par défaut est ANSI et on peut ouvrir le fichier. C’est bien gentil, mais si c’est pour passer par XmlSpy, l’intérêt de notre ruse pour utiliser le fichier directement en C# n’a plus beaucoup d’intérêt…

Heureusement pour nous, le Framework .NET est riche et souple. Il faut ainsi lire le fichier XML non pas directement dans le DataSet mais depuis un flux à qui on indiquera le codage à utiliser:

```
1: sXMLFileName = openFileDialog1.FileName;
2: aDS = new DataSet();
3: var rd = new StreamReader(sXMLFileName, Encoding.Default);
4: aDS.ReadXml(rd);
5: rd.Close();
6: ...
```

Il faut ainsi passer par un StreamReader. Mais lorsqu’on regarde les options de l’énumération Encoding, on ne trouve pas de ANSI ! Pas de panique, c’est le mode Default qu’il faut utiliser et qui correspond à ANSI.

Et voilà pour la lecture !

Phase 2 : Sauvegarder le fichier sans rien casser.

Intermédiaire

La phase 2 implique une phase intermédiaire, la création d’un schéma XSD. Car si on tente de modifier le fichier tel qu’il est ouvert dans la Phase 1, les enregistrements qu’on pourra ajouter à la collection “ROW” vont se retrouver sous la racine du XML, avant ou après la vraie collection “ROW”, autant dire que le logiciel Delphi va planter illico à l’ouverture ! (Cumulus renvoie bien un message permettant de signaler l’erreur et de continuer, c’est un bel effort, mais le fichier n’est plus lu et toutes les données sont perdues).

Créer un schéma XSD à la main est assez enquiquinant, et pour un dimanche, un peu lourd qui plus est, c’est au dessus de mes forces.

Heureusement, XmlSpy sait inférer un schéma à partir d'un fichier XML. Comme tout le monde ne possède pas cet outil assez indispensable pourtant, voici le XSD à utiliser :

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="ROWDATA">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ROW" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="ROW">
        <xs:complexType>
            <xs:attribute name="SnowLying" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="FALSE"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="SnowFalling" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="FALSE"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="SnowDepth" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:byte">
                        <xs:enumeration value="0"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="RowState" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:byte">
                        <xs:enumeration value="4"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="EntryDate" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:int">
                        <xs:enumeration value="20100620"/>
                        <xs:enumeration value="20100710"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="Entry" use="required" type="xs:string"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="PARAMS">
        <xs:complexType>
            <xs:attribute name="CHANGE LOG" use="required">
                <xs:simpleType>
```

```
55:                     <xs:restriction base="xs:string">
56:                         <xs:enumeration value="1 0 4 2 0 4"/>
57:                     </xs:restriction>
58:                 </xs:simpleType>
59:             </xs:attribute>
60:         </xs:complexType>
61:     </xs:element>
62:     <xs:element name="METADATA">
63:         <xs:complexType>
64:             <xs:sequence>
65:                 <xs:element ref="FIELDS"/>
66:                 <xs:element ref="PARAMS"/>
67:             </xs:sequence>
68:         </xs:complexType>
69:     </xs:element>
70:     <xs:element name="FIELDS">
71:         <xs:complexType>
72:             <xs:sequence>
73:                 <xs:element ref="FIELD" maxOccurs="unbounded"/>
74:             </xs:sequence>
75:         </xs:complexType>
76:     </xs:element>
77:     <xs:element name="FIELD">
78:         <xs:complexType>
79:             <xs:attribute name="fieldtype" use="required">
80:                 <xs:simpleType>
81:                     <xs:restriction base="xs:string">
82:                         <xs:enumeration value="boolean"/>
83:                         <xs:enumeration value="date"/>
84:                         <xs:enumeration value="i4"/>
85:                         <xs:enumeration value="string"/>
86:                     </xs:restriction>
87:                 </xs:simpleType>
88:             </xs:attribute>
89:             <xs:attribute name="attrname" use="required">
90:                 <xs:simpleType>
91:                     <xs:restriction base="xs:string">
92:                         <xs:enumeration value="Entry"/>
93:                         <xs:enumeration value="EntryDate"/>
94:                         <xs:enumeration value="SnowDepth"/>
95:                         <xs:enumeration value="SnowFalling"/>
96:                         <xs:enumeration value="SnowLying"/>
97:                     </xs:restriction>
98:                 </xs:simpleType>
99:             </xs:attribute>
100:            <xs:attribute name="WIDTH">
101:                <xs:simpleType>
102:                    <xs:restriction base="xs:short">
103:                        <xs:enumeration value="1024"/>
104:                    </xs:restriction>
105:                </xs:simpleType>
106:            </xs:attribute>
107:        </xs:complexType>
108:    </xs:element>
109:    <xs:element name="DATAPACKET">
110:        <xs:complexType>
111:            <xs:sequence>
112:                <xs:element ref="METADATA"/>
```

```
113:                 <xs:element ref="ROWDATA"/>
114:             </xs:sequence>
115:             <xs:attribute name="Version" use="required">
116:                 <xs:simpleType>
117:                     <xs:restriction base="xs:decimal">
118:                         <xs:enumeration value="2.0"/>
119:                     </xs:restriction>
120:                 </xs:simpleType>
121:             </xs:attribute>
122:         </xs:complexType>
123:     </xs:element>
124: </xs:schema>
```

Ce XSD est “universel” il fonctionne pour tous les XML issus d’un TClientDataSet (enfin normalement, dites-le-moi si vous trouvez des exceptions).
[EDIT] Bien entendu quand je dis "universel" c'est la structure globale... le schéma de la table proprement-dit dépend ... de la table et change selon le cas. Ca me semblait évident mais en relisant le post je me suis aperçu que cela ne l'était pas forcément.[/EDIT]
Grâce à ce schéma nous allons pouvoir lire le fichier XML de façon plus précise, le DataSet s’y retrouvant mieux visiblement.

La lecture définitive devient donc :

```
1: sXMLFileName = openFileDialog1.FileName;
2: aDS = new DataSet();
3: aDS.ReadXmlSchema(Path.ChangeExtension(sXMLFileName,".xsd"));
4: var rd = new StreamReader(sXMLFileName, Encoding.Default);
5: aDS.ReadXml(rd);
6: rd.Close();
```

On suppose ici que le fichier XSD porte le même nom que le fichier XML à lire, avec l’extension “.xsd” au lieu de ‘”.xml”.

Mais à quoi ressemble un fichier XML du TClientDataSet ?
Il est vrai que tant qu’on se contentait de lire, et puisque nous avions trouvé une astuce, la question de savoir comment est réellement fait un tel fichier n’avait guère d’intérêt, sauf pour des archéologues pointilleux du genre à déterrer une dent qui a 3000 ans et l’analyser pour savoir que le type à qui elle appartenait avait manger des carottes dans l’année précédant sa mort. Très utile (sans rire, l’archéologie est essentielle, mais je trouve qu’elle vire parfois à la maniaquerie de psychopathe à tout vouloir déterrer, le moindre fragment de vase, de dent, surtout pour les périodes récentes. Savoir qu’un romain mangeait des carottes ou que les grecs se lavaient les pieds dans des pédiluves ronds, je vois mal l’intérêt, trouver Lucy ou les premiers dinosaures à plume en a un à l’inverse. Mais c’est un point de vue personnel).

Bref, il faut comprendre comment marche ces fichus fichiers XML.

D'abord il faut savoir qu'ils ne possèdent pas de changement de ligne. Une économie un peu mesquine comparée à l'avantage de pouvoir les lire facilement avec le bloc-notes, mais c'est comme ça. Donc il faut utiliser un outil capable de remettre tout ça en forme pour y voir quelque chose (XmlSpy, encore lui, sait le faire. Je n'ai pas d'action chez Altova je précise).

```
 1: <?xml version="1.0" standalone="yes"?>
 2: <DATAPACKET Version="2.0">
 3:     <METADATA>
 4:         <FIELDS>
 5:             <FIELD attrname="EntryDate" fieldtype="date"/>
 6:             <FIELD attrname="Entry" fieldtype="string" WIDTH="1024"/>
 7:             <FIELD attrname="SnowLying" fieldtype="boolean"/>
 8:             <FIELD attrname="SnowFalling" fieldtype="boolean"/>
 9:             <FIELD attrname="SnowDepth" fieldtype="i4"/>
10:         </FIELDS>
11:         <PARAMS CHANGE_LOG="1 0 4 2 0 4"/>
12:     </METADATA>
13:     <ROWDATA>
14:         <ROW RowState="4" EntryDate="20100620" Entry="entrée 1 de test"
SnowLying="FALSE" SnowFalling="FALSE" SnowDepth="0"/>
15:         <ROW RowState="4" EntryDate="20100710" Entry="entrée 2 de test"
SnowLying="FALSE" SnowFalling="FALSE" SnowDepth="0"/>
16:     </ROWDATA>
17: </DATAPACKET>
```

Le fichier contient un entête très succinct puis une racine DATAPACKET qui contient plusieurs sections. On trouve METADATA qui représente le schéma de la table, on trouve aussi ROWDATA une collection de ROW qui sont les vrais enregistrements, mais aussi PARAMS avec un attribut CHANGE_LOG suivi de plein de chiffres.

Normalement, si le programme Delphi est bien écrit (rareté) un appel à la validation des changements devrait être fait avant la sauvegarde (méthode MergeChangeLog du TClientDatSet). Si tel n'est pas le cas, comme ici, le fichier XML va conserver l'historique de tous les changements. C'est la section PARAMS avec son CHANGE_LOG qui va grossir inutilement avec le temps. C'est malin d'économiser des octets en ne mettant pas de changement de ligne et de perdre autant de place faute de comprendre comment marche le composant TClientDataSet ! C'est tout Delphi et les delphistes ça… A noter que si une clé primaire avait été définie (ce qui n'est pas le cas ici, encore un laxisme) le nom du champ serait indiqué dans cette section METADATA (PRIMARY KEY), idem pour le tri par défaut '"DEFAULT_ORDER".

Notre but n'étant pas d'aller à la maniaquerie évoquée plus haut, tenons-nous en à ce qui est utile pour nous : reproduire cette structure.

Il va donc falloir reproduire le CHANGE_LOG dans ce cas puisqu'il est présent. Sa structure est simple mais elle ne se devine pas au premier coup d'œil, ce sont des triplets :

- le premier chiffre indique le numéro de ligne
- le second est le numéro de version de la précédente modification (s'il y en a)
- le troisième vaut 4 pour un ajout, 8 pour une modification.
- Nous n'irons pas chercher plus loin car lorsque nous sauvegarderons nous recréerons une structure de ce type en considérant que toutes les lignes sont des ajouts et qu'elles n'ont pas de version précédente. De fait la ligne 1 aura pour triplet 1 0 4, la ligne 2 : 2 0 4, etc.

Le DataSet nous joue des tours

Il est bien ce DataSet, on peut faire plein de choses, même lire des données aussi biscornues que celles d'un TClientDataSet Delphi ! Mais quand on sauvegarde, il ajoute son petit grain de sel, bien naturel pour un fichier XML bien formé : le nom du dataSet lui-même comme racine. On se retrouve ainsi avec un contenu entouré par la balise <NewDataSet>contenu</NewDataSet>. "NewDataSet" est le nom par défaut d'un DataSet. Ca peut donc être autre chose si vous avez nommé le DataSet.

Donc, pour sauvegarder les données il faudra d'une part recréer le CHANGE_LOG dans notre cas, et supprimer, dans tous les cas, la balise supplémentaire.

La Sauvegarde, enfin !

On y est ! Il est maintenant possible de reproduire la structure et les changements (ajouts, suppression de lignes, etc) tout en faisant en sorte que le logiciel Delphi puisse relire le fichier XML sans se douter que nous sommes passés par là ! (et c'est préférable si on ne veut pas planter le soft Delphi !).

```
 1: var lines = aDS.Tables["ROW"].Rows.Count;
 2: var sb = new StringBuilder();
 3: for (var i = 0; i < lines; i++) sb.Append((i + 1) + " 0 4 ");
 4: var logs = sb.ToString().Trim();
 5: aDS.Tables["PARAMS"].Rows[0][0] = logs;
 6: var sdn = aDS.DataSetName.ToUpper();
 7: var wd = new StreamWriter(sXMLFileName, false, Encoding.Default);
 8: wd.WriteLine(@"<?xml version=""1.0"" standalone=""yes""?>");
 9: wd.NewLine = Environment.NewLine;
10: aDS.WriteXml(wd, XmlWriteMode.IgnoreSchema);
11: wd.Close();
12: var li = File.ReadAllLines(sXMLFileName);
13: var ls = new List<string>(li.Count());
14: foreach (var s in li)
15: {
16:     if (s.ToUpper().Contains("<" + sdn)) continue;
17:     if (s.ToUpper().Contains("</" + sdn)) continue;
18:     ls.Add(s);
19: }
```

```
20: File.WriteAllLines(sXMLFileName, ls);
```

Au départ on compte les lignes de la table “ROW” du DataSet. On s’en sert pour construire une chaîne constituée des fameux triplets du CHANGE_LOG (ligne 3). On stocke la chaine au bon endroit (ligne 5). En ligne 6 on prend note du nom du DataSet (comme cela on n’a pas se soucier de sa valeur, NewDataSet par défaut, mais sait-on jamais…).

Lignes 7 à 11 on utilise l’astuce de la lecture dans l’autre sens, avec un StreamWriter. On notera aussi qu’en ligne 8 on ajoute en début de fichier le marquage utilisé par le fichier Delphi (l’entête xml indiquant notamment le mode stand alone).

Le fichier est maintenant sauvegardé, mais avec une balise de trop (NewDataSet). Il faut une seconde passe (pas très économique j’en conviens surtout si le fichier est gros) qui est réalisée aux lignes 12 à 20. En fait on lit le fichier en mémoire en sautant les deux balises. Ensuite on écrase le fichier disque par cette nouvelle version. Brutal, un chouia goret comme méthode, mais le but est montrer ce qu’il faut mettre dans le fichier XML. A vous d’écrire ça de façon plus propre si nécessaire… A noter : la seconde passe fonctionne parce que le fichier XML produit par le DataSet a des sauts de ligne, s’il n’y en avait pas il faudrait utiliser une autre stratégie pour supprimer les balises gênantes.

Conclusion

Lire et écrire des fichiers XML issus d’un TClientDataSet Delphi n’est vraiment pas une tâche agréable. Mais cela fait partie du job d’un développeur de faire avec les données qu’on lui donne et qu’il n’a pas choisies.

Pas exaltant mais utile.

On ne peut pas se marrer tous les jours en parlant des dernières nouveautés de Microsoft, parfois ressurgissent des profondeurs des monstres d’un autre temps avec lesquels il faut bien composer ?

PropertyChanged sur les indexeurs

Voici un court sujet pour cette rentrée (et surtout pour me remettre du pavé de 92 pages sur MVVM et le toolkit MVVM Light !). En effet, ce bon Anders Hejlsberg, en repompant certaines bonnes idées de Delphi qu’il avait créé quelques années avant C#, a oublié certaines choses pourtant utiles comme les indexeurs nommés. En C# un seul indexeur par classe et pas de nom ! Choix curieux, étrange, peu judicieux, et au bout de tant de temps jamais modifié. Bref un seul indexeur, et sans nom. Mais lors d’un Binding Xaml comment diable prévenir les objets bindés que les valeurs de l’indexeur ont changé (quand ce cas se présente) ?

Beaucoup ne font pas, comme ça l'affaire est réglée, et du coup ils se privent de tout l'intérêt du binding qui est vivant grâce à INotifyPropertyChanged…

Avec MVVM où le binding tient un rôle central, il est encore plus capital d'apporter une solution à ce problème.

Et ce n'est pas une solution mais TROIS que je vais vous proposer. Des approches similaires mais dont les variations sont intéressantes.

Nommer l'indexeur !

Le vrai problème c'est que l'indexeur unique n'a pas de nom… Il suffit donc de lui en donner un ! Riche idée ! Comment pratiquer ? Grâce à l'attribut "IndexerName" avec lequel on peut décorer l'indexeur.

On ne rêve pas, cela ne permettra pas vraiment de donner un nom utilisable à l'indexeur, mais en revanche ce nom sera reconnu dans un PropertyChanged et les bindings liés seront bien mis à jour. C'est peu, mais c'est énorme.

```
 1: [IndexerName("Data")]
 2: public string this[string key]
 3: {
 4:     get { return xxxx[key]; }
 5:     set
 6:     {
 7:         if (value==xxxx[key]) return;
 8:         xxxx[key] = value;
 9:         RaisePropertyChanged("Data[]"); // Ruse !
10:     }
11: }
```

Nommer l'indexeur est donc un moyen simple pour disposer d'un nom de propriété utilisable dans un ProperyChanged. Attention, dans ce dernier il faut bien passer le nom de la propriété avec les accolades ("Data" devient "Data[]" dans l'exemple) !

Le Nom par défaut

En fait il semble que le compilateur (ou la plateforme plus certainement) donne le nom "Item[]" par défaut à l'indexeur. De fait on peut se passer de l'attribut et utiliser comme nom de propriété "Item[]".

Je me méfie de cette solution qui oblige à utiliser "en dur" un nom surgit de nulle part et qui peut changer sans prévenir.

La constante de nom d'indexeur

Se passer dans l'attribut est une bonne idée, moins on écrit de code, moins il y en a à maintenir ! Mais franchement, le nom en dur je n'aime pas ça. Heureusement, il existe

depuis la version 3.0 une constante, passée un peu inaperçue, qui retourne le nom de l'indexeur, ce qui évite d'utiliser "Item[]". Cette constante c'est Binding.IndexerName définie dans le namespace System.Windows.Data (de PresentationFramework.dll).

Du coup il est possible d'écrire ceci :

```
 1: public string this[string key]
 2: {
 3:     get { return _items[key]; }
 4:     set
 5:     {
 6:         _items[key] = value;
 7:
 8:         if (PropertyChanged != null)
 9:             PropertyChanged(this, new PropertyChangedEventArgs(Binding.IndexerName));
10:     }
11: }
```

Mais il y a un "Hic". Cette constante ne semble pas être supportée par Silverlight pour qui les autres astuces restent donc d'actualité.

Conclusion

Notifier les changements de valeurs d'un indexeur peut rendre d'immenses services (pensez à un système de localisation dont on peut changer la langue à l'exécution, des données issues de capteurs qui varient dans le temps, etc).

Encore fallait-il connaître l'astuce. C'est chose faite !

Intégrité bi-directionnelle. Utiliser IEnumerable et des propriétés read-only (C#)

Un peu de C#, ça faisait longtemps que je n'avais pas bloggé sur le sujet. Aujourd'hui quelques points essentiels dans la conception des classes…

Relations entre entités et invariants : un exemple

Pour éviter de trop nous perdre dans les méandres des explications, le plus simple est de regarder directement le code ci-dessous :

```
public class Customer
{
    public string FirstName { get; set; }
```

```
    public string LastName { get; set; }
    public string Province { get; set; }
    public List<Order> Orders { get; set; }

    public string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}

public class Order
{
    public Order(Customer customer)
    {
        Customer = customer;
        customer.Orders.Add(this);
    }

    public Customer Customer { get; set; }
}
```

La question est : qu'est-ce qui ne va pas avec ces deux classes ?

Setter sur une collection

La première chose qui ne va pas est la présence d'un setter sur la collection Orders de la classe Customer. On ne fait jamais ça (mais on le voit hélas souvent). N'importe qui peut remplacer l'objet collection lui-même et couper l'herbe sous les pieds de Customer. Ici l'exemple est simpliste, mais imaginez que Customer gère des événements propres à la collection…

Rendre le setter d'une collection publique, voire tout simplement lui adjoindre un tel setter quel que soit sa visibilité est le plus généralement une grosse erreur de conception.

Publier une liste concrète

Second problème toujours posé par cette liste Orders : elle est visible sous la forme de son implémentation concrète, à savoir List<Order>. Que se passera-t-il si pour faire évoluer notre code dans le futur nous souhaitons utiliser une ObservableCollection ou toute autre structure à la place de List<T> ? Tout le code dépendant de Customer sera à revoir !

On ne fait jamais cela non plus. Les collections de ce type, sauf obligation dûment commentée et justifiée, sont toujours publiées sous forme d'interfaces, par exemple IList<T>.

Relation bi-directionnelle instable.
Il est évident à la lecture du code de Order que le constructeur de cette classe établit une relation bi-directionnelle avec Customer. Publier un setter pour la propriété Customer est une erreur, n'importe quel code pourra modifier le client attaché à la commande de façon anarchique ce qui laissera l'ensemble des données dans un bel état !

La solution

Voici comment les deux classes pourraient être corrigées pour éviter les problèmes indiqués :

```
public class Customer
{
    private readonly IList<Order> _orders = new List<Order>();

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Province { get; set; }
    public IEnumerable<Order> Orders { get { return _orders; } }

    public string GetFullName()
    {
        return LastName + ", " + FirstName;
    }

    internal void AddOrder(Order order)
    {
        _orders.Add(order);
    }
}

public class Order
{
    public Customer Customer { get; protected set; }

    public Order(Customer customer)
    {
        Customer = customer;
```

```
            customer.AddOrder(this);
        }
}
```

Le setter sur la collection a été supprimé, la liste est publiée sous la forme d'une interface IEnumerable<Order> et le constructeur de Order oblige à passer un Customer et c'est ce constructeur qui établit de façon définitive le lien bi-directionnel entre les entités. De même la propriété Customer de Order possède désormais un setter "protected" évitant toute manipulation depuis l'extérieur.

Regardons de plus près
Il y a ici un autre invariant qui a été pris en compte : c'est le fait qu'une commande ne peut pas exister sans client. Le constructeur de Order donne corps à cet invariant puisqu'il n'est pas possible de créer une commande sans passer un Customer et que Order s'occupe d'appeler les méthodes de Customer pour assurer le lien bi-directionnel.

Imaginons un instant qu'après d'âpres discussions au sommet entre experts, il soit décidé qu'une commande ne peut pas non plus exister sans lignes de commandes.

Cela ne pose pas trop de problèmes puisque nous savons maintenant comment résoudre la question : en ajoutant un nouveau paramètre au constructeur de Order pour qu'une liste de lignes de commandes soit passée. On ajoutera aussi les appels nécessaires pour que le lien bi-directionnel soit assuré.

C'est là que tout cela pose un problème de conception… Parce que si notre API semble parfaite vue de l'extérieur, à l'intérieur ça va commencer à se compliquer et à devenir plus difficile à maintenir à force d'ajouter des paramètres, des appels à des méthodes ici et là…

Méthodes d'exentions, génériques, expressions Lambda et Reflexion

En voici une belle brochette !

On se met à rêver quelques instants et on se demande si devant le fatras qui s'annonce dans les classes ainsi modifiées on ne ferait pas mieux de prévoir tout cela dès le départ et de se créer une petite boite à outils versatile pour régler une bonne fois pour toute les problèmes soulevés…

Par exemple il serait vraiment sympa de pouvoir ajouter un item à un IEnumerable depuis une entité en relation sans avoir besoin d'appeler des méthodes internes, non ? Il pourrait être pratique de pouvoir modifier des propriétés protégées (non pas privées, là ça serait pousser le bouchon trop loin). Existe-t-il un moyen de créer une infrastructure simplifiant

ensuite la prise en charge des problèmes soulevés à la fois par la première implémentation de Customer et Order et par l'implémentation de la solution qui en soulève d'autres ?

Comme nous ne souhaitons pas exposer des fonctionnalités aussi dangereuses que celles évoquées dans toute notre application, il s'agit juste de nous aider à créer des implémentations "propres" tout en préservant la clarté de notre API, nous allons ajouter tout cela dans notre *Layer Supertype*.

Layer Supertype ?

Dans "Patterns of Enterprise Application Architecture" de Martin Fowler, la description de cette pattern est donnée page 475. Comme il n'existe pas de version française, le numéro de page devrait être le bon si vous possédez cet indispensable ouvrage chez vous…
En gros il n'est pas idiot pour tous les objets dans un même layer de posséder des méthodes que vous ne voulez pas dupliquer dans tout le système. Dans ce cas vous pouvez toutes les déplacer dans un Layer Supertype commun, une classe spécifique du layer en question qui regroupe donc tous les comportements utilisables ici et là dans le layer mais uniquement dans ce layer.

La lecture du livre de Fowler vous en dira bien plus que quelques lignes ici. C'est un pattern très intéressant (comme le reste du bouquin d'ailleurs) mais que je ne peux pas traiter en profondeur dans ce billet.

Les méthodes du SuperType

```
protected void SetInaccessibleProperty<TObj, TValue>(TObj target, TValue value,
    Expression<Func<TObj, TValue>> propertyExpression)
{
    propertyExpression.ToPropertyInfo().SetValue(target, value, null);
}

protected TValue GetInaccessibleProperty<TObj, TValue>(TObj target,
    Expression<Func<TObj, TValue>> propertyExpression)
{
    return (TValue)propertyExpression.ToPropertyInfo().GetValue(target, null);
}

protected void AddToIEnumerable<TEntity, TValue>(TEntity target, TValue value,
    Expression<Func<TEntity, IEnumerable<TValue>>> propertyExpression)
{
    IEnumerable<TValue> enumerable = GetInaccessibleProperty(target,
propertyExpression);
```

```
    if (enumerable is ICollection<TValue>)
        ((ICollection<TValue>)enumerable).Add(value);
    else
        throw new ArgumentException(
            string.Format("Property must be assignable to ICollection<{0}>",
typeof(TValue).Name));
}

protected void RemoveFromIEnumerable<TEntity, TValue>(TEntity target, TValue
value,
    Expression<Func<TEntity, IEnumerable<TValue>>> propertyExpression)
{
    IEnumerable<TValue> enumerable = GetInaccessibleProperty(target,
propertyExpression);

    if (enumerable is ICollection<TValue>)
        ((ICollection<TValue>)enumerable).Remove(value);
    else
        throw new ArgumentException(string.Format("Property must be assignable to
ICollection<{0}>",
            typeof(TValue).Name));
```

Il est sûr que vu comme ça, au petit déjeuner, ça peut sembler un peu indigeste. Mais relisez ce code au calme, vous verrez ça a du sens :-)

Surtout, ce code va nous service à construire quelque chose de plus intelligent :

```
protected void AddManyToOne<TOne, TMany>(
    TOne one, Expression<Func<TOne, IEnumerable<TMany>>> collectionExpression,
    TMany many, Expression<Func<TMany, TOne>> propertyExpression)
{
    AddToIEnumerable(one, many, collectionExpression);
    SetInaccessibleProperty(many, one, propertyExpression);
}

protected void RemoveManyToOne<TOne, TMany>(
    TOne one, Expression<Func<TOne, IEnumerable<TMany>>> collectionExpression,
    TMany many, Expression<Func<TMany, TOne>> propertyExpression)
    where TOne : class
{
```

```
    RemoveFromIEnumerable(one, many, collectionExpression);
    SetInaccessibleProperty(many, null, propertyExpression);
}

protected void RemoveManyToMany<T1, T2>(
    T1 entity1, Expression<Func<T1, IEnumerable<T2>>> expression1,
    T2 entity2, Expression<Func<T2, IEnumerable<T1>>> expression2)
{
    RemoveFromIEnumerable(entity1, entity2, expression1);
    RemoveFromIEnumerable(entity2, entity1, expression2);
}

protected void AddManyToMany<T1, T2>(
    T1 entity1, Expression<Func<T1, IEnumerable<T2>>> expression1,
    T2 entity2, Expression<Func<T2, IEnumerable<T1>>> expression2)
{
    AddToIEnumerable(entity1, entity2, expression1);
    AddToIEnumerable(entity2, entity1, expression2);
}
```

Déjà on commence à voir l'intérêt de la manœuvre. Cette "seconde couche" exploite les premières méthodes pour autoriser des comportements de plus haut niveau, notamment l'ajout et la suppression d'éléments à des IEnumerable sans avoir accès aux implémentations concrètes !

Vous noterez que toutes les méthodes sont "protected" donc uniquement utilisable dans les classes dérivées, celles du layer en cours qui descendent donc toutes du Layer SuperType…

En réalité il y a un petit morceau qui fait exception et qui est tellement pratique qu'il a été transformé en méthode d'extension :

```
public static class ExpressionExtensions
{
    public static PropertyInfo ToPropertyInfo(this LambdaExpression expression)
    {
        var prop = expression.Body as MemberExpression;

        if (prop != null)
        {
            var info = prop.Member as PropertyInfo;
            if (info != null)
```

```
                return info;
            }

            throw new ArgumentException("The expression target is not a Property");
        }
    }
```

Ce code est très simple, par le biais de la Reflection il permet d'atteindre une propriété passée sous la forme d'une expression Lambda et de modifier son contenu. C'est un peu tordu mais c'est très utile, vous allez le voir dans l'exemple ci-dessous.

La solution améliorée par le SuperType et ses méthodes

Le SuperType s'appelle DomainBase, une convention que chacun pourra utiliser ou non (mais après avoir lu le livre de Fowler !).

```
public class Customer : DomainBase
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Province { get; set; }
    public IEnumerable<Order> Orders { get; protected set; }

    public string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}

public class Order : DomainBase
{
    public Customer Customer { get; protected set; }

    public Order(Customer customer)
    {
        AddManyToOne(customer, x => x.Orders, this, x => x.Customer);
    }
}
```

La nouvelle implémentation a tout de même fière allure ! Elle est plus simple à lire, son API est aussi claire que son contenu et nous avons préparé le terrain avec le SuperType pour régler de façon aussi élégante les mêmes problèmes dans tout le layer concerné !

Conclusion

Comme toute solution générique démontrée sur un pauvre ensemble de deux mini classes, forcément, on semble utiliser un tank pour tuer une mouche.

Si votre logiciel ne contient qu'une classe Customer et qu'une classe Order comme ici, alors nous ne parlons pas du même type de logiciel et je vous présente mes excuses.

Bien entendu, pour tous les autres lecteurs, nous savons tous que ce type de solution ne prend son intérêt que dans de larges solutions exposants de dizaines voire des centaines de classes.

La solution présentée ici a un impact sur les performances, c'est certain. De combien ? Je n'ai pas mesuré. Mais si faire la balance entre performances brutes et code maintenable est toujours quelque chose de grave et délicat, j'opte systématiquement pour la maintenabilité et la clarté du code. D'autres développeurs préféreront dupliquer du code partout dans le fol espoir de gratter quelques millisecondes. Chacun ses choix, cela ne me dérange pas. Mais je serai curieux de voir alors combien de lignes dupliquées contiendra ce code et au final combien de clients et de commandes un tel code sera capable de traiter réellement par secondes…

Bref, si l'idée de Fowler du SuperType est à exploiter sans trop de contrainte, les transformations à outrance présentées ici mélangeant Expression Lambda, Reflexion, méthodes d'extensions le tout à la sauce générique sont plutôt à prendre comme des possibilités et des idées d'implémentation ouvrant de nouvelles façons de concevoir des couches riches en classes (un BOL, un DAL par exemple). Tout n'est peut-être pas à prendre au pied de la lettre et le but du jeu était principalement de vous faire réfléchir à ces possibilités.

Si ce but est atteint alors c'est une bonne chose !

PS: j'ai donné les coordonnées du livre de Fowler par l'hyperlien qu'il suffit de suivre (ça tombe chez Amazon France pour le commander). Concernant le billet lui-même je me suis inspiré d'une publication de Sean Blakemore, non par paresse ou manque d'idée mais parce qu'il me semblait que son propos valait largement l'intérêt d'être proposé dans notre belle langue pour en faire profiter tous les lecteurs de Dot.Blog qui, je le sais, ne sont pas tous des amis de l'anglais…

Random et bonnes pratiques

Générer des nombres aléatoires a toujours été un casse-tête pour nos pauvres ordinateurs totalement déterministes. Le Framework .NET nous offre quelques solutions encore faut-il en connaitre les limites et s'en servir correctement….

Nombres aléatoires ?

Définissons d'abord ce qu'est un nombre aléatoire : cela n'existe pas 

En effet, seule **une série de nombres** peut, *éventuellement*, se voir qualifier d'aléatoire. Un nombre pris seul, de façon totalement isolé n'est ni aléatoire ni non aléatoire, cela est indéterminable.
Ensuite une telle série, pour mériter le qualificatif d'aléatoire doit répondre à des exigences mathématiques précises. Sans entrer dans les méandres des théories que vous trouverez aisément sur le Web, il faut être convaincu qu'aucun procédé algorithmique, aussi sophistiqué ou "rusé" soit-il ne peut générer une suite de nombres aléatoires.

Au mieux, il s'agira d'une suite de nombre **pseudo aléatoires** répondant à certains critères (c'est à dire simulant au plus proche certaines courbes ou lois comme celle de Poisson ou d'autres formes de distribution).
Un ordinateur étant une machine déterministes (même si certains bugs peuvent parfois nous en faire douter !) l'aléatoire est totalement hors du champ de ses possibilités, quelle que soit sa taille, sa puissance, sa mémoire ou la présence d'un coprocesseur mathématique (dont on ne parle plus depuis quelques années puisque systématiquement intégré aux CPU ce qui ne fut pas le cas pendant longtemps).

Les nombres aléatoires ne peuvent ainsi être générés qu'en s'appuyant sur des phénomènes physiques eux-mêmes réputés aléatoires. C'est pour cela que pour générer des vraies séries de nombres aléatoires sur un ordinateur il faut absolument utiliser un hardware spécifique. Il en existe de différentes nature selon le degré de précision dans l'aléatoire qu'on désire (s'il est possible de parler de précision ici). Ces boitiers qui peuvent se brancher sur un port USB par exemple, utilisent des phénomènes quantiques qu'ils amplifient et numérisent pour obtenir des nombres : bruit thermique d'une résistance en général.

Donc hors de ces hardwares, parler de nombres aléatoires avec un ordinateur est un abus de langage dans le meilleur des cas et une hérésie mathématique dans le pire…

La classe Random

Tout le monde la connait, c'est le moyen le plus simple d'obtenir des nombres pseudo aléatoires sous .NET.

Mais de ce que je peux constater lorsque j'audite du code, cette classe est mal utilisée dans la grande majorité des cas.

Erreur n°1 – Initialisation avec l'heure

A vouloir trop bien faire sans savoir ce qui se cache derrière une classe, on fait des bêtises ou du code inutile, voire les deux à la fois.

La classe Random, lorsqu'elle est instanciée utilise déjà l'horloge pour créer une graine ! Inutile donc d'écrire du code du type :

```
1: var r = new Random(DateTime.Now.Millisecond);
```

Cela est totalement inutile.

Erreur n°2 – Multiplier les instances pour avoir "plus" d'aléatoire

Imaginons plusieurs instances d'une classe métier qui doivent chacune être en mesure de s'appuyer sur des nombres aléatoires (que ces instances fonctionnent dans le même thread ou en multi-thread n'a pas d'importance). Je vois souvent du code qui déclare une instance de Random dans chaque instance de la dite classe métier.

Illusion… et surtout grosse erreur !

Si les instances en questions sont créées les unes à la suite des autres il y a de fortes chances pour qu'elles s'initialisent dans la même tranche horaire (résolution de 20 ms environ) et qu'elles génèrent toutes la même série de nombres !

Pour s'en convaincre écrivons le code suivant :

```
1: var r  = new Random();
2: var r2 = new Random();
3: for (var i = 0; i<10; i++)
4: {
5:     Console.WriteLine(r.Next(100)+" -- "+r2.Next(100) );
6: }
```

(Pour tester des bouts de code ce genre sans charger VS et créer un projet, ce qui est très enquiquinant, je vous conseille fortement l'utilisation de LinqPad qui intègre aussi un petit éditeur de CSharp. C'est gratuit et génial pour tester des requêtes LINQ aussi).
La sortie sera la suivante par exemple :

```
3 -- 3
20 -- 20
15 -- 15
18 -- 18
83 -- 83
55 -- 55
52 -- 52
2 -- 2
39 -- 39
39 -- 39
```

Bien entendu à chaque “run” la série sera différence, mais regardez bien les deux colonnes de nombres... Et oui, elles sont identiques. La raison ? les variables r et r2 sont créées à la suite et il s’écoule moins de 20 ms entre ces créations, elles possèdent donc toutes deux la même graine (et *fabriqueront ainsi exactement la même série de nombres*).

Centraliser les appels

Il ne sert donc à rien de multiplier les instances de Random dans une application en espérant avoir “plus” d’aléatoire au final. Bien au contraire on risque d’obtenir, comme l’exemple ci-dessus le démontre, une uniformité qui n’a vraiment plus rien d’aléatoire, même “pseudo” !

Si les exigences mathématiques sont assez faibles on peut parfaitement se contenter de Random. Mais alors, le plus malin consiste à créer une seule instance pour toute l’application. On s’assure bien ainsi que chaque run de l’application se basera sur une série différence et surtout qu’au sein de l’application tous les nombres sembleront bien être aléatoires...

Je vous passe l’exemple d’une classe statique déclarant une instance de Random et exposant des méthodes statiques calquant les méthodes principales de cette dernière. C’est enfantin. En revanche, n’oubliez pas de déclarer un variable objet servant de verrou pour faire un lock sur les méthodes afin de s’assurer du bon fonctionnement en multi-threading. S’agissant d’une seule instruction j’avoue avoir d’un seul coup un doute sur ce conseil et cela mériterait que je le teste avant d’affirmer à mon tour des bêtises... (charité bien ordonnée commence par soi même, n’est-ce pas). Donc le sujet réclame plus ample investigation, prenez le conseil comme tel (à vérifier donc).

Bref, la façon la plus simple d’avoir réellement des nombres pseudo aléatoires dans une application est de n’utiliser **qu’une seule instance centralisée de Random**.

System.Security.Cryptography

Ce namespace, comme son nom le laisse deviner, contient de nombreuses classes fort utiles en cryptographie. Et qui dit cryptographie dit nombres (pseudo) aléatoires. Mais comme il s'agit ici de sécurité les exigences mathématiques placent la barre un peu plus haut.

Loin de moi l'idée d'aborder ce sujet en quelques lignes. Je veux juste attirer votre attention sur le fait que dans cet espace de noms se trouve un générateur qui remplace Random de façon plus efficace en évitant le risque de répétition des valeurs si plusieurs instances doivent être créées de façon proche dans le temps.

Si la solution d'une classe centralisant tous les appels de génération de nombre aléatoire vers une instance unique de Random ne vous convient pas, si les méthodes de Random ne vous semblent pas assez "solides" pour votre application, alors utiliser RNGCryptoServiceProvider du namespace indiqué.

Cette classe permet de créer des instances de RandomNumberGenerator offrant un comportement plus fiable que Random.

Pour s'en convaincre, reprenons l'exemple utilisé plus haut mais cette fois-ci en utilisant la nouvelle classe :

```
 1: var r3 = new System.Security.Cryptography.RNGCryptoServiceProvider();
 2: var r4 = new System.Security.Cryptography.RNGCryptoServiceProvider();
 3: byte[] b1 = new byte[1];
 4: byte[] b2 = new byte[1];
 5: for (var i = 0; i<10; i++)
 6: {
 7:       r3.GetBytes(b1);
 8:     r4.GetBytes(b2);
 9:     Console.WriteLine(b1[0]+" -- "+b2[0] );
10: }
```

Le code est similaire mais pas tout à fait équivalent car à la différence de Random la classe retournée par RNGCryptoServiceProvider génère des tableaux de bytes uniquement. Ici j'ai choisi des tableaux de 1 byte pour simuler des valeurs entières courtes ressemblant à celles du premier exemple (qui lui limitait les nombres à l'intervalle 0-99). Ici ce sont ainsi des nombres dans l'espace 0-255 qui seront tirés au sort. Regardons un run :

```
150 -- 102
15 -- 224
132 -- 128
```

```
167 -- 160
92 -- 76
129 -- 90
218 -- 253
226 -- 58
140 -- 225
37 - 252
```

Bien que les deux variables r3 et r4 soient créées à la suite, les deux générateurs obtenus ne sont pas synchronisés comme ils l'étaient avec Random.

Efficacité

Sur ma machine en 64bits octocoeur, il faut 3321,19 millisecondes pour générer 1 million de bytes aléatoires selon la méthode du second exemple, soit 0,00332119 millisecondes par octet. Il y a donc peu de chance que la vitesse d'exécution pénalise une application, s'agirait-il d'un jeu en 120 frames / seconde !

Avec Random on obtient un temps de 19,0011 millisecondes toujours pour 1 million d'octets générés, soir 1,90011^10-5 millisecondes par octet ! Environ 175 fois plus rapide donc.

Comme toujours dans notre métier il faut choisir entre consommation CPU et consommation mémoire ou bien, comme ici, entre sophistication et rapidité.

Conclusion

Il y aurait bien d'autres choses à dire sur les nombres aléatoires, pseudo ou non. C'est un sujet passionnant. Mais le but de ce petit billet était principalement d'attirer votre attention sur les mauvaises utilisations de Random et vous signaler l'existence dans le Framework d'autres classes plus "pointues" pour produire des séries pseudo aléatoires.

Passez de bonnes fêtes (pas aléatoires je l'espère, mais pas trop déterministes non plus, l'aléatoire fait malgré tout le sel de la vie !)

Placer un point d'arrêt dans la pile d'appel

Savez-vous qu'il est possible de placer un point d'arrêt directement dans la pile d'appel (dans le debugger de Visual Studio ? Peut-être pas car c'est une astuce assez peu utilisée.

La pile d'appel, vous connaissez

C'est l'une des fenêtres du debugger. Elle montre l'empilement des appels de méthodes à un moment donné (un break point, une exception) ce qui permet de remonter le fil des appels pour trouver où se cache une éventuelle erreur.

La pile se lit à l'envers, du bas vers le haut, si on veut la lire dans l'ordre chronologique des évènements. C'est la nature même d'une pile ... l'élément le plus récent (ajouter en dernier) se trouvant au sommet de la pile.

Vous savez aussi qu'en double cliquant sur l'une des lignes vous accédez directement au code source qui peut alors être inspecté.

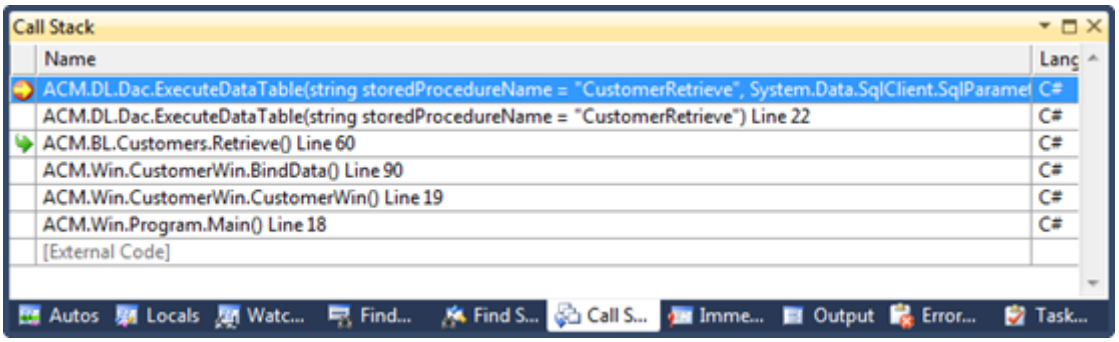
Vous vous êtes aussi rendu compte certainement que pour faciliter la lecture la ligne "courante" est surlignée en jaune et que ses appelants le sont en vert.

Une fois qu'on a dit tout cela, on a dit le principal sur cette fenêtre de debug.

Ajouter des points d'arrêt dans la pile d'appel

Mais il y a une astuce qui est rarement utilisée, parce qu'on n'y pense pas, parce que personne ne vous l'a fait voir avant. Question d'information et d'habitude, Visual Studio est tellement vaste et offre tellement d'options et d'astuces que je ne connais personne qui les maitrise toutes de toute façon.

Reprenons une vue sur la pile d'appel lors d'une session de debug :

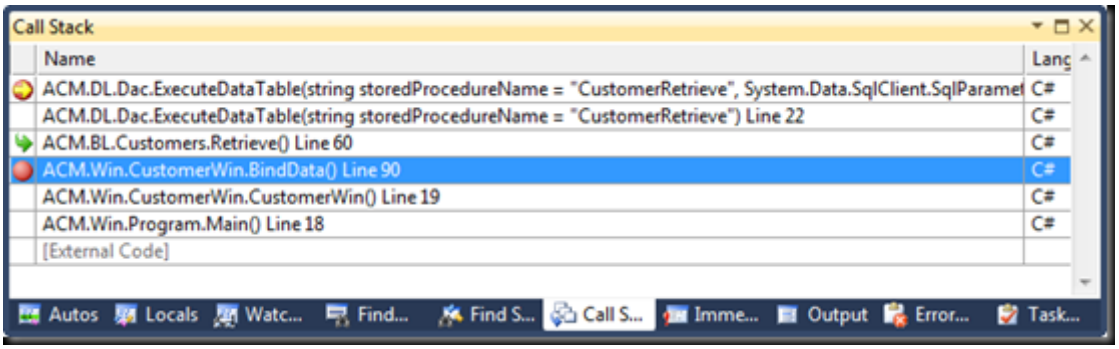


On peut voir ici une cascade d'appels depuis le Main() du programme jusqu'à la méthode ExecuteDataTable, en passant par toutes les couches de l'architecture du logiciel traversées par l'appel.

Imaginons que dans cette suite d'évènements nous désirions placer un point d'arrêt sur la méthode BinData(). Dans le cas le plus habituel le développeur double-cliquera sur cette ligne pour afficher le code et y placera le break point.

Mais il y a plus simple : faites un clic droit sur la ligne en question (par exemple ici BinData()) et sélectionnez dans le menu Break point / Insert Break point (dans un VS en français, que je n'utilise pas, cela doit être en toute logique Point d'arrêt / Insérer un point d'arrêt).

Le break point sera visualiser dans la pile d'appel :



Lors du prochain passage sur cette méthode le debugger s'arrêtera, comme sur n'importe quel break point.

Conclusion

C'est simple, pratique, ça ne casse pas trois pattes à un canard, mais ça vaut le coup de le savoir...

#if versus Conditional

La compilation conditionnelle n'est pas une grande nouveauté, les #if sont utilisés sous cette forme ou d'autres dans de nombreux langages depuis des temps immémoriaux... Sous C# nous disposons d'un outil de plus, l'attribut "Contional" qui reste à ma grande surprise méconnu, en tout cas fort peu utilisé. Réparons cette injustice et découvrons rapidement cet outil.

#if – la compilation conditionnelle

La compilation conditionnelle, en quelques mots, représente la capacité de certains compilateurs comme C#, et grâce à un marqueur introduit dans le code source, de pouvoir sauter des morceaux de codes qui ne seront pas compilés dans certaines conditions. Le cas le plus classique est le code de debug. Quand un code est ainsi instrumentalisé, plutôt que de fabriquer un nouveau code source pour la release qui serait débarrassée du code de debug, c'est ce dernier qui disparait automatique du code binaire tout en restant en place dans le code source.

La compilation conditionnelle est utilisée dans d'autres cas puisque, en général, et c'est le cas sous C#, on peut tester des conditions assez variées. Ainsi il est possible de prévoir un code spécifique Silverlight dans un source et sa variante WPF à la fois sans risque de mélange. Un seul code source existe ce qui évite la double maintenance.

Sous C# la compilation conditionnelle commence par un marqueur #if, de même nature que #region par exemple. Sauf que ce marqueur (ou “directive”) est suivi de la condition à tester (ou “pragma”). On peut écrire simplement :

```
#if Silverlight
    ...code spécifique SL
#endif

#if DEBUG
   Console.WriteLine("On est en debug!");
#endif
```

L’utilisation de #if peut intervenir n’importe où, tant que le code reste “compilable”. #if agit comme un “masque” qui supprime ou ajoute du code. Selon les mots clé définis Visual Studio grise dans l’éditeur le code qui n’est pas actif ce qui permet de voir facilement ce qui sera compilé ou non. Les aides à l’écriture du code comme IntelliSense, les messages d’erreur sous éditeur, etc, tout cela prend en compte le code à exécuter et gomme le code grisé comme s’il n’existait pas.

Le #if s’utilise aussi avec d’autre directives :

#endif qui termine le bloc #if
#define et #undef pour définir ou supprimer la définition d’un mot clé;
#else pour écrire un code alternatif si la condition échoue;
#elif, très peu connu, qui se comporte comme un #else #if en cascade.
Les conditions peuvent utiliser les opérateurs == (égalité), != (inégalité), && (et), || (ou). Les parenthèses sont aussi acceptées.

Tout code jugé non actif au moment de la compilation (ce qui dépend des mots clé définis) est tout simplement ignoré et non incorporé au binaire final.

Les problèmes posés par #if

Ils ne sont pas rédhibitoires mais ils existent.

Le premier et certainement le plus grave est l’atteinte à la lisibilité du code source.
Les directives comme #if sont alignées collées à gauche or le code suit généralement des règles de mise en page tabulées. Les parties grisées s’insèrent alors dans des parties utiles, l’alignement visuel est perturbé, etc. Ponctuellement cela n’est pas gênant, mais dès qu’on utilise beaucoup la directive #if le code est plus difficile à lire.

Et la lecture du code source doit toujours être facilitée, même dans des cas où le code serait réputé plus académique ou plus rapide, un professionnel, un vrai (pas un frimeur qui étale sa science) choisira toujours la lisibilité. Car un logiciel professionnel se doit d'être maintenable à tout moment, même par des gens ne l'ayant pas écrit. C'est "le" critère peut-être premier d'un "bon code" (mais pas le seul !).

Donc tout ce qui altère la lisibilité doit être supprimé. Les directives #if, #else et consorts sont ainsi à fuir selon ce principe. Pourtant elles sont utiles... Heureusement il y a une alternative que nous verrons plus bas.

Autre problème plus factuel causé par l'utilisation de #if : la complexité de mise en œuvre.

Ne rigolez pas ! (enfin si, rire est bon pour la santé). Quand je parle de complexité de mise œuvre je ne parle bien entendu pas de celle du #if en lui-même... Mais qui dit code conditionnel, dit aussi *appels* à ce code conditionnel. Donc appels qui doivent *eux-mêmes devenir conditionnels*, sinon le code ne compile tout simplement pas ! Et là ça peut devenir le Bronx niveau lisibilité du source...
Imaginons le code suivant :

```
...
#if DEBUG
    private void sendDebugInfo(string message)
    { ... }
#endif

...
    string s = OperationA();
    sendDebugInfo(s);
    Operation(b)
...
```

Si nous compilons en mode Debug, tout ira bien. Mais si nous passons en mode Release, la compilation ne passera pas, "sendDebugInfo(s);" en plein milieu de notre code application est alors inconnu.

Il faut donc écrire :

```
...
#if DEBUG
    private void sendDebugInfo(string message)
    { ... }
#endif
```

```
...
    string s = OperationA();
#if DEBUG
    sendDebugInfo(s);
#endif
    Operation(b)
...
```

Et là tout de suite, si on colle plein d'appel de ce genre dans son code, c'est ce que j'appelais le Bronx plus haut... Ca devient illisible, en dehors d'être pénible à écrire.

Conclusion : le #if c'est super, ça marche, mais c'est un truc vieux comme le C et certainement avant et on ne peut vraiment pas dire que ces langages étaient réputés pour leur lisibilité...

Faire autrement s'impose. Heureusement le Framework apporte une solution bien plus élégante.

L'attribut Conditional

L'attribut "Conditional" (ou la classe ConditionalAttribute) permet de marquer une méthode ou une classe. Le code ainsi marqué est "conditionnel" dans le sens ou l'attribut prend en paramètre les mêmes pragmas que #if (par exemple "Debug" ou "Silverlight").

Les différences essentielles avec #if

La première est que l'attribut se pose sur une classe ou une méthode. On ne le met plus n'importe où. Cela clarifie déjà un peu les choses. Il n'y a pas de bouts de code conditionnels.

La seconde découle de la première : c'est mille fois plus lisible et cela s'intègre parfaitement à la mise en page globale du code source.

La troisième est qu'il y a une part de magie derrière cet attribut : si je marque une méthode [Conditional("DEBUG"], tout comme si elle était entourée d'un #if DEBUG elle ne sera plus compilée, mais surtout : tous les appels à cette méthode disparaitront du code final, ce qui règle l'un des problèmes essentiels de #if expliqué plus haut.

Pour être exact le code conditionnel n'est pas supprimé du binaire final, un test avec Reflector par exemple vous le prouvera facilement. Mais il n'est pas envoyé au JIT à l'exécution et l'ensemble des appels à ce code a bien disparu. En ce sens #if est donc plus "efficace" question "ménage" dans le binaire mais c'est très relatif.

Utilisation

L'exemple de code précédent devient ainsi :

```
...
    [Conditional("DEBUG"]
    private void sendDebugInfo(string message);
    { ... }

...
    string s = OperationA();
    sendDebugInfo(s);
    OperationB();
...
```

C'est beaucoup plus clair, plus concis et le code de l'application n'est pas perturbé par des directives #if. Il l'est déjà par l'instrumentalisation (l'appel à sendDebugInfo()), c'est déjà bien assez comme cela...

L'attribut Conditional peut être multiplié pour tester sur plusieurs conditions. Ce n'est pas comme #if qui accepte une expression (simple) qui sera évaluée. Dans le cas de l'attribut Conditional, si on veut tester deux conditions, il suffit de mettre deux fois l'attribut, chacun avec son test.

Conclusion

La directive #if rend le code moins lisible et moins maintenable mais elle fait ce qu'elle dit, totalement : si la condition ne s'évalue pas à True, tout le code marqué disparait du binaire final. Si on insère des blocs de codes énormes qu'on gère avec des #if (en se rappelant qu'on peut tester des tas de pragmas et pas seulement Debug!) on obtiendra un compilé plus léger. Dans ce cas précis #if prend l'avantage.

Dans tous les autres cas l'attribut Conditional est plus efficace, moins verbeux (les appels aux méthodes conditionnelles ne doivent pas être entourées de #if), plus lisible, et parfois souvent équivalent question taille de l'exe final (le code conditionnel de quelques lignes reste dans l'exe, mais s'il y a 100 appels, ils seront supprimés sans avoir rien de plus à écrire; le ratio final est proche de zéro).

Il ne s'agit pas de haute technologie du futur, mais discuter des petites choses simples qui rendent le code plus lisible et plus maintenable est tout aussi important !

Les events : le talon d'Achille de .NET...

Les events (gestion d'évènements) sont d'une grande puissance et existent dans presque tous les langages récents (et même quelques un plus anciens). Ils autorisent un modèle de programmation évènementiel qui se calque bien sur la façon dont sont gérées les IHM des OS modernes (pilotés par l'utilisateur et ses clics souris). Hélas ce concept réutilisé par le Framework .NET ne lui va pas très bien. Pire, dans un environnement managé (avec Garbage Collector) les évènements sont une source inépuisable de pertes mémoire !

Des memory leaks en managé ?

On nous aurait menti ? Un environnement managé peu connaitre des pertes mémoire, comme ça, juste en programmant "normalement" et sans bug ?

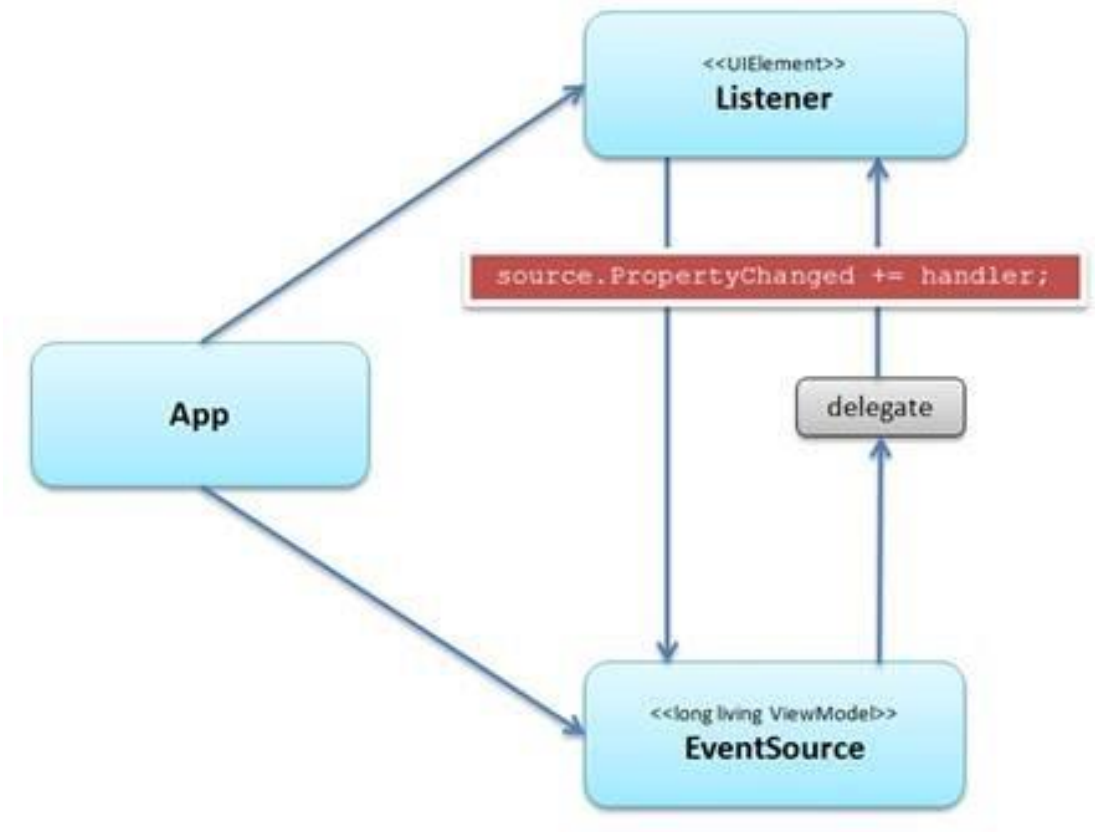
Et bien oui !

Vous ne le saviez pas ? Alors il est grand temps d'envisager vos gestions d'évènement sous un autre angle et de vérifier le code que vous avez écrit jusqu'ici !

Le problème

Vous allez vite comprendre : en utilisant des évènements CLR classiques on créé, par force, des références fortes entre objets. Je m'explique : si la classe A propose l'évènement PropertyChanged par exemple (c'est-à-dire toute classe bien construite !), lorsqu'un objet B s'abonne à ce dernier, il existe une référence forte dans l'instance de A vers l'instance B. Un évènement n'est jamais qu'une gestion de callback, ce qui implique la présence d'une liste de pointeurs chez l'émetteur de l'évènement, pointeurs vers les méthodes enregistrées par tous les souscripteurs. Lorsque les conditions de l'évènement sont favorables à son apparition, la liste des abonnés est balayée et chaque méthode de chaque abonné est appelée.

Bref, Si B souscrit à l'évènement PropertyChanged de A, il existe une référence vers B stockée dans l'instance A. Ce mécanisme est automatique sous .NET ce qui fait que le programmeur s'en rend moins compte. Mais il ne s'agit rien de plus que du pattern Observer (Gamma, Helm, Johnson, Vlissides).



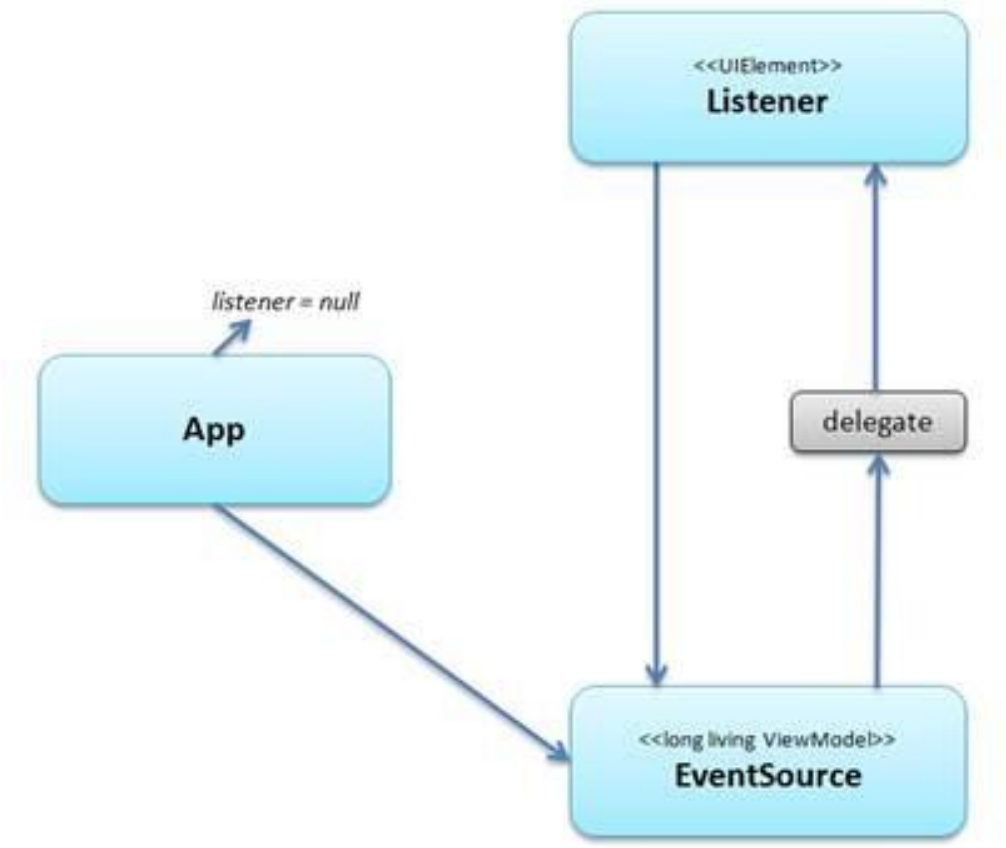
Le schéma ci-dessus nous montre le jeu entre source de l'évènement (EventSource en bas) et écouteur (ou abonné, Listener en haut). L'application représentant la "glue" qui permet à EventSource et Listener d'exister ensembles dans un tout cohérent.

La source montrée sur le schéma est un ViewModel, cas classique aujourd'hui mais ce n'est qu'un simple exemple. De même, le récepteur, le Listener, est un UIElement mais ce pourrait être n'importe quoi d'autre (un UserControl, un Control...).

Que se passe-t-il si la source d'évènement a une durée de vie plus longue que celle de l'abonné ?

Dans notre exemple, supposons que l'UIElement soit supprimé de l'arbre visuel. Le ViewModel étant toujours actif. Que va-t-il se passer au niveau de la libération mémoire de l'abonné ?

... Rien. Bien qu'il semble ne plus être référencé nulle part, bien qu'il ait disparu de l'arbre visuel, il ne sera jamais effacé par le Garbage Collector. La raison ? ... Il existe toujours une référence forte vers l'abonné dans la mémoire de l'évènement exposé par la source (le ViewModel) !

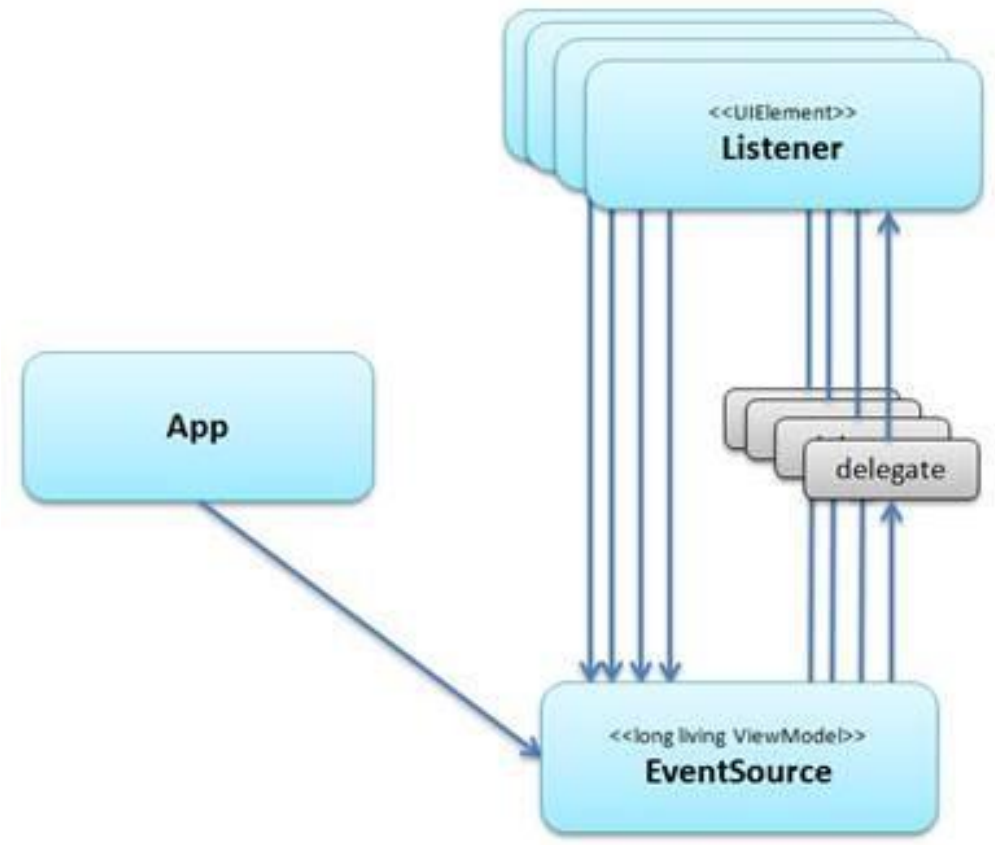


Comme on le voit ci-dessus, l'application ne possède plus de référence vers le Listener, mais il existe toujours bien une telle référence dans EventSource, c'est le delegate qui pointe la méthode du Listener à appeler lorsque l'évènement se produit !

Cela est doublement fâcheux : d'abord et comme je le disais, nous avons là un cas typique de perte de mémoire puisqu'un objet qui n'est plus référencé restera malgré tout en mémoire, et puis il peut y avoir des effets de bord car à chaque fois que l'évènement se produira, la méthode du Listener continuera à être appelée... Supposons maintenant que des tas d'instances de Listener soient créées et détruites au cours de la vie du ViewModel possédant l'EventSource, on entrevoit les conséquences délétères sur la mémoire consommée par l'application ainsi même que sur sa vitesse d'exécution et donc sa réactivité (plein de code inutile continue à être appelé).

Si l'exemple utilise comme Listener un élément visuel c'est que ces objets ne proposent pas d'évènement de type *Unloaded* qui pourrait être attrapé pour permettre de se désabonné à tous les évènements qui étaient écoutés. Et aucune autre classe habituelle ne possède un tel évènement. Enfin rappelons que le destructeur n'est pas forcément appelé dans un environnement managé ce qui fait qu'on ne peut pas compter sur lui pour faire le ménange.

Supposons que le ViewModel en question ait une durée de vie vraiment longue (le ViewModel de la MainPage d'une application ayant une vie aussi longue que l'application elle-même par exemple), on comprend que l'entassement des pertes mémoires peut devenir énorme comme le montre le schéma suivant :



Se désabonner ?

La règle d'or, quel que soit le contexte de l'application et la méthodologie utilisée (MVVM ou non entre autre), c'est qu'il faut toujours qu'un objet se désabonne de tous les évènements qu'il écoutait avant d'être détruit (au sens le plus large, comme dans notre exemple "supprimé de l'arbre visuel" est une forme de destruction mais qui ne va pas à son terme, justement).

Dans de nombreux cas mettre en place une telle logique est simple (si les objets sont créés et détruits en des points bien connus de l'application).

Dans d'autres cela est purement impossible puisque l'objet ne sait même pas qu'il est déférencé (le déférencement étant une action d'un objet tiers, par nature).

Le désabonnement n'est donc pas aussi simple que cela à implémenter... Ce ne peut donc pas être une réponse globale au problème posé, en tout cas pas sous une forme aussi simpliste.

La solution

Même si je peux constater au quotidien que bon nombres de développeurs n'ont pas forcément conscience de ce problème, il est malgré tout connu de longue date. Et les équipes de développement du Framework autant que des produits annexes comme Silverlight, WPF ou le Toolkit sont conscientes du risque et programment d'une façon qui évite bien entendu le piège. Des évènements comme ceux supportés par les interfaces INotifyPropertyChanged (ou même INotifyCollectionChanged) sont malgré tout très souvent utilisés.

Pour régler le problème de façon radicale sans trop avoir à se poser de question ni mettre en place des usines à gaz l'équipe du ToolKit Silverlight a mis en place une parade ... imparable !

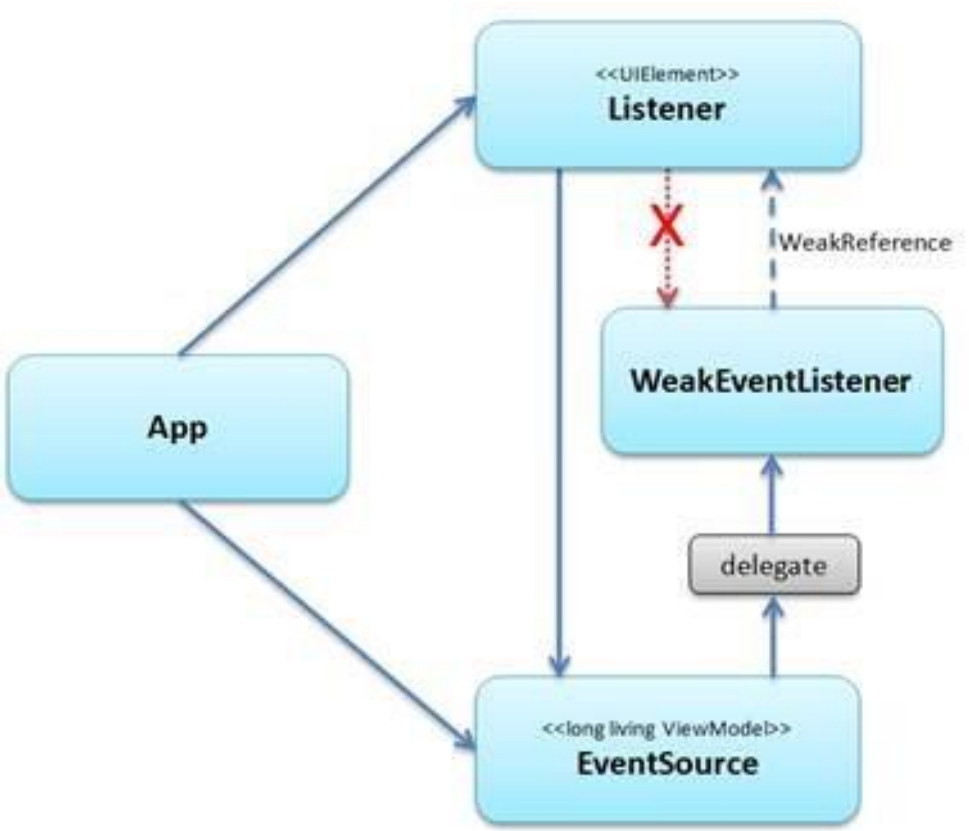
Il s'agit d'une toute petite classe, WeakEventListener, hélas ayant une visibilité "internal" ne permettant pas de la ré exploiter dans nos applications. Mais étant donnée sa taille, chacun est loisible d'en avoir une copie dans ses applications.

Les Weak References

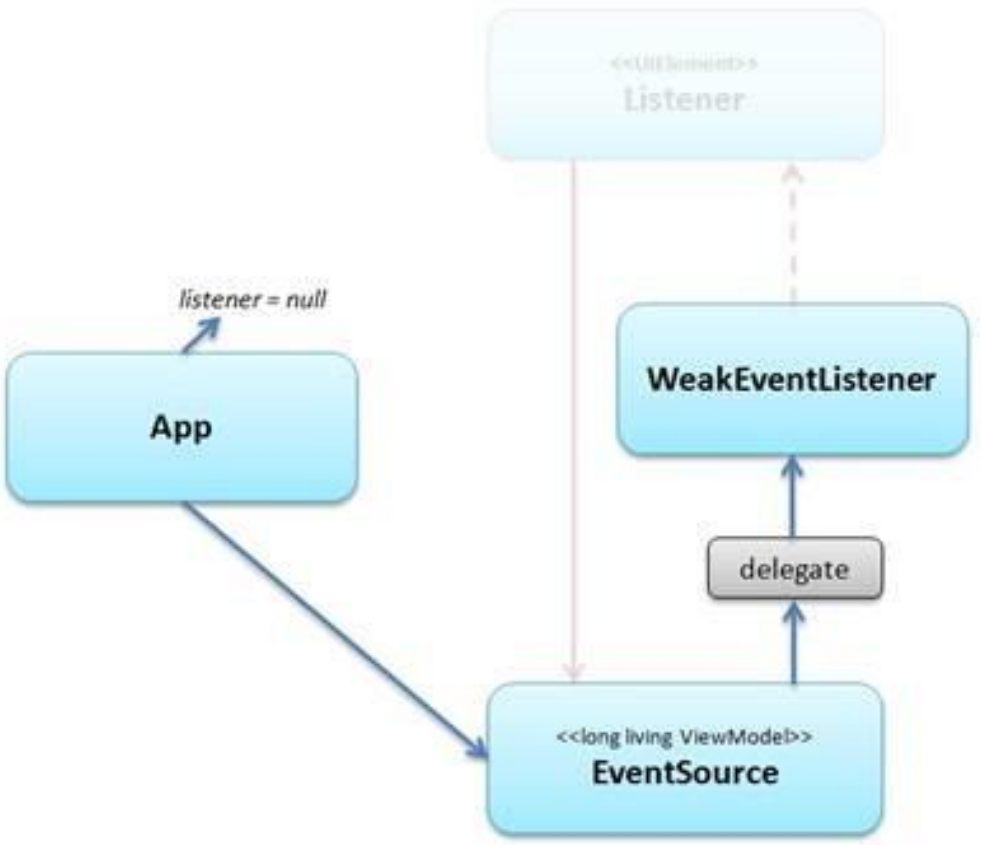
Je renvoie le lecteur à l'un des anciens articles qui faisait justement le point sur la notion de référence faible sous .NET, un concept intégré dès le départ mais omis de la plupart des livres et des formations... Les lecteurs de Dot.Blog, gratuitement, eux connaissent le sujet depuis longtemps : Les références faibles sous .NET (weak references) (un article de 2009). En gros, les références faibles permettent de garder un pointeur sur un objet sans que cela n'empêche le Garbage Collector de le libérer s'il n'est plus référencé ailleurs. Simple et efficace, mais cela complexifie un tout petit peu l'écriture du code, bien entendu.

En Pratique

Il suffit donc de mimer l'équipe du Toolkit dans vos applications pour vous protéger du dangereux problème présenté dans ce billet. La classe WeakEventListener fonctionne de façon très simple : c'est une instance intermédiaire entre la source de l'évènement et son abonné. Elle est référencée par la source et contient la référence vers l'abonné. Si l'abonné n'est plus référencé ailleurs, il sera supprimé. L'instance de la référence faible ne l'interdira pas. Ce fonctonnement est schématisé ici :



Quand l'objet Listener n'est plus utilisé, la mémoire ressemble à cela :



Remplacer un problème par un autre ?

Hmmm ... ceux qui ont tout suivi l'ont déjà compris, la recette n'est pas miraculeuse, elle ne fait que déplacer le problème, voire même le remplacer par exactement le même ! En effet, et le schéma ci-dessus le montre parfaitement, si le Listener peut enfin être libérer sans créer de fuite mémoire, c'est l'instance de WeakEventListener qui reste accrochée à la source !

Cela fait un peut penser à ces papiers collants qui se recollent immédiatement sur une autre partie de la main quand on secoue cette dernière pour tenter de s'en débarrasser...

Ce n'est pas faux mais il faut nuancer les choses.

Tout d'abord une instance de WeakEventListener ne pèse pas lourd et causera une fuite mémoire bien moins grave qu'un gros objet plein de variables, de code Xaml, d'animations etc...

Ensuite il n'est pas interdit pour la source de faire le ménage. Mais comment ? Il n'est pas simple de balayer la liste des abonnés d'un évènement tellement cette gestion est cachée par le Framework.

L'équipe du ToolKit aurait-elle juste lâché la proie pour l'ombre ?

La réponse : la lévitation objectivée

La réponse se trouve dans la façon de faire ce montage de références faibles. En réalité si on ne fait que mimer le système en place pour remplacer l'abonné par une référence vers l'abonné, nous venons de le voir, nous ne faisons que remplacer une fuite mémoire par une autre.

Il est clair qu'il ne faut pas implémenter la solution de façon aussi abrupte.

Il faut trouver un moyen de faire en sorte que l'objet WeakEventListener soit en "lévitation" dans le vide, il doit relier les deux intervenants (source et abonné) par des références faibles et n'être lui-même référencé par personne. Il doit "flotter" et pouvoir être libéré par le Garbarge Collector quand le Listener n'existe plus.

La mise en place est un peu délicate et repose justement sur des petites ruses d'implémentation de la classe WeakEventListener et surtout de son utilisation... Elle doit pointer des actions codées de façon statiques pour qu'aucune cible ne lui soit accrochée (méthode Target de System.Delegate, puisqu'un pointeur de méthode est un délégué).

Bref ce n'est pas si simple que ça mais une fois le concept bien compris on peut développer un code libéré de cette épée de Damoclès que sont les évènements CLR classiques...

Le Code ?

Comme je le disais il se trouve dans le Toolkit... Et comme ce dernier est fourni en code source aussi (www.codeplex.com/Silverlight) rien de plus facile que de l'extraire.
Il y a un peu plus facile en fait... Beat Kiener, un développeur suisse, s'est donné la peine d'extraire la classe, d'ajouter deux ou trois contrôles pour éviter qu'elle soit mal utilisée (ce qui ruine son effet) et d'englober tout cela dans un exemple.

Vous pouvez lire sont billet (en anglais) en complément de celui-ci (il expose plus en détail le fonctionnement de la classe, et pour illustrer mon propos je lui ai emprunté les schémas – rendre à César ce qui est sien est important)
: http://blog.thekieners.com/2010/02/11/simple-weak-event-listener-for-silverlight/
Vous pouvez télécharger le code qu'il propose : code source et exemple

Conclusion

La solution du Toolkit est intéressante, les petites modifications faites par Kiener sont un plus, mais très franchement l'objet de ce billet n'est pas forcément de vous obliger à mettre tout cela en œuvre. Mon objectif était surtout de vous alerter sur un problème récurrent si ce n'est méconnu en tout cas fort peu débattu et rarement présenté. Pourtant il s'agit là d'un vrai problème qui pose question, quelque chose qui devrait être réglé par le Framework lui-même à mon avis.

Désormais vous savez que le problème existe, qu'il y a des parades (développer en connaissance de cause ou utiliser WeakEventListener), et je suis certain que vous ne regarderez plus un event de la même façon maintenant (certains vont même stresser et se replonger dans leur code pour voir si ...).

StringFormat se joue de votre culture !

Silverlight 4 a introduit le paramètre StringFormat dans la syntaxe du Binding. C'est une excellente chose et supprime le besoin de développer un convertisseur pour la moindre mise en forme de données. Toutefois il y a un petit bug... StringFormat ignore la culture de l'utilisateur et en bon ricain qu'il est, il considère que tout le monde parle la langue des cowboys...

On aime bien les cowboys

On les aime bien, c'est un fait, sinon leurs films, leurs musiques et même leurs hamburgers ne se vendraient pas comme des ... petits pains dans notre joli pays à la culture millénaire...

L'utilisateur cette bête étrange

Mais l'utilisateur, cette animal étrange que certains disent avoir déjà vu (info ou intox ? légendes urbaines ? Les témoignages sont-ils fiables ?) semble avoir des gouts pour le moins paradoxaux... S'il se jette sur le premier IPhone venu, s'il déjeune le midi en se "régalant" d'un big Mac (il y a une astuce ou pas ?), s'il se gave de streaming d'Avatar en écoutant le top 50 chanté en langue Hollywood (pour le chewing-gum qu'ils ont dans la bouche en parlant certainement), l'utilisateur, entité pourtant pensante (mais aucun article là dessus n'a été publié dans une revue scientifique à comité de lecture, restons méfiant donc) ne supporte pas un seul instant que ne serait-ce qu'un bout de texte apparaisse en anglais sur son écran !

Il est soudain pris de convulsions, on parle même d'une forme d'œdème du visage entrainant, certainement par mauvaise oxygénation du cerveau, l'émission non contrôlée de quelques jurons gaulois. "Fascinating" aurait dit M. Spock en prenant connaissance de ce comportement terrien.

Drôle de bestiaux quand même... Certainement qu'il est plus facile de se gaver de hamburgers et de musique made in USA que d'apprendre la langue. Le français a une réputation de feignant peut-être méritée, allez savoir...

L'informaticien ce coupable !

Forcément coupable puisque complice de l'anti-France, la fameuse sous-culture-américaine-qui ne-vaut-rien, disent-ils la bouche pleine de Cheeseburger et les oreillettes de leur IPod crachant à fond la rétrospective de Mickael Jackson...

C'est donc forcément lui, en bout de course, qui se fera remonter les bretelles (expression idiote, puisqu'à part Harold Hyman sur BFM TV personne ne porte plus de bretelles depuis la dernière guerre – remarquons qu'il est américain le bougre, si c'est pas une preuve !).

La solution !

Bon, je vais vous la donner, mais lorsque vous allez voir la longueur de la chose vous allez comprendre pourquoi j'ai un peu brodé autour du sujet !

Le plus simple pour régler la question est ainsi d'ajouter dans le constructeur de toutes les Vues d'une application (on ne sait jamais où on utilisera un StringFormat en Xaml) :

```
this.Language =
XmlLanguage.GetLanguage(Thread.CurrentThread.CurrentCulture.Name);
```

Voilà... A noter que XmlLanguage se trouve dans le namespace System.Windows.Markup qu'il faut ajouter (soit en using, soit avec un point devant XmlLanguage...).

Conclusion

Il fallait un peu d'humour pour habiller cette "ruse" qui permet de contourner ce léger bug de StringFormat tant le sujet est peu passionnant en lui-même et la solution d'une brièveté déconcertante. D'autant que ce problème n'a jamais été corrigé en tant d'années.

J'ai encore sauvé quelques informaticiens de la fusillade... Mais ne me remerciez pas, les utilisateurs trouveront bien d'autres raisons pour vous maudire ! 

Conversion d'énumérations générique et localisation

Lorsqu'on travaille avec des énumérations il est très fréquent d'avoir à traduire leurs valeurs par d'autres chaines de caractères. Soit parce que les valeurs ne sont pas assez parlantes pour l'utilisateur, soit parce qu'il est nécessaire de localiser les chaines pour s'adapter à la culture de l'utilisateur.Il faut aussi ajouter les cas où les énumérations sont traduites en des valeurs d'un autre type (des couleurs par exemple) ce qui très courant avec le databinding. Prenons une simple énumération :

```
public enum ProgramState
{
    Idle,
    Working,
    InError
}
```

Il s'agit du cas fictif d'une énumération indiquant l'état du programme. Elle prend trois valeurs.

Imaginons que nous souhaitions afficher un petit rond de couleur dans un coin de la page représentant l'état, vert pour Idle (en attente), jaune pour Working (travail en cours) et rouge pour InError (en erreur).

La programmation par Binding sous Xaml a cela de pénible que dans les cas de ce type, courants, il faut à chaque fois écrire un convertisseur. Cela n'est pas grand chose mais c'est fastidieux. Lorsqu'on utilise le modèle MVVM il est possible de se passer de ces convertisseurs en laissant le travail au ViewModel (après tout c'est son boulot que d'adapter les données à la Vue). On peut aussi préférer conserver le rôle des convertisseurs.

Dans ce dernier cas comment ne pas avoir à écrire un convertisseur pour chaque cas particulier ?

Convertisseur générique

L'idée serait de disposer d'un convertisseur "générique" écrit une seule fois et qui s'adapterait à tous les cas de figures les plus classiques. Il serait paramétrable à volonté et plutôt que d'écriture plusieurs convertisseurs on utiliserait plusieurs instance du même convertisseur avec des paramètres différents.

Un code déclaratif, conforme à l'esprit de Xaml, plutôt que du code fonctionnel en dur donc.

En réalité un tel convertisseur s'écrit de façon très simple en quelques lignes. On en doit l'idée à Andrea Boschin, un MVP italien.

Voyons d'abord comment résoudre le problème posé en introduction...

Résoudre la conversion énumération / couleur

Ce n'est qu'un exemple et vous allez vite comprendre qu'on peut remplacer "couleur" par n'importe quelle type d'objet, voire une autre énumération pour des opérations de transcodage. On peut bien entendu utiliser la même stratégie pour traduire une énumération en allant piocher les valeurs dans le Resource Manager. Mais revenons aux couleurs.

Imaginons notre indicateur rond placé dans un UserControl :

```
<UserControl
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         ...  >
```

```
    <UserControl.Resources>
            <gc:EnumConverter x:Key="stateToColor">
                <gc:EnumConverter.Items>
                    <SolidColorBrush Color="green" />
                    <SolidColorBrush Color="yellow" />
                    <SolidColorBrush Color="red" />
                </gc:EnumConverter.Items>
            </gc:EnumConverter>
    </UserControl.Resources>
       <Grid x:Name="LayoutRoot">
          <Ellipse Fill="{Binding CurrentProgramState,
Converter={StaticResource stateToColor}}"
             Width="10" Height="10" />
       </Grid>
   </UserControl>
```

On supposera ici que la propriété CurrentProgramState est de type ProgramState (l'énumération, voir plus haut) et que cette valeur est disponible dans le DataContext courant.

La première chose qu'on observe est la déclaration, dans les ressources du UserControl, d'une instance de la classe EnumConverter (dans le namespace "gc" pour "Generic Converter"). Cette instance possède la clé "stateToColor".

La chose intéressante est la déclaration d'une section "Items" dans l'instance du convertisseur. Ici on trouve trois lignes, chacune déclarant une SolidBrushColor, une verte, une jaune et une rouge.

Ensuite, dans le code du UserControl on trouve une Ellipse dont la propriété Fill (le remplissage) est bindée à la propriété CurrentProgramState (de type ProgramState), mais en passant par notre convertisseur générique (l'instance dont la clé est "stateToColor").

Et c'est tout... Dès que la propriété CurrentProgramState changera de valeur (si elle est bien implémentée) l'Ellipse (enfin le rond ici) prendra automatiquement la couleur voulue. Sans écrire de code "en dur".

Avantages

Il y a plusieurs avantages à cette technique. D'abord le fait qu'on puisse traduire n'importe quelle énumération en une série de valeurs de n'importe quel type.

Ensuite, le mode d'utilisation est totalement déclaratif en Xaml, ce qui permet facilement de modifier les conventions sans toucher le code de l'application. Un Designer pourra ainsi très bien décider de changer l'Ellipse en quelque chose de plus "sexy" et adapter les trois couleurs pour qu'elles correspondent mieux à la charte couleur par exemple.

On peut utiliser ce procédé pour retourner des chaines de caractère traduites en piochant directement dans le Resource Manager.

Enfin, on peut déclarer le convertisseur dans App.Xalm au lieu des ressources propres à un UserControl et rendre disponible les conversions dans toute l'application de façon homogène et fiable.

Inconvénients

Rien n'est parfait, surtout un code si simple (nous le verrons plus bas). Ici, vous l'avez compris, la correspondance s'effectue de façon directe entre la valeur numérique des éléments de l'énumération et l'ordre de déclaration des valeurs retournées par le convertisseur.

C'est parfait pour la majorité des énumérations qu'on déclare généralement comme je l'ai fait pour l'exemple plus haut.

Mais si le développeur a numéroté lui-même les valeurs (imaginons que "InError" dans l'énumération exemple soit déclarée "InError=255") cette belle correspondance 1 à 1 disparait et le procédé n'est plus applicable...

Les énumérations marquées avec l'attribut [Flags] ne sont pas utilisables non plus avec ce convertisseur pour des raisons évidentes.

Se pose aussi le problème des évolutions du code. Si la déclaration de l'énumération est modifiée, le programme fonctionnera toujours (puisqu'il est compilé en se basant sur les noms des items) mais plus le ou les convertisseurs déclarés sur l'énumération. Cela n'est pas choquant en soi. Modifier une énumération après coup est une prise de risque qui réclamera quelques contrôles dans le code malgré tout. Toutefois, si on déclare les convertisseurs génériques dans App.Xaml comme je l'indiquais plus haut, cette centralisation facilitera la révision du code. Si les convertisseurs sont éparpillés dans des tas de contrôles, le travail sera plus dur. Mais travailler sans méthode ni rigueur rend toujours la maintenance plus difficile, c'est une évidence !

Le code

```
public class EnumConverter : IValueConverter
{
```

```
    private List<object> items;
    public List<object> Items
    {
        get  { return  (items == null) ? items = new List<object>() : items;   }
    }

    public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
    {
        if (value == null)
            throw new ArgumentNullException("value");
        else if (value is bool)
            return this.Items.ElementAtOrDefault(System.Convert.ToByte(value));
        else if (value is byte)
            return this.Items.ElementAtOrDefault(System.Convert.ToByte(value));
        else if (value is short)
            return this.Items.ElementAtOrDefault(System.Convert.ToInt16(value));
        else if (value is int)
            return this.Items.ElementAtOrDefault(System.Convert.ToInt32(value));
        else if (value is long)
             return this.Items.ElementAtOrDefault(System.Convert.ToInt32(value));
        else if (value is Enum)
             return this.Items.ElementAtOrDefault(System.Convert.ToInt32(value));
        throw
         new InvalidOperationException(string.Format("Invalid input value of type
'{0}'", value.GetType()));
     }

     public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture)
     {
         if (value == null)
             throw new ArgumentNullException("value");
         return this.Items.Where(b => b.Equals(value)).Select((a, b) => b);
     }
}
```

Le support des booléens est un peu la cerise sur la gâteau. C'est un besoin assez fréquent que de convertir un booléen en autre chose, notamment sous Xaml en Visibility.Collapse/Visible.

Grâce au convertisseur générique on peut écrire :

```
<gc:EnumConverter x:Key="boolToVisibility">
    <gc:EnumConverter.Items>
        <Visibility>Collapsed</Visibility>
        <Visibility>Visible</Visibility>
    </gc:EnumConverter.Items>
</gc:EnumConverter>
```

On utilise ensuite l'instance du convertisseur dans un binding entre un booléen et la propriété Visibility d'un élément visuel.

Conclusion

Idée simple et pratique, qui a quelques limites mais généralement peu gênantes au quotidien, le convertisseur générique peut éviter l'écriture de nombreux petits convertisseurs.

C# : créer des descendants du type String

C'est un peu un piège, bien entendu, la classe String est "sealed" et il est donc impossible d'en hériter, comme d'autres classes de base du Framework... Pourtant le besoin existe. Pourquoi vouloir des chaines de caractères descendant de string (ou d'autres de base) ? Comment contourner l'interdiction du Framework ? Répondre à ces questions est le thème du jour !

Pourquoi ?

C'est la première question, et la plus importante peut-être. Pourquoi vouloir créer des types descendant de string (ou d'autres types de base sealed) ? En quoi cela peut-il être utile ?

Si je parle d'utilité c'est bien parce que le code doit répondre à cet impératif, tout code sans exception. On code pour faire quelque chose d'utile. Sinon coder n'a pas de sens.

Le Framework ne permet pas de la création de classes héritant de string ou, et pour bloquer toute velléité en ce sens, la classe string est celée (sealed). Les concepteurs du Framework ont définitivement fermé cette porte. Mais ils en ont ouvert une autre : les extensions de classe. Cela permet d'étendre les possibilités de toute classe, même sealed, donc de string aussi.

Cela serait parfait si le besoin d'hériter d'une classe se limitait à vouloir lui ajouter des méthodes...

Prenons un cas concret : vous créez un logiciel qui pour autoriser la saisie de nombreux paramètres de classes différentes utilise une PropertyGrid (comme celle de Windows Forms, il en existe certaines implémentations pour Silverlight et celle de WF peut s'utiliser sans problème sous WPF). Au sein d'un tel mécanisme vous pouvez généralement définir vos propres éditeurs personnalisés, qui dépendent du type de la valeur. Par exemple, pour une propriété de type Color vous pourrez écrire un éditeur offrant un nuancier Pantone et une "pipette". Cela sera plus agréable à vos utilisateurs que de taper à l'aveugle un code hexadécimal pour définir une couleur.

Imaginons une seconde que parmi ces paramètres qui seront saisis dans une PropertyGrid (ou son équivalent Silverlight) il se trouve certaines chaines de caractères définissant par exemple le nom d'un fichier externe.

Dans un tel cas vous souhaitez que plutôt qu'un simple éditeur de string s'affiche aussi un petit bouton "..." qui permettra à l'utilisateur de browser les disques pour directement sélectionner un nom de fichier existant. Peut-être même la zone gèrera-t-elle le drag'n drop depuis l'explorateur.

Hélas... Soit vous enregistrez le nouvel éditeur pour le nom d'une propriété précise (ce qui est très contraignant et source de bogues), soit vous l'enregistrez pour son type, string, et dès lors ce seront toutes les strings qui bénéficieront du browser de fichiers, ce qui n'a aucun sens.

Que ne serait-il pas plus facile de définir juste "public class NomDeFichier : string {} " et Hop ! l'affaire serait jouée !

L'éditeur serait enregistré pour le type "NomDeFichier", les noms de fichiers dans les paramètres ne seraient plus de type "string" mais de type "NomDeFichier" et tout irait pour le mieux dans le meilleur des mondes.

Donc voici concrètement un cas qui montre l'utilité évidente de créer des classes héritant de string (ou d'autres classes sealed), même totalement vides, juste pour créer une CLASSification, à la base même de la programmation objet malgré tout...

Je ne doute pas qu'éclairez par cet exemple vous en trouviez d'autres, même totalement différents.

En tout cas nous avons répondu à la première question. C'est utile, et puis la programmation objet se base sur l'héritage pour régler de nombreux problèmes, il y a donc une légitimité naturelle à vouloir hériter d'une classe. "sealed" est un peu frustrant. C'est presque un contre-sens dans un monde objet. La justification du code plus efficace produit par une

classe sealed me semble assez artificielle et ne se justifiant pas. Mais C# est ainsi fait, la perfection n'existe pas. Heureusement la grande souplesse du langage permet de contourner assez facilement ce genre de problème !

Comment ?

Je vous l'ai déjà dit : ce n'est pas possible, n'insistez pas ! ...

Mais comme ce billet n'existerait pas si je n'avais pas une solution à vous proposer, vous vous dites qu'il doit y avoir un "truc".

La classe string est sealed. Donc il n'y a pas de "truc" magique. Pas de moyen de bricoler le Framework non plus.

La solution est toute autre.

Elle consiste tout simplement à développer une autre classe qui n'hérite de rien.

Hou là ! Réinventer le type string juste pour une raison de classification semble carrément overkilling !

C'est vrai, et nous ne nous lancerons pas sur une voie aussi complexe. En revanche on peut être rusé et tenter d'en écrire le moins possible tout en se faisant passer par une string...

En fait c'est assez facile mais cela utilise des éléments syntaxiques peu utilisés comme les opérateurs implicites.

L'astuce consiste à créer une classe "normale" n'héritant de rien, et possédant une seule propriété, Value, de type string (ou d'un autre type sealed dont on souhaiterait hériter).

C'est sûr que ce n'est pas compliqué à écrire mais cela ne règle pas la question. Il n'est pas possible de faire passer notre classe pour string. Partout il faudra changer 'x = "toto"' par 'x.Value = "toto"' et ce n'est pas du tout ce qu'on cherche !

C'est oublier les opérateurs "implicit" qui permettent de convertir une instance d'une classe en d'autres types (et réciproquement). Implicitement. C'est à dire sans avoir à écrire quoi que ce soit dans le code qui utilise la dite classe à convertir.

Pour commencer nous aurons ainsi un code qui ressemble à cela :

```
public class MyString : IEquatable<MyString>, IConvertible
   {
```

```
private string value;
public MyString() { }
public MyString(string value)
{
   this.value = value;
}

public string Value
{
    get { return value; }
    set { this.value = value; }
}

public override string ToString() { return value; }

public static implicit operator MyString(string str)
{ return new MyString(str); }

public static implicit operator string(MyString myString)
{ return myString.value; } ...
```

Le type MyString déclare une propriété Value de type string, mais surtout elle déclare deux opérateurs implicites : l'un permettant de convertir une string en MyString, et l'autre s'occupant du sens inverse.

C'est presque tout. Ca marche. Je peux écrire 'MyString x = "toto"' et l'inverse aussi (affecter à une variable de type string directement une variable de type MyString).

Dans la réalité il faudra s'occuper d'autres détails, comme les opérateurs d'égalité par exemple, ou bien les conversions de type (interface IConvertible), etc.

Mais la majorité de ce code peut être directement vampirisé de la classe string puisque la valeur Value est de ce type et que notre classe ne contient rien d'autre à convertir.

On en arrive à un code final de ce type (le nom de la classe est un peu long mais correspond à un cas réel) :

```
public class DictionaryNameString : IEquatable<DictionaryNameString>,
IConvertible
    {
```

```
        private string value;

        public DictionaryNameString() { }

        public DictionaryNameString(string value)
        {
            this.value = value;
        }

        public string Value
        {
            get { return value; }
            set { this.value = value; }
        }

        public override string ToString() { return value; }

        public static implicit operator DictionaryNameString(string str)
        {
            return new DictionaryNameString(str);
        }

        public static implicit operator string(DictionaryNameString
dictionary)
        { return dictionary.value; }

        public bool Equals(DictionaryNameString other)
        {
            if (ReferenceEquals(null, other)) return false;
            return ReferenceEquals(this, other) || Equals(other.value,
value);
        }

        public override bool Equals(object obj)
        {
            if (ReferenceEquals(null, obj)) return false;
            if (ReferenceEquals(this, obj)) return true;
            return obj.GetType() == typeof(DictionaryNameString) &&
                                    Equals((DictionaryNameString)obj);
        }
```

```
        public override int GetHashCode()
        {
            return (value != null ? value.GetHashCode() : 0);
        }

        public static bool operator ==(DictionaryNameString left,
DictionaryNameString right)
        { return Equals(left, right); }

        public static bool operator !=(DictionaryNameString left,
DictionaryNameString right)
        { return !Equals(left, right); }

        #region IConvertible Members

        public TypeCode GetTypeCode() { return TypeCode.String; }

        public bool ToBoolean(IFormatProvider provider)
        { return Convert.ToBoolean(value, provider); }

        public byte ToByte(IFormatProvider provider)
        { return Convert.ToByte(value, provider); }

        public char ToChar(IFormatProvider provider)
        { return Convert.ToChar(value, provider); }

        public DateTime ToDateTime(IFormatProvider provider)
        { return Convert.ToDateTime(value, provider); }

        public decimal ToDecimal(IFormatProvider provider)
        { return Convert.ToDecimal(value, provider); }

        public double ToDouble(IFormatProvider provider)
        { return Convert.ToDouble(value, provider); }

        public short ToInt16(IFormatProvider provider)
        { return Convert.ToInt16(value, provider); }

        public int ToInt32(IFormatProvider provider)
        { return Convert.ToInt32(value, provider); }
```

```
        public long ToInt64(IFormatProvider provider)
        { return Convert.ToInt64(value, provider); }

        public sbyte ToSByte(IFormatProvider provider)
        { return Convert.ToSByte(value, provider); }

        public float ToSingle(IFormatProvider provider)
        { return Convert.ToSingle(value, provider); }

        public string ToString(IFormatProvider provider)
        { return value; }

        public object ToType(Type conversionType, IFormatProvider provider)
        { return Convert.ChangeType(value, conversionType, provider); }

        public ushort ToUInt16(IFormatProvider provider)
        { return Convert.ToUInt16(value, provider); }

        public uint ToUInt32(IFormatProvider provider)
        { return Convert.ToUInt32(value, provider); }

        public ulong ToUInt64(IFormatProvider provider)
        { return Convert.ToUInt64(value, provider); }

        #endregion
    }
```

Et voici une classe “string” personnalisée, utilisable comme string et offrant globalement les mêmes services dans 99% des cas (affectations dans un sens ou dans l’autre, conversions).

Petit plus : notre classe n’est pas “sealed”... Il suffit de l’appeler “MyStringBase” et d’hériter ensuite de cette classe pour se créer des tas de types “string” personnalisés.

En dehors de l’exemple que je donnais, on peut imaginer de nombreux cas où faire un “if (variable is MySpecialString)...” pourra simplifier beaucoup les choses. Tout en conservant une écriture simple et limpide, un code propre et maintenable.

Conclusion

Je parle moins souvent de C# qu’il y a quelques années car les nouveautés se font rares, le langage est stabilisé et commence à être bien connu. Mais ce n’est pas une raison pour ne

pas rappeler certaines de ses possibilités qui sont loin d'être toutes maitrisées et encore moins utilisées fréquemment. Même les choses les moins exotiques.

Gérer les changements de propriétés (Silverlight, WPF, WinRT…)

S'il y a bien une chose qui est "ze" base de la programmation sous .NET quel que soit la technologie d'affichage, c'est bien la notification des changements de valeur des propriétés ! Bizarrement cette fonctionnalité cruciale sur laquelle tout DAL, tout BOL, tout modèle Entity Framework se base, sans lequel MVVM n'existerait pas, ni Prism, ni Jounce, ni rien, bizarrement disais-je, Microsoft n'a jamais rien fait pour l'améliorer, laissant chacun se débrouiller et bricoler sa solution !

INotifyPropertyChanged

Une interface, une pauvre interface ne définissant qu'une seule chose, un évènement "**PropertyChanged**". Au développeur de faire le reste...
Or cet évènement attend en paramètre le nom de la propriété dont la valeur a changé.

En dehors d'être lourd à gérer, répétitif, c'est dangereux ces chaines de caractères qui ne seront pas modifiées lors d'un refactoring par exemple. Sans compter sur les erreurs de frappe.

Et comme tout repose, *in fine*, sur PropertyChanged, la moindre erreur à ce niveau et c'est l'assurance d'un bug pas toujours évident à comprendre et encore moins à localiser.
C'est pourquoi j'ai décidé de faire un tour des différentes manières de gérer cette interface et d'ouvrir la discussion avec vous sur la méthode que vous utilisez ou préférez. Peut-être découvrirez-vous ici certaines astuces que vous n'utilisez pas encore...

La base

Une classe soucieuse de pouvoir participer à la grande aventure qu'est une application .NET se doit sauf rarissimes exceptions de supporter **INotifyPropertyChanged**. C'est le strict minimum.
En réalité, en dehors des instances "immutables" dont on se sert parfois en programmation multithread pour simplifier la gestion des conflits, toutes les classes doivent supporter cette interface.

La méthode la plus basique se résume à l'exemple de code ci-dessous :

```
public class BasicNotify : INotifyPropertyChanged
{
    private string data1;
```

```
    public string Data1
    {
        get
        {
            return data1;
        }
        set
        {
           if (data1 == value) return;
           data1 = value;
           if (PropertyChanged!=null)
             PropertyChanged(this,new PropertyChangedEventArgs("Data1"));
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Une propriété est définie avec un “backing field”, c’est à dire un champ caché (privé). Le getter de la propriété retourne ce dernier, et le setter est un peu plus compliqué : Après avoir vérifié que la valeur a bien changé, le backing field est modifié et l’évènement PropertyChanged est invoqué.

On remarque qu’il faut tester si un gestionnaire d’évènement a bien été associé (test sur de nullité), on voit aussi que le nom de la propriété est passé sous forme d’un chaine de caractères.

La classe supporte bien entendu INotifyPropertyChanged, c’est à dire qu’elle implémente l’évènement public PropertyChanged.

C’est simple et efficace.

Mais il y a des choses qui chiffonnent un peu.

La première bien entendu c’est de passer le nom de la propriété sous forme de chaine. C’est très risqué puisque non contrôlé à la compilation.

Ensuite c’est verbeux. Pour chaque propriété il faudra réécrire le même code d’appel à PropertyChanged.

Enfin ce n'est pas thread safe, puisque dans un environnement multitâche il se peut qu'entre le test de nullité de PropertyChanged et l'appel proprement dit des choses se soient passées... Ainsi au moment du test le PropertyChanged peut ne pas être nul mais peut très bien l'être devenu au moment de l'appel. Et boom !

Une base plus réaliste

Les propriétés sont des bêtes parfois étranges. Toutes ne sont pas de simples "proxy" pour un backing field. Certaines propriétés sont des fantômes ! C'est à dire qu'elle n'ont pas d'existence propre dans l'objet et qu'elles sont élaborées à partir des états courants du dit objet.

Regardons le code suivant :

```
public class BasicNotify2 : INotifyPropertyChanged
{
    private string data1;
    public string Data1
    {
        get
        {
            return data1;
        }
        set
        {
            if (data1 == value) return;
            data1 = value;
            if (PropertyChanged != null)
              PropertyChanged(this, new PropertyChangedEventArgs("Data1"));
            if (PropertyChanged != null)
              PropertyChanged(this,
                 new PropertyChangedEventArgs("DerivedData"));
        }
    }

    public string DerivedData
    {
        get
        {
            return "<" + data1 + ">";
        }
    }
```

```
    public event PropertyChangedEventHandler PropertyChanged;
}
```

La propriété “DerivedData” n’existe pas dans la réalité ... objective de la classe BasicNotify2. C’est une sorte d’artéfact, un pur fantôme dont la valeur évolue dans le temps selon l’état interne de l’objet. Ici le cas est simple, DerivedData ne dépend que de la valeur de “Data”. Parfois la propriété dérivée dépend de plusieurs autres valeurs, toutes n’étant pas forcément des propriétés publiques ce qui complique encore plus la tâche.

Comme on le voit dans le code ci-dessus, DerivedData ne possède qu’un getter. Normal puisqu’elle n’a aucune valeur propre d’arrière plan.

Mais lorsque que “Data” change, il faut s’assurer et surtout ne pas oublier d’émettre un avis de changement de propriété pour “DerivedData” aussi ! C’est pourquoi le setter de Data contient désormais deux appels à PropertyChanged.

Cela ne règle d’ailleurs aucun des problèmes soulevés plus haut, c’est juste plus proche de la réalité.

Créer une notification thread safe

C’est peut-être le premier point, le plus urgent à gérer dans le support de INotifyPropertyChanged car il peut être directement source de bug très difficiles à pister et à corriger.

Voici la classe du second exemple réécrite pour être thread safe (au niveau de PropertyChanged, pas au niveau de la propriété Data ni de la classe elle-même, attention, nuance !) :

```
public class ThreadSafeNotify : INotifyPropertyChanged
{
    private string data1;
    public string Data1
    {
        get
        {
            return data1;
        }
        set
        {
            if (data1 == value) return;
```

```
            data1 = value;
            var p = PropertyChanged;
            if (p == null) return;
            p(this, new PropertyChangedEventArgs("Data1"));
            p(this, new PropertyChangedEventArgs("DerivedData"));
        }
    }

    public string DerivedData
    {
       get
        {
            return "<" + data1 + ">";
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

Qu'ai-je changé ici ?

Peu de choses, mais c'est essentiel. Tout d'abord je fabrique une copie de la référence PropertyChanged, c'est à dire qu'à ce moment précis (p=PropertyChanged) je capture la valeur de PropertyChanged, je la fige dans le temps. Elle peut changer à l'instruction suivante, ce n'est plus mon problème.

Ensuite je teste la nullité comme précédemment mais sur ma valeur copie.

Et seulement si la valeur copie n'est pas nulle, là je peux l'utiliser (toujours elle et non pas PropertyChanged) pour invoquer les gestionnaires d'évènements éventuellement liés.

Peu de choses, mais c'est vraiment important.

Centraliser et simplifier

Comme on le voit sur les exemples de code présentés jusqu'ici, la notification est verbeuse, et puisqu'elle réclame des tests, répéter tout cela pour chaque propriété peut devenir très vite fastidieux.

Il est donc urgent de centraliser un peu le code et de simplifier la mise en œuvre de l'appel à la notification.

```
public class SimplifyNotify : INotifyPropertyChanged
{
    private string data1;
    private int data2;

    public string Data1
    {
        get
        {
            return data1;
        }
        set
        {
            if (data1 == value) return;
            data1 = value;
            doNotify("Data");
            doNotify("DerivedData");
        }
    }

    public string DerivedData
    {
        get
        {
            return "<" + data1 + ">";
        }
    }

    public int Data2
    {
        get
        {
            return data2;
        }
        set
        {
            if (data2==value) return;
            data2 = value;
            doNotify("Data2");
        }
    }
```

```
    private void doNotify(string propertyName)
    {
        var p = PropertyChanged;
        if(p==null) return;
        p(this,new PropertyChangedEventArgs(propertyName));
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

Dans la classe ci-dessus j'ai créé une nouvelle méthode privée "DoNotify" dont le rôle sera justement de faire les tests vis à vis de PropertyChanged et d'appeler ou non la notification. Elle prend aussi en charge la création de l'objet argument.

J'ai ajouté une nouvelle propriété (Data2) pour bien faire voir l'économie d'écriture qu'une telle centralisation procure.

Une classe "Observable"

Quel que soit le nom qu'on lui donne, on voit clairement apparaitre le besoin d'une classe de base offrant par défaut toute la mécanique de base. Finalement sous C# créer une classe c'est toujours dériver d'une classe mère, même si on ne dit rien. Dans ce cas la classe descend de "Object". Ne pas le mettre est un simple raccourci d'écriture, techniquement toute classe descend de Object.

Du coup, comme nous avons vu que la gestion de PropertyChanged était une sorte de passage obligé pour une classe dans une vraie application, autant remplacer Object par une classe de base qui prend en compte la notification de changement des propriétés... Toutes les classes d'une application peuvent descendre de cette nouvelle classe "Observable" sans aucun problème.

La classe de base

Pour l'instant elle va être très simple, elle ne fera que fournir ce service "obligatoire" qu'est la notification de changement de valeur :

```
public class Observable : INotifyPropertyChanged
{
```

```
    protected void DoNotifyChanged(string propertyName)
    {
        var p = PropertyChanged;
        if (p==null) return;
        p(this,new PropertyChangedEventArgs(propertyName));
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

La classe “Observable” offre le support de INotifyPropertyChanged à tous ces descendants ainsi qu’une méthode centrale pour effectuer proprement cette notification “DoNotifyChanged”. On note que cette dernière est désormais “protected” puisqu’on ne veut pas qu’elle puisse être appelée en dehors de l’objet (mais en même temps elle doit pouvoir être appelée depuis tout descendant).

Une classe dérivée

Je reprend ici l’exemple de la classe “SimplifyNotify” en y ajoutant une troisième propriété dont dépend aussi la propriété dérivée. Cela se rapproche plus de la complexité réelle. En revanche cette nouvelle classe hérite de Observable, notre classe de base gérant la notification de changement de valeur de propriété.

```
public class MyObservableType : Observable
{
    private string data1;
    private int data2;
    public string Data
    {
        get
        {
            return Data;
        }
        set
        {
            if (data1==value) return;
            data1 = value;
            DoNotifyChanged("Data");
            DoNotifyChanged("DerivedData");
        }
    }
```

```
    public int Data2
    {
        get
        {
            return data2;
        }
        set
        {
            if (data2==value) return;
            data2 = value;
            DoNotifyChanged("Data2");
            DoNotifyChanged("DerivedData");
        }
    }

    public string DerivedData
    {
        get
        {
            return "{" + data1 + "}" +
              data2.ToString(CultureInfo.InvariantCulture);
        }
    }
}
```

Qu'est-ce qu'il manque ?

Arrivé à ce stade nous avons réglé quelques problèmes :

- la systématisation du support de INotifyPropertyChanged via une classe de base "Observable"
- le contrôle thread safe de l'appel à la notification

Il s'agit de deux des principaux problèmes évoqués au début de ce billet.

Il en reste un troisième, et de taille, le contrôle du nom de la propriété...

Contrôler les noms de propriété

En effet, la pire des choses qui puisse exister c'est le code non typé et non contrôlé à la compilation. Raison pour laquelle je déteste (et c'est un faible mot) tous les langages de type JavaScript. Tous ces machins "dynamiques" ou non fortement typés, sans étape de

compilation qui est le seul garde-fou sérieux contre toute une série de bugs parmi les plus sournois et les plus graves.

Je parle de développer des applications professionnelles, parfois lourdes, souvent de grande taille. Pas de faire un tétris ou le énième lecteur de flux Rss pour IPhone ou Android. Mon chien qui est très bien éduqué pourrait écrire ce genre de truc j'en suis presque sûr ("c'est pas un chien ! c'est mon Toby. Un pt'it bisou ?").

Or, à plusieurs endroits, .NET s'est autorisé des écarts. On l'a vu dans le Binding en Xaml qui offre un langage dans le langage mais non contrôlé, on le voit ici où il faut passer une chaine de caractères pour spécifier le nom de la propriété en cours de changement...

A force de petites concessions stupides (comme les Dynamic en C#) et de libertés comme les chaines de caractères non contrôlée, .NET et C# perdent un peu de leur beauté conceptuelle, de leur pureté, c'est dommage.

Bref, ne croyez pas que cette digression est purement oiseuse, non, elle traduit clairement ma déception devant la gestion de INotifyPropertyChanged et de cette fichue chaine de caractères non contrôlée qu'il faut passer en guise de référence à la propriété en cours.

Donc, il faut contrôler les noms des propriétés si on veut que ce mécanisme, à la base de tout dans une application .NET, ne vienne pas gâcher une belle application.

Des stratégies différentes

Il existe plusieurs tentatives pour régler ce délicat problème. Depuis dix ans j'aurais préféré que la solution vienne de Microsoft dans l'une des versions de C#.

Puisque cela n'est jamais venu, et ne viendra certainement pas, regardons ce qui peut être fait côté développeur.

Les constantes

La première stratégie qu'on peut voir à l'œuvre est l'utilisation de constantes. C'est bien, çà a au moins l'avantage de centraliser les chaines pour les contrôler en cas de doute. Mais hélas le nom lui même de la propriété ne peut pas utiliser cette chaine, du coup il s'agit bien d'un doublon non contrôlé. On ne fait que rendre plus propre les choses en mettant tout ce qui peut poser problème à un seul endroit.

Etant donné que cela ne règle pas le problème, je ne m'attarderai pas sur cette stratégie.

Contrôle par expression Lambda et Réflexion

Ici il s'agit de régler vraiment le problème. Mais il y a un coût : il faudra utiliser la réflexion et cela peut diminuer les performances de l'application, surtout pour les objets dont les propriétés varient très souvent où lorsque que beaucoup d'objets sont manipulés dans une boucle par exemple.

Il faut assumer ce prix si on veut un contrôle permanent, même au runtime, de tous les noms de propriétés.

Partons de notre classe de base et rajoutons le code nécessaire à l'utilisation des expressions Lambda. Tout l'intérêt d'avoir créé une classe base se trouve un peu là, dans la possibilité d'augmenter d'un seul geste les capacités de toutes les classes dérivées.

```
public class ObservableLambda : INotifyPropertyChanged
{
    protected void DoNotifyChanged<T>(Expression<Func<T>> property)
    {
        var member = property.Body as MemberExpression;
        if (member==null)
          throw new Exception("property is not a valid expression");
        DoNotifyChanged(member.Member.Name);
    }

    protected void DoNotifyChanged(string propertyName)
    {
        var p = PropertyChanged;
        if (p == null) return;
        p(this, new PropertyChangedEventArgs(propertyName));
    }
    public event PropertyChangedEventHandler PropertyChanged;
}
```

J'ai volontairement laissé la version en chaine de caractères de DoNotyfichanged. La méthode surchargée qui utilise une expression Lambda s'en sert ce qui permet d'avoir les deux solutions en une.

Comme je le disais l'astuce d'utiliser en paramètre une expression Lambda et ensuite la Réflexion pour extraire le nom de la propriété pose le problème de la dégradation des performances. En laissant les deux possibilités on peut ainsi utiliser systématiquement la version contrôlée pour les objets dont les propriétés changent peu souvent (les propriété d'une fiche client ou article par exemple) et on peut, en assumant le risque, utiliser la

version en chaine de caractères pour des objets spéciaux mis à jour plusieurs fois par secondes (dans un jeu par exemple, ou une classe statistique qui est mise à jour dans un boucle, etc...).

La déclaration de la version avec expression Lambda est intéressante, je vous laisse méditer dessus.... Mais je vais vous montrer un exemple d'utilisation en reprenant la dernière classe "MyObservableType" et en lui faisant supporter notre nouvelle classe de base :

```
public class MyNewObservableClass : ObservableLambda
{
    private string data1;
    private int data2;
    public string Data
    {
        get
        {
            return Data;
        }
        set
        {
            if (data1 == value) return;
            data1 = value;
            DoNotifyChanged(()=>Data);
            DoNotifyChanged(()=>DerivedData);
        }
    }

    public int Data2
    {
        get
        {
            return data2;
        }
        set
        {
            if (data2 == value) return;
            data2 = value;
            DoNotifyChanged(()=>Data2);
            DoNotifyChanged("DerivedData");
        }
    }
```

```
    public string DerivedData
    {
        get
        {
            return "{" + data1 + "}" +
               data2.ToString(CultureInfo.InvariantCulture);
        }
    }
}
```

On voit qu'il suffit de passer une expression lambda très simple à DoNotifyChanged, une expression vide ne retournant que la propriété en cours. Cela sera suffisant pour que le code exposé plus haut puisse extraire le nom de la propriété par Réflexion.

On note aussi que j'ai volontairement laissé un appel avec chaine dans le setter de Data2, afin de montrer que la possibilité existe toujours et quelle sera forcément plus rapide. Le mixage des deux méthodes n'est pas cohérent, c'est juste un exemple.

Le contrôle au Debug

J'aime bien la solution retenue dans MVVM Light : il existe un contrôle utilisant la Réflexion tant qu'on est en debug. Le code de contrôle étant supprimé en mode Release.

C'est une idée séduisante la Réflexion comme le montre la solution précédente. Hélas elle coute cher en temps de calcul. Raison pour laquelle MVVM Light limite son utilisation en mode Debug.

L'approche de MVVM Light est donc différente : des contrôles, mais uniquement en mode Debug. Cela peut paraitre un excellent compromis, il n'est pas mauvais d'ailleurs, mais c'est un peu gênant quand même.

Rien ne dit en effet qu'en Debug le développeur sera passé partout dans le logiciel, aura changé au moins une fois toutes les propriétés de tous les objets... Et c'est en exploitation qu'on tombera sur le problème, d'autant plus difficile à trouver que les informations de Debug ne seront pas forcément là pour aider...

C'est une bonne idée, un entre-deux acceptable, mais c'est un parapluie avec des trous il faut en avoir conscience. Personnellement je préfère l'approche présentée juste avant avec des classes totalement et toujours contrôlées et d'autres non contrôlées où, comme dans

une base de données bien faite on va accepter ponctuellement de “dénormaliser”, ici d’utiliser des chaines, pour des raisons de performance.

Mais je fais le tour des idées, et celle de MVVM Light mérite d’être présentée. D’autant que MVVM Light 4 rajoute le support de la solution avec expression Lambda... Finalement cela devient une solution globale laissant au développeur le choix entre les deux approches tout en bénéficiant d’un contrôle en Debug pour les propriétés passées sous forme de chaines...

Donc dans MVVM Light les choses sont gérées de la façon suivante (j’ai pris la liberté de simplifier le code complet de la classe de MVVM Light 4 pour ne laisser que ce qui concerne notre sujet) :

```
public class ObservableObject : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    /// <summary>
    /// Provides access to the event handler to derived classes.
    /// </summary>
    protected PropertyChangedEventHandler PropertyChangedHandler
    {
        get
        {
            return PropertyChanged;
        }
    }

    [Conditional("DEBUG")]
    [DebuggerStepThrough]
    public void VerifyPropertyName(string propertyName)
    {
        var myType = GetType();
        if (!string.IsNullOrEmpty(propertyName)
            && myType.GetProperty(propertyName) == null)
        {
            throw
             new ArgumentException("Property not found", propertyName);
        }
    }
```

```
    protected virtual void RaisePropertyChanged(string propertyName)
    {
        VerifyPropertyName(propertyName);
        var handler = PropertyChanged;
        if (handler == null) return;
        handler(this, new PropertyChangedEventArgs(propertyName));
    }



    protected virtual void RaisePropertyChanged<T>(Expression<Func<T>>
propertyExpression)
    {
        var handler = PropertyChanged;
        if (handler == null) return;
        var propertyName = GetPropertyName(propertyExpression);
        handler(this, new PropertyChangedEventArgs(propertyName));
    }

    protected string GetPropertyName<T>(Expression<Func<T>>
propertyExpression)
    {
        if (propertyExpression == null)
        {
            throw new ArgumentNullException("propertyExpression");
        }
        var body = propertyExpression.Body as MemberExpression;
        if (body == null)
        {
            throw new ArgumentException("Invalid argument",
                  "propertyExpression");
        }

        var property = body.Member as PropertyInfo;

        if (property == null)
        {
            throw new ArgumentException("Argument is not a property",
                                        "propertyExpression");
        }

        return property.Name;
```

```
    }

    protected bool Set<T>(
        Expression<Func<T>> propertyExpression,
        ref T field,
        T newValue)
    {
        if (EqualityComparer<T>.Default.Equals(field, newValue))
        {
            return false;
        }
        field = newValue;
        RaisePropertyChanged(propertyExpression);
        return true;
    }

    protected bool Set<T>(
        string propertyName,
        ref T field,
        T newValue)
    {
        if (EqualityComparer<T>.Default.Equals(field, newValue))
        {
            return false;
        }
        field = newValue;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

Ce code va un cran plus loin que le contrôle puisqu'il propose même une méthode générique "Set" qui automatise l'ensemble des opérations usuelles pour changer la valeur d'une propriété. C'est une approche très intéressante qui peut se marier d'ailleurs avec la solution de l'expression Lambda, et c'est ce qui est fait dans MVVM Light 4 d'ailleurs.

Si vous lisez bien le code (assez court) vous remarquerez en effet que MVVM Light utilise aussi une variante de la méthode de notification avec expression Lambda... Petite compétition entre frameworks MVVM, disons-le pour rendre à César ce qui lui appartient que c'est Jounce qui a été le premier à proposer cette solution. Mais c'est une saine émulation qui permet que les frameworks évoluent. Comme Jounce et MVVM Light sont

gratuits et sont publiés avec leur code source, on ne peut pas parler de copiage ni de brevets violés et c'est profitable pour tous.

Toujours en repartant du même objet, mais en le pliant à la nouvelle classe mère, voici un exemple d'utilisation de ce code :

```
public class MyNewObservableType : ObservableObject
{
    private string data1;
    private int data2;
    public string Data
    {
        get
        {
            return Data;
        }
        set
        {
            Set(() => Data, ref data1, value);
            RaisePropertyChanged(()=>DerivedData);
        }
    }

    public int Data2
    {
        get
        {
            return data2;
        }
        set
        {
            Set(() => Data2, ref data2, value);
            RaisePropertyChanged(()=>DerivedData);
        }
    }

    public string DerivedData
    {
        get
        {
            return "{" + data1 + "}" +
             data2.ToString(CultureInfo.InvariantCulture);
```

```
            }
        }
}
```

J'utilise ici la possibilité de passer une expression Lambda dans les deux cas en utilisant soit le Set pour la propriété en cours, soit le RaisePropertyChanged pour la propriété dérivée.

En réalité ici ce code est identique à la solution précédente... Il faudrait utiliser des chaines de caractères pour bénéficier du contrôle uniquement en Debug.

Le mode expression Lambda de MVVM Light 4 est exactement comme celui présenté plus haut : permanent.

De fait, MVVM Light 4 permet de mettre en œuvre la stratégie que j'évoquais : des classes toujours contrôlées (propriétés passées en expressions Lambda) et des classes où les performances priment (propriétés passées en chaines).

L'avantage de MVVM Light 4 est que, en Debug, les propriétés passées en chaines seront malgré tout contrôlées. Un peu le beurre et l'argent du beurre.

Pour être complet on notera que j'ai supprimé du code original la partie gérant un évènement PropertyChanging bien intéressant puisqu'on peut ainsi éviter qu'une propriété change de valeur même après qu'elle ait été assignée.

MVVM Light a toujours été un bon framework et ses dernières évolutions renforcent quelques de ses points faibles, même s'il reste fondamentalement différent de Jounce.

Je renvoie le lecteur intéressé par plus de détails sur ces deux frameworks vers les deux mini-livres gratuits que j'ai écrit eux (une simple recherche dans Dot.Blog vous renverra vers le téléchargement des PDF).

Conclusion

La notification du changement de valeur des propriétés est un vaste sujet, bien plus passionnant que le seul Event publié par l'interface ne le laisse supposer...

Ce petit tour d'horizon permet de mieux comprendre les problèmes qui se posent ainsi que d'étudier les principales solutions éprouvées et, peut-être, de vous faire réfléchir à la façon dont vous gérer le problème. Si vous utilisez d'autres approches que celles présentées ici, n'hésitez pas à les présenter, les commentaires sont ouverts pour ça.

C# : initialisation d'instance, une syntaxe méconnue

C# est d'une telle finesse qu'on oublie parfois de les utiliser, habituer à écrire les choses d'une certaine façon. Les initialisations d'instance par exemple disposent d'une syntaxe si ce n'est méconnue en tout cas fort peu utilisée et qui, pourtant, est bien pratique. Une ruse à connaitre…

Initialisation d'instance

C'est très simple, plutôt que d'écrire :

```
var b = new Button();
b.Content = "Ok";
b.Visibility=Visibility.Collapsed
```

Il est plus facile d'écrire :

```
var b = new Button { Content = "Ok", Visibility=Visibility.Collapsed };
```

Rien de sorcier, c'est pratique, plus lisible, bref cela n'a que des avantages.

Une limite à faire sauter

Même si cela est très pratique, là où cela se corse c'est lorsque l'objet créé en contient d'autres.

Prenons un cas concret : une ChildWindow sous Silverlight qui possède donc deux boutons, CancelButton et OkButton, plus, généralement, un TextBlock que nous appellerons TxtMessage.

Si je veux utiliser la même syntaxe réduite pour initialiser une nouvelle instance de ChildWindow je vais me retrouver "coincé" lorsque je vais vouloir adresser le texte du TextBlock.

En effet, écrire :

```
var dialog = new MyChildWindow { TxtMessage.Text = "Coucou!" }
```

Ça ne passe pas…

Je ne peux pas déférencer la propriété Text à l'intérieur de la propriété TxtMessage de la ChildWindow.

Coincé ?

C'est ce qu'on pense généralement, du coup on extrait de la séquence d'initialisation qui ne passe pas et on se retrouve avec un code spaghetti, une partie des propriétés initialisées avec la syntaxe réduite, et en dessous le reste, initialisé "normalement" (du genre "dialog.TxtMessage.Text="Coucou!"").

La feinte à connaitre

C# nous révèle souvent des surprises quand on prend le temps de lire la documentation de sa syntaxe… Mais on oublie souvent de tout lire, pensant déjà connaitre le principal.

Justement, c'est dans le détail que les choses se jouent…

Voici donc comment écrire en syntaxe courte l'initialisation donnée en exemple plus haut :

```
var dialog = new MyChildWindow { TxtMessage = { Text = "Coucou!"}  };
```

Etonnant non ? Affecter le résultat d'une opération est un truc connu (var a = (b = 2x3); donnera à "a" la valeur du résultat tout en l'affectant déjà à "b"). Mais ici c'est quelque chose d'autre…

Vous noterez qu'après le signe égal un niveau d'accolades américaines supplémentaire est ouvert. Ce qui est affecté à TxtMessage n'est donc pas le résultat de l'affectation "Text="Coucou!"" car cela planterait (on ne peut pas affecter une String à un TextBlock les types ne sont pas compatibles).

Cette syntaxe permet en réalité d'ouvrir une "sous affectation" sur la propriété indiquée (ici on créé on ouverture sur les propriétés de TxtMessage qui est lui même une propriété de la child Window).

Le résultat est celui escompté, à la sortie de l'initialisation le TextBlock portant le nom TxtMessage placé à l'intérieur de MyChildWindow aura bien son texte initialisé à "Coucou!".

Fantastique C# non ?

Moi il me fascine toujours 

Une Preuve

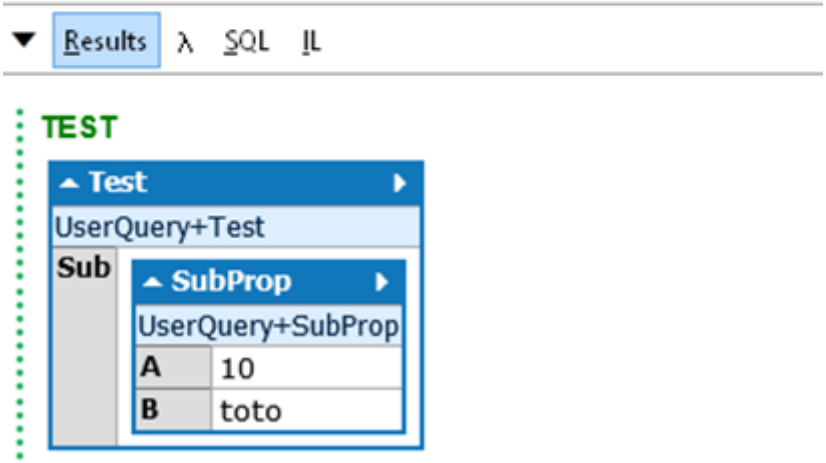
Le plus simple pour tester des petits trucs comme cela c'est d'utiliser LinqPad, un outil indispensable.

Ainsi, voici un exemple tapé et visualisé sous LinqPad qui illustre l'utilisation de cette syntaxe spéciale et son résultat :

```
void Main()
{
var t = new Test { Sub = {A = 10, B="toto" }};
t.Dump("TEST");
}

// Define other methods and classes here
class Test
{
  public SubProp Sub { get; set; }
  public Test() { Sub = new SubProp(); }
}

class SubProp
{
public int A {get; set;}
public string B {get; set;}
}
```



Conclusion

Je ne sais pas combien d'entre vous connaissaient déjà cette syntaxe et l'avaient utilisée. Personnellement j'avoue bien humblement que si Resharper ne me l'avait pas proposée je ne saurai toujours pas que c'est possible, et pourtant j'ai été MVP C# (honte sur moi !).

Multithreading simplifié

Le multithreading c'est l'épouvantail du développeur. Vous en parlez, hop! tout le monde s'en va de la machine à café... et s'il y en a un qui ne part pas, c'est le genre fanatique qui va débaler une science opaque sur les AppDomains, les mutex et autres mots qui fâchent, du coup, c'est vous qui partez :-)

Je carricature à peine...

C'est tout le problème du multithreading. Pratiqué avec simplicité c'est une technique de plus en plus indispensable pour tirer partie des microprocesseurs multicoeurs et fluidifier les interfaces, mais voilà, comment faire simple avec une telle technique ?

Les puristes vous diront qu'il faut absolument comprendre la technique, et qu'en suite c'est facile... Un peu comme Coluche qui expliquait dans l'un de ses sketchs que son professeur de violon lui avait dit d'apprendre à jouer avec des gants de boxe parce que quand on les enlève ça semble facile...

Je ne vais pas vous dire qu'une démarche rigoureuse est inutile, j'ai un module de multithreading avancé dans mes plans de cours et, bien entendu, voir les choses en profondeur au sein d'une formation est le seul moyen de maîtriser cette technique. Mais il existe aussi des façons simples d'introduire un peu de multitâche dans vos applications.

Il s'agit du composant BackgroundWorker des Windows Forms. Certes le sujet tranche avec mes billets généralement plus orientés vers les super nouveautés hypra fraîches à tel point qu'elles sont même parfois en bêta... Mais il faut bien maintenir les applications existantes, les améliorer, et pour cela il existe, comme le BackgroundWorker des solutions pratiques qui ne nécessitent pas d'installer le framework 3.5 puisque cette classe a été fournie avec .NET 2.0.

De plus, ce composant Windows Forms n'impose pas de connaître les mécanismes du multithreading, il suffit de programmer ces événements comme un bouton. Trop facile ? Peut-être que cela choquera les puristes parce que "cela cache la réalité de ce qui se passe vraiment dans la machine", je leur répondrais que faire du C# au lieu de faire de l'assembleur c'est un peu pareil... Là où je les rejoindrais c'est que, bien entendu, la classe backgroundWorker ne doit pas être utilisée à tord et à travers. Si l'on désire concevoir des classes gérant finement le multitâche, il faut réellement comprendre et donc apprendre. Mais dans de nombreux cas, le BackgroundWorker pourra vous être utile et rendre plus fluide vos applications Windows Forms sans avoir à entrer dans les détails d'une technique un peu aride.

Mais trève de mots, le plus simple c'est de jouer avec ce composant pour se rendre compte de son utilité. Pour facilité la.. tâche...
Un peu de douceur multitâche dans ce monde de multicoeurs...

Appels synchrones de services. Est-ce possible ou faut-il penser "autrement" ?

Silverlight ne gère que des appels asynchrones aux Ria Services et autres communications WCF. Le Thread de l'UI ne doit jamais être bloqué assurant la fluidité des applications. Mais comment régler certains problèmes très basiques qui réclament le synchronisme des opérations ? Comme nous allons le voir la solution passe par un inévitable changement de point de vue et une façon nouvelle de penser l'écriture du code.

Ô asynchronisme ennemi, n'ai-je donc tant vécu que pour cette infamie ? ...

Corneille, s'il avait vécu de nos jours et avait été informaticien aurait peut-être écrit ainsi cette célèbre tirade du Cid.

L'asynchronisme s'est immiscé partout dans la programmation et certains environnements comme Silverlight le rendent même obligatoire alors même que les autres plateformes .NET autorisent aussi des communications synchrones.

Avec Silverlight tout appel à une communication externe (web service, Ria services...) est par force asynchrone.

Lutter contre l'asynchronisme c'est comme mettre des sacs de sables devant sa porte quand la rivière toute proche est en crue : beaucoup de sueur, de travail inutile, pour un résultat généralement pas suffisant, l'eau finissant toujours par trouver un passage...

Or on le voit encore tous les jours, il suffit même d'une recherche sous Google ou Bing pour s'en convaincre, nombre d'informaticiens posent encore des questions comme "comment faire des appels synchrones aux Ria services ?".

La réponse est inlassablement la même : ce n'est pas possible ou le prix à payer est trop cher, mieux vaut changer d'état d'esprit et faire autrement.

Autrement ?

Quand on voit certains bricolages qui utilisent des mutex, des threads joints ou des ManualResetEvents ou autres astuces plus ou moins savantes on comprend aisément que ce n'est pas la solution. D'abord toutes ces solutions, quand elles marchent, finiront pas bloquer le thread principal si l'action à contrôler s'inscrit dans une manipulation utilisateur,

et c'est mal. On ne bloque pas le thread principal qui contrôle l'UI, c'est de la programmation de grand-papa qui paralyse l'interface et offre une expérience utilisateur déplorable.

Il faut donc pratiquer d'une autre façon. Mais c'est plus facile à dire qu'à faire.

Pour mieux comprendre le changement d'état d'esprit nécessaire je vais prendre un exemple, celui d'un Login.

Un exemple réducteur mais parlant

Imaginons une application Silverlight qui ne peut être utilisée qu'après s'être authentifié. Nous allons faire très simple dans cette "expérience de pensée" car je ne montrerai pas de code ici. C'est la façon de penser qui compte et que je veux vous expliquer.

Donc imaginons une telle application. Le développeur a créé une ChildWindow de Login qui apparait immédiatement au chargement de l'application. Cette fenêtre modale (pseudo-modale) est très simple : un ID et un mot de passe à saisir et un bouton de validation.

Si le login est correct la fenêtre se ferme, s'il n'est pas valide la fenêtre change son affichage pour indiquer un message d'erreur et proposer des options : retenter sa chance, se faire envoyer ses identifiants par mail ou bien créer un nouveau compte utilisateur.

Je n'entrerai pas dans les détails de ce mécanisme, vous pouvez je pense aisément imaginer comment cela pourrait se présenter.

Comme le développeur de cette application fictive a beaucoup appris de la "componentisation", de la "réutilisabilité" et autres techniques visant à ne pas réécrire cent fois le même code, il a décidé de créer un UserControl qui va gérer tout le dialogue de login et ses différentes options.

C'est un bon réflexe.

Mais, forcément, le contrôle ne peut pas faire des accès à la base de données puisque celle-ci est spécifique à une application donnée. Toujours en suivant les canons de la réutilisabilité notre développeur se dit "lorsque l'utilisateur cliquera sur la validation de son login, puisque je ne peux pas valider ce dernier dans mon contrôle, il faut que j'émette un évènement.".

Bonne façon de penser. Cela s'appelle la délégation, un style de programmation rendu célèbre par Visual Basic et Delphi il y a déjà bien longtemps... C'est grâce à la délégation qu'on peut construire des contrôles réutilisable et c'est une avancée majeure.

Par exemple la classe Button propose un évènement Click. Il suffit de fournir l'adresse d'une méthode qui gère ce Click et l'affaire est jouée. Le contrôle ne sait rien du code qui sera utilisé donc il peut être réutilisé dans milles circonstances, il suffira à chaque fois de lui fournir l'adresse de la méthode qui réalisera le travail. Je parle d'adresse de méthode car originellement c'est bien de cela qu'il s'agit. Même aujourd'hui cela fonctionne de cette manière mais .NET cache les adresses et gère une collection permettant à plusieurs bouts de code de venir "s'abonner" au Click d'un seul bouton.

C'est une évolution qui rend la réutilisabilité encore meilleure et le codage encore plus simple mais qui, fondamentalement, reste basée sur la même technique.

De fait, pour en revenir au contrôle de Login, l'idée du développeur est simple : mon contrôle ne sait pas si le login est valide ou non, mais il a besoin de le savoir pour adapter sa réponse (on ferme la fenêtre car le login est ok, on affiche l'écran d'erreur dans le cas contraire). Pour régler ce problème, je décide donc de créer un évènement "CheckLogin" auquel s'abonnera l'application utilisatrice du contrôle. Dans les arguments de l'évènement je prévois une propriété booléenne "IsLoginOk" que le gestionnaire de l'évènement positionnera à true ou false.

Ce développeur pense logiquement et pour l'instant on ne voit pas ce qui cloche...

Le principe qu'il utilise ici est le même que celui qu'on trouve d'ailleurs un peu partout dans le Framework .NET. En suivant un si brillant modèle comment pourrait-il être dans l'erreur ?

Regardons par exemple les évènements de type KeyUp ou KeyDown. Eux aussi ont une propriété transportée dans l'instance des arguments, "Handled" qui permet au gestionnaire de retourner une valeur booléenne indiquant si l'évènement doit suivre son cours (false) ou bien si le contrôle doit considérer que la touche a été gérée et qu'il ne faut pas faire remonter l'information aux autres contrôles (Handled = true).

Jusqu'ici tout semble être écrit dans le respect des bonnes pratiques de notre métier : componentisation pour une meilleure réutilisabilité, utilisation de la délégation, utilisation des arguments pour permettre la remontée d'une information à l'appelant en suivant le modèle du Framework.

Mais ça coince où alors ?

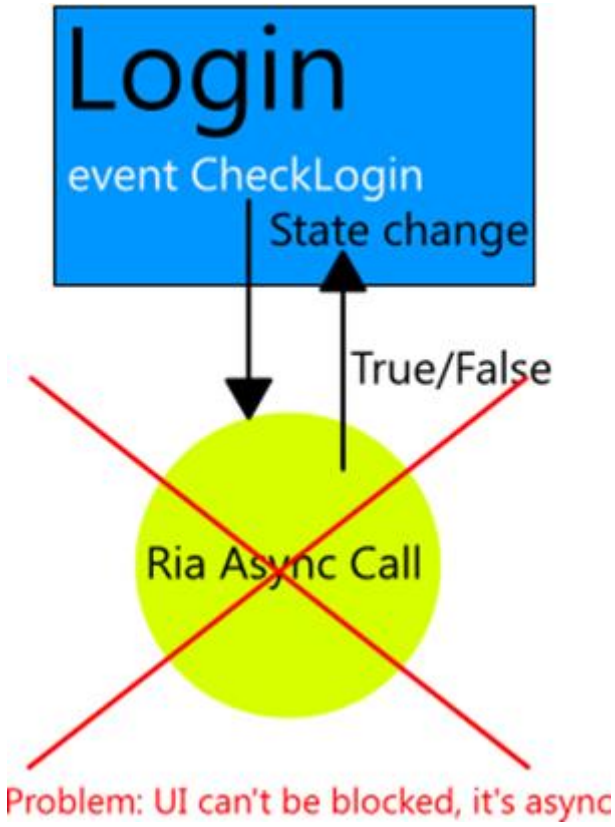
J'y viens... C'est vrai que le décor, pourtant simple, n'est pas si évident que cela à "raconter". Mais je pense que vous avez saisi l'affaire.

Un contrôle de Login qui expose un évènement “CheckLogin” qui délègue le test d’authentification à l’application et qui récupère la réponse de cette dernière dans les arguments de l’évènement pour savoir ce qu’il doit faire.

Le contrôle est bien développé et pour savoir quelle “page” afficher, il utilise même la gestion des états visuels du VSM de Silverlight. Selon cet état les différents affichages possibles seront montrés à l’utilisateur. L’état “LoginState” montrera la page demandant l’ID et le mot de passe, l’état “LoginErrorState” affichera la page indiquant qu’il y a erreur d’authentification, l’état “CreateNewAccountState” affichera une page autorisant la création d’un nouveau compte client.

Franchement il n’y a rien à dire, ce contrôle est vraiment bien développé !

Hélas non...



Regardez le petit schéma ci-dessus.

Le rectangle bleu c’est notre contrôle de Login. Lorsque l’utilisateur valide son authentification l’évènement CheckLogin se déclenche. En face, dans l’application, il faudra bien appeler quelque chose pour répondre si oui ou non l’utilisateur est vraiment reconnu.

Pour cela l'application utilise les Ria Services avec à l'autre bout un modèle Entity Framework et une base de données SQL contenant une table des utilisateurs.

Or, l'appel Ria Service est asynchrone par nature, donc non bloquant.

Le gestionnaire d'évènement accroché à "CheckLogin" va retourner immédiatement les arguments au contrôle. De fait celui-ci ne récupèrera jamais la bonne valeur d'authentification mais la valeur par défaut de "IsLoginOk" contenu dans les arguments. Par sécurité cette valeur est initialisée à false, donc en permanence le contrôle va afficher l'écran d'erreur de login... Même si l'utilisateur est connu car la réponse arrivera bien après que l'argument sera retourné vers le contrôle de Login qui sera déjà passé à la page d'erreur...

Cela ne marche pas car l'appel à une requête Ria Services n'est pas bloquant. Il est donc impossible dans le gestionnaire d'évènement "CheckLogin" de faire l'authentification et de retourner celle-ci au contrôle de Login à temps.

C'est là qu'entrent en scène les fameux bricolages que j'évoque plus haut pour tenter de rendre bloquant l'appel au service externe.

Une autre façon de penser

Nous l'avons vu, en suivant la logique du développeur (fictif) qui a créé ce contrôle on ne détecte aucun défaut, aucune violation des bonnes pratiques. Pourtant cela ne marche pas.

Tenter de rendre synchrone l'appel asynchrone au service externe est peine perdue. Bidouillage sans intérêt ne réglant en rien le problème. Il faut d'emblée s'ôter cette idée de la tête.

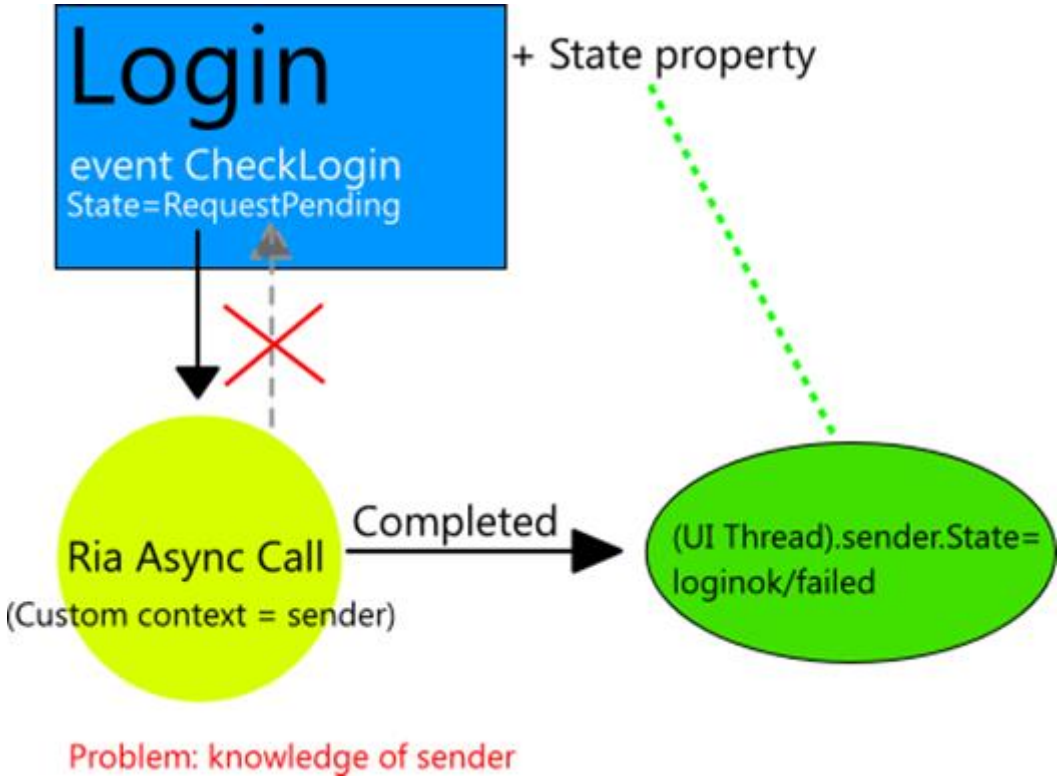
Alors ?

Alors, il faut penser autrement. Je l'ai déjà dit. Bravo aux lecteurs qui ont suivi 

L'approche par la ruse

En réfléchissant un peu, tout bon développeur est capable de "ruser", de trouver une feinte. Ici tout le monde a compris qu'il faudrait découpler la demande d'authentification et le retour de cette information.

Après quelques cogitations notre développeur a imaginé le schéma suivant :



Il y a toujours un évènement CheckLogin dans le contrôle qui est déclenché lorsque l'utilisateur clique sur le bouton servant à valider son login. Toutefois on note plusieurs changements :

- L'évènement ne tente plus de récupérer l'information d'authentification, il ne sert qu'à déclencher la séquence d'authentification, ce qui est différent.
- Le gestionnaire d'évènement de l'application, celui qui va faire l'appel asynchrone, mémorise dans un "custom context" l'adresse du Sender (donc du contrôle de Login).
- Le contrôle gère maintenant un état, la propriété State. Quand l'évènement est déclenché cet état est à "RequestPending" (une requête est en attente).
- Dans l'évènement Completed de la requête asynchrone (quel que soit sa nature, ici Ria Services mais ce n'est qu'un exemple), en passant par un Dispatcher permettant d'utiliser le Thread de l'UI, le code de réponse asynchrone va directement modifier l'état du Contrôle de Login (sa propriété State) en réutilisant l'adresse du Sender mémorisée plus haut.

Cette approche n'est pas sotte, elle permet en effet de découpler l'évènement Checklogin de la réponse d'authentification qui arrivera plus tard. En utilisant un Dispatcher et en ayant pris soin de mémoriser le Sender de CheckLogin, le code asynchrone peut en effet modifier l'état du contrôle et le faire passer à "LoginOk" ou à "LoginFailed" par exemple, ce qui déclenchera alors le bon comportement du contrôle.

Un grand pas vient d'être sauté : ne plus lutter contre l'asynchronisme et concevoir son code "autrement" pour qu'il s'y adapte sans gêne.

Si cette ruse n'est pas idiote, elle pose tout de même des problèmes. Le plus gros étant la nécessité pour le code asynchrone de l'application de connaitre la classe du Contrôle de Login pour attaquer sa propriété State (et connaitre aussi l'énumération qui permet de modifier State).

Ce n'est pas très propre... Cela viole même la séparation UI / Code de MVVM.

Pas bête, bien tenté, mais ce n'est pas encore la solution...

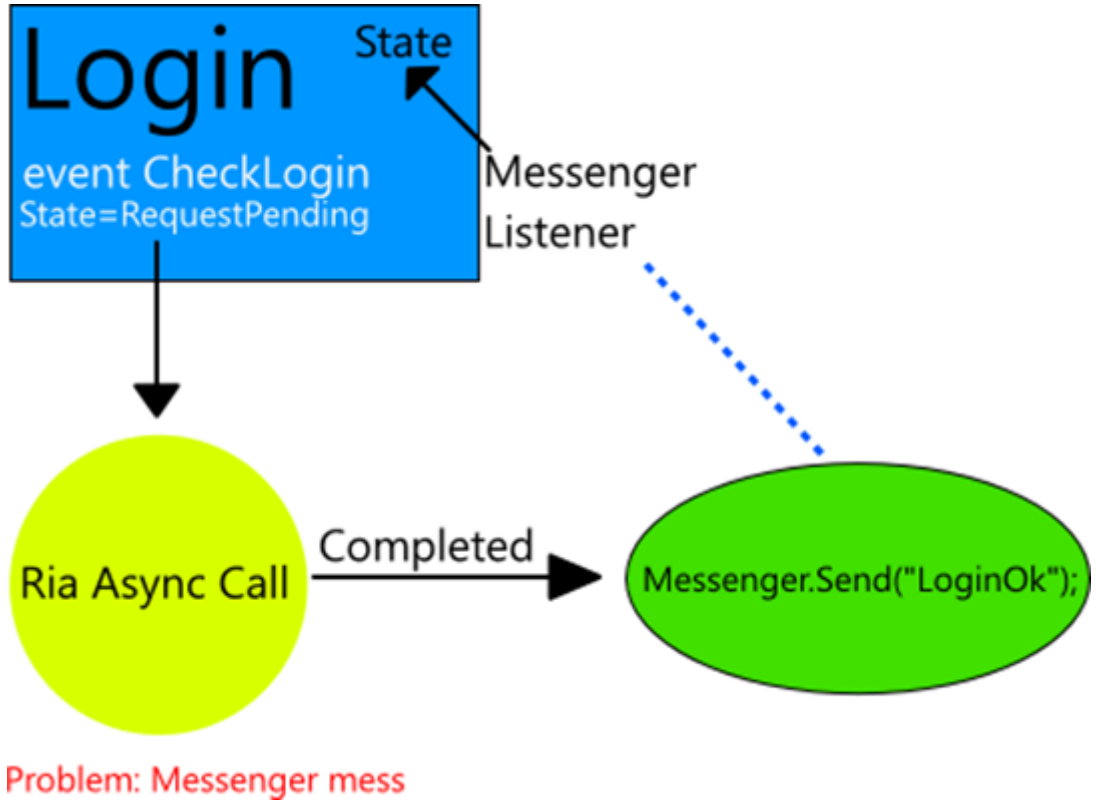
L'approche par messagerie

Cent fois sur le métier tu remettras ton ouvrage ...

Dans un environnement où l'on suit le pattern MVVM on dispose le plus souvent d'une messagerie. Que cela soit MVVM-Light, Jounce ou d'autres framework, tous proposent un tel mécanisme car il permet de résoudre certains problèmes posés par l'implémentation de MVVM.

Ainsi, il est tout à fait possible de revoir la logique du contrôle de Login en jouant cette fois-ci sur des échanges de messages.

Le nouveau schéma devient alors :



Bon ! Cette fois-ci on doit y être ! Le découplage entre l'évènement CheckLogin et la réponse asynchrone est préservé et le découplage UI / Code cher à MVVM est aussi respecté !

Dans ce scénario on retrouve bien entendu l'évènement CheckLogin qui sert de déclencheur à la séquence d'authentification.

On retrouve aussi la propriété State dans le Contrôle de Login.

Mais dans l'évènement Completed de l'appel asynchrone, au lieu d'accéder directement au Sender de CheckLogin et de modifier directement son état, le code se contente d'envoyer un message. Par exemple "LoginOk" ou "LoginFailed".

Tout semble désormais parfait !

Enfin presque...

Ce n'est pas que je veuille à tout prix jouer les rabat-joie, mais ça cloche toujours un peu.

Un exemple ? Prenez le Contrôle de Login lui-même. Il est maintenant obligé de s'abonner à la messagerie pour capter le message qui sera envoyé par le code asynchrone. Côté réutilisabilité personnellement cela me chiffonne. La messagerie ce n'est pas un composant

du Framework Silverlight. Il en existe autant que de toolkit MVVM. Cela veut dire que notre contrôle est “marié” désormais à un Framework donné et que même dans une mini application ne nécessitant pas de framework MVVM il faudrait s’en trimbaler un.

De plus les messageries MVVM je m’en méfie comme de la peste. Très vite on arrive à ce que j’appelle du “message spaghetti”, quelque chose de pire que le code spaghetti. Des classes spécialisées de messages qui se baladent de ci de là, des messages portant des noms en chaines de caractères, un ballet incontrôlable, quasi non maintenable de messages qui transitent partout dans un ordre difficile à prédire...

J’ai testé et croyez-moi c’est difficilement acceptable comme solution en pratique. Jounce propose un logger qui autorise le traçage des messages, c’est déjà beaucoup mieux que MVVM light qui ne possède aucun moyen de vérifier les messages transmis.

Donc ici ce qui ne va pas c’est cette dépendance à une messagerie qui dépend elle même d’un code externe. Notre contrôle n’est plus indépendant, il devient ainsi plus difficilement réutilisable, ce qui était la motivation de sa création. On doit pouvoir trouver mieux.

Il n’en reste pas moins vrai que cette solution est la plus acceptable de toutes celles que nous venons de voir.

Si seulement on pouvait éviter cette satanée messagerie... La dépendance à un toolkit MVVM on peut faire avec. Après tout on ne change pas de toolkit à chaque application et on suppose que le développeur restera fidèle à celui qu’il finira par bien maitriser. Cette dépendance à du code externe continue à me chiffonner, mais elle peut s’accepter.

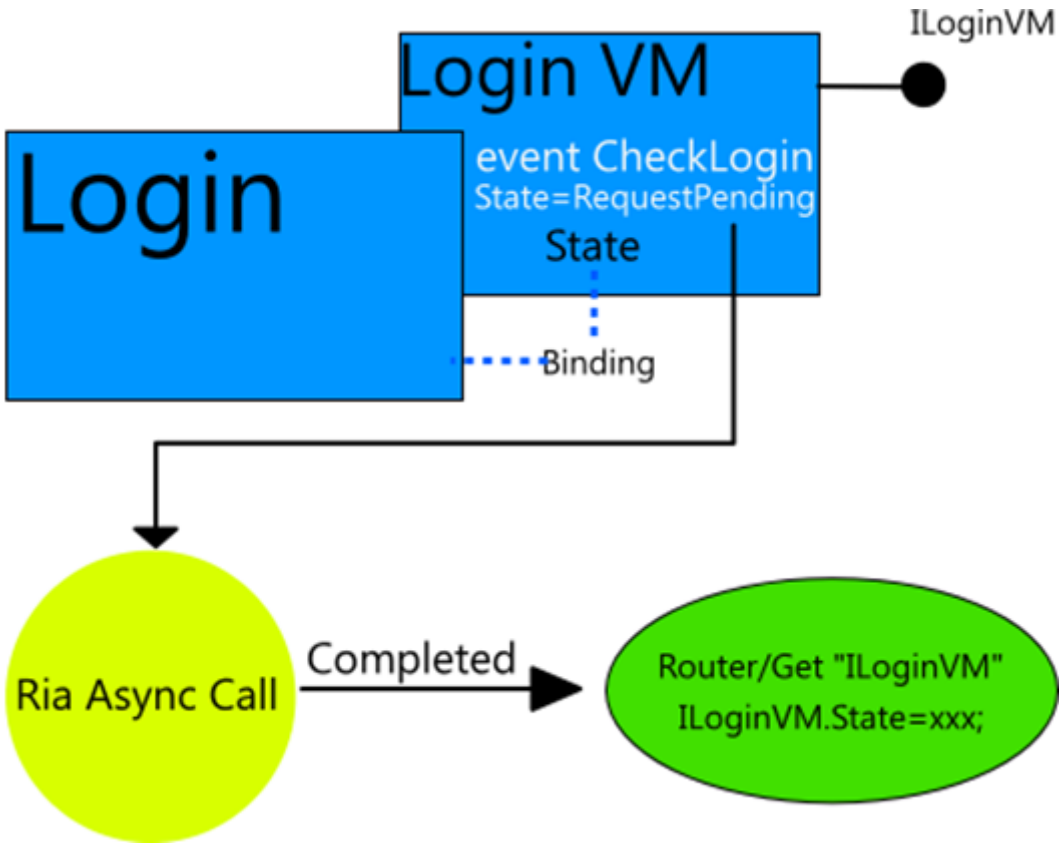
Mais la messagerie. Brrr. Ca me glace le sang. D’autant plus que le Contrôle de Login devra répondre à un message précis qui sera codé en dur. Il faudra bien documenté tout cela pour que les applications qui s’en serviront sachent quel message il faut utiliser, s’il s’agit d’une notification en chaine de caractères il faudra même se rappeler de sa casse exacte. Pas d'IntelliSense ici.

Tout ce qui dépend d’un savoir occulte en programmation me rend méfiant. Non par crainte maladive, mais parce que je sais que cela rend le code inutilisable dans le temps, que cela complique la maintenance et rend l’intégration d’un nouveau développeur dans une équipe particulièrement pénible (lui transmettre toutes ces petites choses non évidentes mais indispensables).

Découplage maximum

Peut-on découpler encore plus les intervenants dans notre scénario et surtout nous passer de la messagerie ?

Regardez le schéma suivant :



Puisque nous suivons le pattern MVVM, autant le faire jusqu'au bout. Et puisqu'il faut choisir un toolkit, j'opte pour Jounce.

Dans un tel contexte je règle le problème de la façon suivante :

- Le Contrôle de Login possède lui aussi un ViewModel.
- C'est ce VM qui porte et expose la propriété State.
- L'évènement CheckLogin est bien entendu géré par le VM. S'agissant d'un UserControl il sera malgré tout "repiqué" dans ce dernier pour être exposé à l'extérieur (l'application utilisatrice), ce qui n'est pas montré ici.
- L'état visuel de la Vue est bindé à la propriété State de son VM. Cela pour rappeler que maintenant nous avons une Vue et un ViewModel et que la première va communiquer avec le dernier par le biais du binding et des ICommand.
- Le VM implémente l'interface ILoginVM selon un mode favorisé par Jounce (non obligatoire mais comme on le voit dans cet exemple qui permet un découplage fort entre les différents codes).

- L'appel au déclenchement de la séquence d'authentification par l'évènement CheckLogin reste identique.
- L'évènement Completed du code asynchrone utilise le Router de Jounce pour obtenir le VM du contrôle de Login, mais uniquement au travers de son interface ILoginVM, donc sans rien connaitre de la classe réelle qui l'implémente.
- Grâce à cette indirection, le code asynchrone Modifie directement la propriété State du VM, ce qui déclenchera automatiquement la mise en conformité visuelle de la Vue.

Plus de messagerie ! Jounce permet ce genre de choses car il est un peu plus sophistiqué que MVVM Light. Le Router enregistre la liste de toutes les "routes", c'est-à-dire de tous les couples possibles "Vue / ViewModel". Puisque le VM implémente une interface il est possible d'obtenir celle-ci plutôt que le VM réel. On conserve ainsi un découplage fort entre le code de l'application qui ne sait rien de l'implémentation réelle du VM du Contrôle de Login.

Le schéma le montre bien visuellement, on a bien deux partie très différentes et bien différenciées surtout : en haut le Contrôle de Login qui ne sait rien de l'application, en bas l'application et son code asynchrone qui ne sait rien du Contrôle de Login et qui ne connait qu'une Interface (et l'évènement CheckLogin).

Pensez-vous "autrement" ?

C'est la question piège : arrivez-vous maintenant à penser "autrement" que la logique du premier exemple ? Car bien entendu tout ce billet porte sur cette question et non pas la mise en œuvre d'un Contrôle de Login...

Avez-vous capté le glissement progressif entre le premier et le dernier schéma ? Cette transformation qui fait que désormais l'asynchronisme n'est plus un ennemi contre lequel on cherche l'arme absolue mais un mécanisme naturellement intégré à la conception de toute l'application ?

Voyez-vous pourquoi je parlais de bricolages en évoquant toutes les solutions qui permettraient de rendre bloquant les appels asynchrones ?

Conclusion

On pourrait se dire que la dernière solution substitue à la connaissance d'un message celle d'une Interface et qu'il existe donc toujours un lien entre l'application et le Contrôle de Login. C'est un peu vrai.

En fait, mon expérience me prouve qu'il faut limiter l'usage des messageries sous MVVM sous peine de se retrouver avec un fatras non maintenable. Je préfère un code qui implémente une Interface qu'un autre qui utilisera la messagerie.

La solution de l'Interface est aussi plus facilement portable. C'est un procédé légitime du langage C#, la messagerie est un mécanisme "propriétaire" dépendant d'un toolkit précis.

Mais le plus important n'est pas d'ergoter sur les deux derniers schémas, le plus essentiel c'est bien entendu que vous puissiez vous rendre compte comment "penser autrement" face à l'asynchronisme et comment passer d'une logique dépassée et inopérante à une logique de codage en harmonie avec les nouvelles contraintes.

Ne vous posez plus jamais la question de savoir comment rendre bloquant un appel asynchrone sous Silverlight.

Demandez-vous systématiquement comment concevoir autrement votre code pour qu'il s'adapte sans peine ni bricolage à l'asynchronisme...

(en passant, une autre chose à ne pas oublier : Stay Tuned !)

NB: les schémas ont été réalisés sont Expression Design qui n'est pas fait pour cela, mais c'était une façon de parler de ce soft que j'aime beaucoup et qui n'est pas assez connu (et qui est mort au moment où je mets en page ce PDF…) !

Parallel FX, P-Linq et maintenant les Reactive Extensions…

Les Parallel Extensions, connues jusqu'à lors sous le nom de Parallel Framework Extensions (ou PFX) forment une librairie permettant de faciliter la construction d'algorithmes parallèles (multi-thread) tirant partie des machines multi-cœur. Je vous en avais déjà parlé, ainsi que de P-Linq les extensions parallèles pour LINQ. Deux choses importantes à savoir aujourd'hui : les Parallel Extensions font partie de .NET 4 et une nouvelle librairie arrive, les Reactive Extensions !

Parallélisme

J'avais abordé le sujet dans mon billet "La bombe Parallèle ! PLINQ, et PCP Parallel Computing Platform" il y a 2 ans presque jour pour jour… Pour ceux qui n'ont pas suivi je rappellerai donc juste quelques points essentiels développés dans ce billet :

- La loi de Moore reste valable mais l'équivalence gain de puissance = fréquence du processeur plus élevée, elle, est morte…

- Pour continuer à faire évoluer la puissance des ordinateurs les fondeurs optimisent les puces mais surtout *multiplient les cœurs* au sein d'une même puce.

- Il y a deux ans, les double-cœurs se banalisaient. Les QuadCore se sont banalisés entre temps et pour une poignée de dollar ont achète aujourd'hui par exemple des i870 à 4 cœurs multithreadés (donc 8 cœurs apparents). Demain les 32, 64 cores seront la norme.

- Pour tirer partie de cette puissance le développeur ne peut plus se dire "si mon client trouve mon soft trop lent, il n'a qu'à acheter une machine plus rapide". Cela ne servira à rien… *Aujourd'hui la responsabilité est totalement du coté du développeur qui doit apprendre à utiliser tous les cœurs des machines en même temps* s'il veut que ses programmes profitent de la puissance des nouvelles machines. Pas d'échappatoire.

- En dehors de quelques geeks, je le vois bien autour de moi et chez mes clients, peu de développeurs semblent avoir bien saisi ce que cela impliquait…

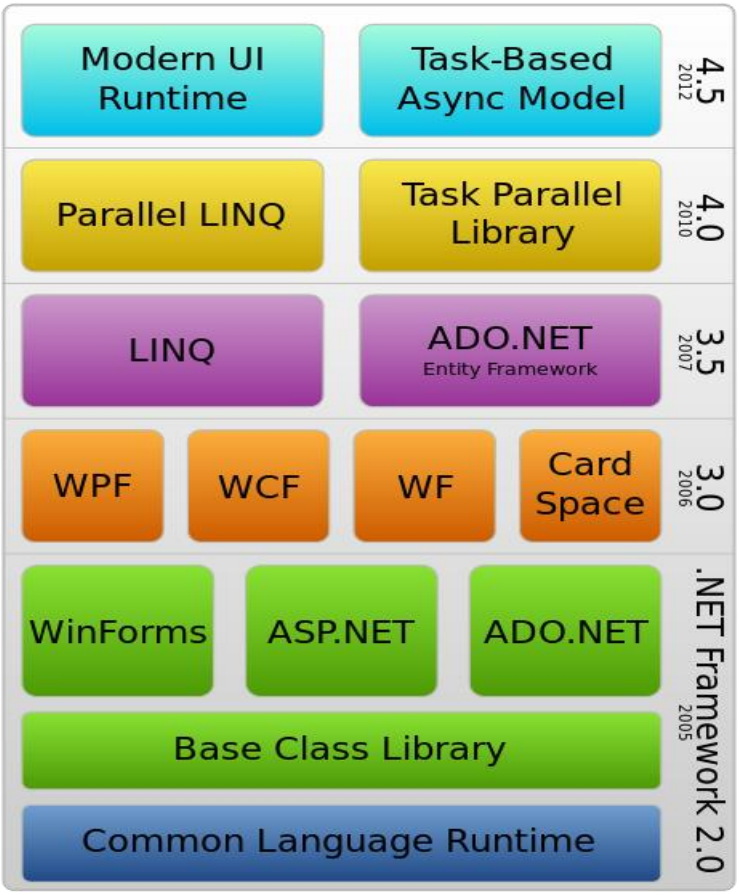
- Dès qu'on créé un nouveau thread dans une application on peut dire qu'on "fait du parallélisme". Certes, ce n'est pas faux. Donc beaucoup de développeurs l'ont fait au moins une fois dans les dernières années, un peu comme M. Jourdain faisait de la prose sans le savoir…

Mais cela n'a rien à voir avec le vrai parallélisme qui implique des langages, des librairies, des plateformes étudiés pour et *surtout une nouvelle façon de penser le code* ! Programmer en asynchrone est particulièrement déroutant (on voit comment certains ont du mal sous Silverlight avec ce concept qui devient incontournable dans certaines situations).

Pour les anglophones je vous renvoie à un autre de mes billets listant articles et vidéos sur le sujet, c'est une introduction toujours valable au sujet : "Mondes Parallèles [Webcasts sur le parallélisme + article]"

Le Framework .NET et le parallélisme

Le petit schéma ci-dessous (repiqué de Wikipédia) rappelle l'architecture simplifiée du Framework .NET ainsi que les ajouts essentiels selon les versions. On remarque au sommet, version 4.0, que le parallélisme fait son entrée "officielle" après quelques années de béta test :



The .NET Framework Stack

Sous .NET 4.0 les choses se présentent ainsi en deux volets distincts: La Task Parallel Livrary, et Parallel LINQ (P-LINQ).

P-LINQ

P-LINQ est une extension parallèle du moteur LINQ. Il parallélise l'exécution des requêtes automatiquement, pour l'instant dans les saveurs Linq to Xml et Linq to Objects. Tous les objets peuvent être utilisés, toutes les requêtes existantes peuvent être transformées pour utiliser P-LINQ (ou presque). Il faut en effet que les listes d'objets supportent IParallelEnumerable une interface de P-LINQ qui utilise TPL pour son exécution (mais il existe une méthode System.Linq.ParallelEnumerable.AsParallel qui transforme une liste IEnumerable<T> en une liste utilisable sous P-LINQ, rien de compliqué donc !).

Vous pouvez lire en complément cet article de MSDN Magazine (en français) : Exécution de requêtes sur les processeurs multicœur.

TPL (Task Parallel Library)

TPL est la librairie qui expose les facilités du moteur parallèle de .NET 4.0. C'est TPL qui offre, notamment, des extensions comme For ou ForEach entièrement parallèles bien qu'utilisant des appels de méthodes et des delegates standard. Sous cette forme, tous les langages .NET

peuvent bénéficier de ces services. Tout le travail pénible et délicat consistant à répartir la tâche entre plusieurs threads, terminer correctement ces derniers, autant que celui consistant à créer le nombre de threads adapté au nombre de cœurs de la machine hôte, tout cela est pris en charge par TPL !

Tasks et Future

TPL introduit aussi la notion de Task et Future. Une Task (tâche) peut être vue comme un thread, mais en beaucoup plus léger (sans forcément créer de threads au niveau OS). Les Tasks sont empilées et gérées par le Task Manager et sont exécutées quand leur tour arrive dans un thread pool.

Les "Future" sont des Task qui retournent des données. Le principe reste le même sauf qu'un mécanisme permet d'être averti que le travail est terminé et que les données sont prêtes.

Les méthodes

On trouve par exemple Parallel.Invoke qui exécute tout delegate de type Action de façon parallèle sans avoir à se soucier du "comment".

Evoqués plus haut, on trouve aussi Parallel.For et Parallel.ForEach qui sont les équivalents parallèles des boucles For et Foreach de C#. On peut utiliser ces boucles partout dans son code où il y avait des For et Foreach "normaux" (en prenant garde aux effets de bord liés à l'asynchronisme !).

Le blog de l'équipe Parallel Programming

Si le sujet vous intéresse (et il doit vous intéresser ! – dans le sens du *devoir* professionnel) je vous recommande le blog de l'équipe qui a créée les extensions parallèles, c'est un point fort chez MS aujourd'hui de permettre aux équipes de communiquer directement avec les développeurs, il faut en profiter. C'est ici : http://blogs.msdn.com/b/pfxteam/

Les Reactive Extensions (Rx)

Quelle est cette nouvelle librairie et comment se positionne-t-elle par rapport à ce que nous venons de voir ?

Les Rx n'introduisent pas vraiment de nouveautés quant aux mécanismes parallèles qui, eux, sont implémentés dans le Framerwork 4.0 (TPL et P-LINQ). En revanche, les Rx viennent jouer le rôle de chef d'orchestre, d'unificateurs pour simplifier l'accès à toutes les méthodes de programmation asynchrone proposées.

Les Rx sont encore dans leur labo, disponibles mais pas encore intégrées au Framework (il faut donc les télécharger à part). Toutefois elles se basent sur deux interfaces, IObservable<T> et IObserver<T> qui elles sont déjà définies dans .NET 4.0.

Le fonctionnement de ces interfaces est simple : IObservable<T> est une sorte de IEnumerable<T>. La différence essentielle est qu'on peut énumérer tout le contenu d'une liste IENumerable en se déplaçant dedans, alors qu'avec IObservable<T> on s'enregistre auprès de la liste pour être prévenu qu'un nouvel élément est disponible. De l'asynchrone pur et une mise en œuvre de la design pattern "Producteur-Consommateur" .

A ce titre vous pouvez lire dans le C# Programming Guide de MSDN l'article suivant "How to: Synchronize a Producer and a Consumer Thread (C# Programming Guide)".

Des exemples

La programmation synchrone est un exercice de style qui réclame de se "mettre dans le mood", et les exemples, même simples, deviennent forcément plus compliqués que le "Hello World" traditionnel…

Je vous reparlerais des Rx, de TPL et de P-LINQ dans les semaines et mois à venir car ces extensions qui sont supportées par WPF mais aussi par Silverlight permettent d'améliorer sensiblement la réactivité des applications.

Pour l'instant je vais éviter les redites, et puisqu'il existe sur la question une série de billets très bien faits (et en français) écrits par Flavien Charlon, je ne peux que vous en conseiller la lecture :

- Reactive Extensions Partie 1 : Introduction
- Reactive Extensions Partie 2 : IScheduler
- Reactive Extensions Partie 3 : Implémenter un IScheduler

De même, toujours en français, un article de Thomas Jaskula décrit comment gérer un drag'n drop asynchrone avec les RX : Commencer à jouer avec Reactive Extensions (Rx) – Drag & Drop.
Sans oublier le DevLab des Rx chez Microsoft : DevLabs: Reactive Extensions for .NET (Rx)

Conclusion

Je devrais plutôt dire "Introduction" ! Car il ne s'agit pas ici de conclure sur le sujet mais bien de l'introduire et de vous inciter à aller de l'avant en suivant les liens proposés et en pratiquant vous mêmes des tests de ces nouvelles technologies.

N'oubliez pas qu'il s'agit d'une modification *radicale et incontournable* de la programmation moderne, un passage obligatoire pour faire en sorte que vos applications puissent tourner sur les machines vendues aujourd'hui et demain. ***Ne négligez pas aujourd'hui le parallélisme si vous ne voulez pas que demain vos clients ne négligent vos applications !***

Rx Extensions, TPL et Async CTP : L'asynchronisme arrive en parallèle !

Les RX Extensions, TPL et Async CTP sont trois technologies releasées ou en cours de l'être, toutes les trois traitent d'asynchronisme et de parallélisme. Toutes les trois déboulent presque en même temps, ce qui est une belle illustration d'auto-référence ! Mais en dehors de ça, comment comprendre cette avalanche et que choisir ?

Ils sont fous ces Microsoftiens !

Un peu comme les Romains d'Astérix et Obélix se tapent la tête contre les murs essayant vainement de comprendre quelle logique anime ces satanés Gaulois, le développeur s'arrache un peu les cheveux devant un tel tir groupé de trois technologies concurrentes chez le même éditeur...

Pourquoi trois procédés proches, quelles sont les différences entre eux, lequel choisir ?

Pour les lecteurs pressés

Pour ceux qui veulent tout savoir sans rien lire, on pourrait faire une version courte qui serait :

- Les Reactive Extensions (Rx) sont des opérateurs pour travailler sur des flux de données
- TPL (Task Parallel Library) est une sorte de ThreadPool sous stéroïdes
- L'Async CTP c'est TPL encore plus dopé.

Ce n'est pas forcément avec ça que vous en saurez beaucoup plus, mais quand on est très pressé, forcément, on se contente de bribes d'informations...

Pour ceux qui veulent mieux comprendre, heureusement je vais écrire une suite 

La concurrence sous .NET

La concurrence désigne un état dans lequel plusieurs code différents tournent en même temps, généralement au sein d'un même processus (au sens large) et qui accèdent à des ressources communes (d'où la concurrence, sinon il s'agit de simple multitâche). Par exemple, les applications qui tournent en même en temps sous Windows sont du code concurrent vis à vis de l'OS ou du processeur, même si chacune ne gère pas du tout de multitâche. L'OS doit gérer le fait que les applications vont accéder aux mêmes ressources comme l'écran, les imprimantes...

Sous .NET, et au sein d'une même application, on parlera de concurrence dans plusieurs cas précis :

- Une tâche est créée et démarrée en utilisant la classe Thread
- Une ou plusieurs tâches sont crées et démarrées en utilisant la classe ThreadPool
- Des tâches asynchrones sont exécutées comme une invocation de délégué par BeginInvoke, des opérations d'E/S au travers du ThreadPool, etc.
- Des tâches sont asynchrones par nature dans un environnement donné comme par exemple les Ria Services ou toute communication Wcf avec Silverlight, ou bien le choix est fait d'un tel asynchronisme dans des environnements comme WPF.
- L'asynchronisme et le parallélisme "naturel" de certains environnements comme IIS vis à vis de pages ASP.NET par exemple.

Bref dans tous ces cas il y exécution concurrente de code (mais pas forcément concurrence vis à vis des ressources communes).

Mais on pourrait étendre cette liste à bien d'autres situations, ce n'est qu'un aperçu de ce qui créée de la concurrence usuellement.

Asynchronisme et parallélisme

Blanc bonnet et bonnet blanc ? Pas tout à fait. Mais dans les faits cela reviendra à peu près au même du point de vue du développeur : des portions différentes de code vont tourner en même temps et il faudra le gérer.

Dans certains cas cela n'a pas d'importance comme une page ASP.NET accédée par cent utilisateurs en même temps. C'est IIS, .NET et l'OS qui gère la concurrence bien que dans certaines circonstances le développeur ait à prendre en compte la concurrence vis à vis des ressources si la page accède à une base de données ou un librairie ou ressource commune qui n'est pas thread safe.

Dans Silverlight, l'asynchronisme se manifeste dès qu'on appelle une fonction de communication, par un exemple un Web service ou des Ria Services. Dans un tel cas il n'y a pas vraiment concurrence, le code du thread principal continue à tourner et un appel distant est effectué, un autre code va tourner en même temps mais ailleurs, sur le serveur contacté, ce qui n'a aucun impact sur le code SL. En revanche la réponse arrivera n'importe quand et depuis le thread de communication. Si la réponse implique de manipuler l'UI il faudra s'assurer, via un Dispatcher en général, que cette manipulation s'opère bien sur le thread de l'UI et non celui de la communication.

La concurrence d'exécution ce sont plusieurs morceaux de codes qui tournent en même temps, la concurrence vis à vis des ressources ("race condition") c'est quand ces morceaux de codes accèdent à des ressources communes, et l'asynchronisme c'est plutôt la pochette surprise, des évènements qui arrivent n'importe quand.

Quand on parle de parallélisme il y a bien entendu de l'asynchronisme le plus souvent, mais on souligne plutôt ici le caractère simultané du déroulement de plusieurs tâches ou bien le découpage d'une même tâche en plusieurs morceaux exécutés en même temps dans le but précis d'accélérer l'exécution ou de la rendre plus fluide.

L'asynchronisme est présent depuis toujours ou presque (le simple fait de demander à une imprimante si elle est prête et d'attendre la réponse sans bloquer l'interface utilisateur par exemple) alors que le parallélisme n'avait cours que dans les super ordinateurs. Ce n'est que depuis que les machines sont dotées de plus d'un cœur qu'on peut réellement parler de parallélisme sur un PC. Découper une tâche en plusieurs threads dans l'espoir que cela aille plus vite n'a pas de sens sur un processeur mono cœur alors qu'avec un multi-cœur cette stratégie sera terriblement payante.

Jusqu'à lors, le parallélisme n'existait donc pas, ou sous une forme "atténuée", le multi-tâche qui longtemps fut juste simulé : le processeur passant rapidement d'une tâche à l'autre plusieurs fois par seconde donnant l'impression de la simultanéité alors qu'en réalité il n'exécute qu'une seule tâche à la fois.

Avec les machines multi-cœurs l'affaire devient toute autre car le multi-tâche devient parallèle, ce qui fait aussi apparaitre de l'asynchronisme.

Tout cela peut rapidement devenir complexe. On sait que la gestion du multi-tâche est de longue date réputée réservée aux "pointures" qui sont capables de manipuler les subtilités de ce mécanisme d'horlogerie sans se mélanger les pinceaux.

L'intérêt des librairies évoquées ici est de rendre plus simple l'implémentation de code sachant gérer le parallélisme et l'asynchronisme mais chacune avec une orientation différente :

- Ainsi Task Parallel Library (TPL) est un plutôt une API moderne autour du ThreadPool qui permet de raisonner en termes de tâches plutôt que de threads. Le niveau d'abstraction atteint libère le développeur de certains détails pénibles et lui permet de mieux se concentrer sur ce qu'il cherche à faire (au lieu de comment le faire).
- De son côté Asyn CTP est un peu comme ce qu'est Linq au traitement des données : une nouvelle syntaxe qui s'ajoute au langage pour rendre ici le traitement des tâches asynchrones

plus simples et plus naturelles. Selon toute évidence Async CPT deviendra un élément de C# aussi indispensable que l'est devenu Linq.

- Les Reactives extensions (Rx) ciblent autre chose. Ce sont plutôt des opérateurs de type Linq qui permettent de traiter des flux de données selon un mode consommateur. Tout devient flux de données, comme les évènements, ce qui permet d'écrire par exemple des opérations complètes comme un drag'n drop sous la forme d'une sorte de requête Linq. Un peu déroutant mais puissant.

Task Parallel Library (TPL)

La TPL a été releasée avec .NET 4.0 avec deux autres technologies qui tournent autour des mêmes problématiques :

- Des structures de données améliorées pour la coordination (la très méconnue Barrier ou la ConcurrentQueue<T>)
- Parallel Linq (PLINQ) qui est une extension de Linq construite sur la TPL offrant ainsi une syntaxe fluide autour de la gestion des flux de données.
- TPL introduit un découplage plus fort entre ce qu'on veut faire (une tâche) et comment l'API le gère (un thread). Ainsi TPL nous offres des tâches pour raisonner et concevoir notre code en fonction de ce qu'il doit réaliser. On retrouve de fait des concepts de haut niveau plus simples à manipuler que la gestion du multithreading .NET usuel :

- Des unités de travail ayant un cycle de vie bien défini (Created, Running…)
- Des moyens simples d'attendre qu'une Task soit terminée ou qu'un groupe de Task le soit
- Un moyen simple d'exprimer les dépendances mères/filles entre les tâches
- L'annulation qui d'une option plus ou moins simple à mettre en œuvre devient un concept clé
- La possibilité de construire des workflow conditionnels (la tâche 1 continue avec la tache 2 si la tâche 1 s'est bien terminée ou si elle a été annulée, etc.)
- TPL permet aussi de planifier des tâches avec le TaskScheduler qui existe en plusieurs implémentations. La plus commune étant le ThreadPool avec ThreadPoolScheduler qui offre un ThreadPool fonctionnant avec des Tasks en optimisant ce traitement. Il est possible en partant de la classe abstraite TaskScheduler de créer son propre planificateur de tâches.

TPL est capable de gérer intelligemment le parallélisme en utilisant habilement les capacités de la machine hôte. Par exemple si TPL détecte que le CPU peut accepter une tâche de plus, une nouvelle sous-tâche est créée et exécutée automatiquement, si le CPU est déjà trop chargé, la sous-tâche est placée dans la file du thread en cours.

Ces capacités sont utilisées par PLINQ qui assure qu'une requête parallélisée le sera toujours au mieux de ce que peut supporter le CPU au moment de son exécution. C'est un progrès énorme si on pense au code qu'il faut écrire pour atteindre une telle souplesse.

TPL offre aussi la TaskFactory<T> avec des méthodes comme FromAsync() qui permettent de faire le lien entre l'ancienne modèle de programmation asynchrone et le nouveau monde des tâches.

Il devient ainsi possible d'écrire un code comme celui-ci basé sur un appel asynchrone HTTP GET :

```
   1: Task parentTask = new Task(
   2:    () =>
   3:    {
   4:      WebRequest webRequest =
WebRequest.Create("http://www.microsoft.com");
   5:
   6:      Task<WebResponse> task =
   7:        Task<WebResponse>.Factory.FromAsync(
   8:          webRequest.BeginGetResponse, webRequest.EndGetResponse,
   9:          TaskCreationOptions.AttachedToParent);
  10:
  11:      task.ContinueWith(
  12:        tr =>
  13:        {
  14:          using (Stream stream = tr.Result.GetResponseStream())
  15:          {
  16:            using (StreamReader reader = new StreamReader(stream))
  17:            {
  18:              string content = reader.ReadToEnd();
  19:              Console.WriteLine(content);
  20:              reader.Close();
  21:            }
  22:            stream.Close();
  23:          }
  24:        }, TaskContinuationOptions.AttachedToParent);
  25:    });
  26:
  27: parentTask.RunSynchronously();
  28: Console.WriteLine("Done");
  29: Console.ReadLine();
```

Une tâche “parent” est créée décrivant à la fois la requête asynchrone HTTP ainsi qu’une tâche fille, liée à la première, se chargeant de gérer la réponse.

Le code effectue un parentTask.RunSynchronously(), ce qui signifie que les deux tâches asynchrones vont être ainsi contrôlée et contraintes dans un appel qui lui est bloquant, simplifiant énormément l’écriture du code. L’écriture de “Done” à la console est placée à la ligne suivante et ne sera exécuté que lorsque que l’ensemble de la tâche et ses sous-tâches seront terminées.

On retrouve ici un peu l’esprit des coroutines exploitées par la gestion des Workflows de Jounce mais techniquement cela est très différent (Le Workflow Jounce garantit l’exécution séquentielle de plusieurs tâches mais qui n’est pas bloquant au niveau de l’exécution du Workflow lui-même, TPL offrant ainsi plus de confort).

Il ne s’agit pas dans ce billet de faire un cours détaillé de TPL mais d’expliquer les nuances entre les trois technologies présentées en introduction. Je pense que vous avez compris ce que TPL fait, ce qu’il offre comme avantages principaux. J’y reviendrai certainement plus en détail dans de prochains billets (surtout Parallel LINQ).

Async CTP

Async CTP est une nouvelle technologie qui appartient en réalité à C# 5, elle n’existe donc qu’en l’état de bêta pour les tests. Pas de production avec Async CTP pour le moment, mais prochainement.

TPL est une avancée intéressante mais transitoire. TPL sera certainement plus utilisée au travers de Parallel LINQ que directement car Async CTP qui sera intégré à C# 5 rendra son utilisation presque caduque.

En effet, TPL oblige à une certaine gymnastique pas toujours évidente dès qu’on souhaite synchroniser plusieurs tâches qui s’enchainent. L’écriture du code devient fastidieuse car trop mécanique tout en réclamant un bon niveau de concentration pour ne pas faire de bêtise.

Async CTP ajoute des éléments au langage C#, ces éléments savent déclencher la génération automatique du code fastidieux, rendant le code utilisateur bien plus clair, plus fluide et évitant de l’encombrer de portions très mécaniques mais essentielles.

Par exemple il devient possible d’écrire une méthode qui retourne une Task ou Task<T> plutôt qu’un résultat standard. L’appelant de cette méthode peut alors attendre l’exécution de la tâche asynchrone plutôt que de recevoir un résultat immédiat. Une fonction peut aussi

présenter de “faux” multiples points de retour (“await”) que le compilateur va restructurer en une série de callbacks de façon totalement transparente pour le développeur.

Un code utilisant Async CTP et réalisant la même chose que le code précédent ressemblerait à cela :

```
 1: Func<string, Task> task = async (url) =>
 2: {
 3:   WebRequest request = WebRequest.Create(url);
 4:   Task<WebResponse> responseTask = request.GetResponseAsync();
 5:   await responseTask;
 6:
 7:   Stream responseStream = responseTask.Result.GetResponseStream();
 8:   Stream consoleStream = Console.OpenStandardOutput();
 9:
10:   byte[] buffer = new byte[24];
11:
12:   Task<int> readTask = responseStream.ReadAsync(buffer, 0,
buffer.Length);
13:   await readTask;
14:
15:   while (readTask.Result > 0)
16:   {
17:     await consoleStream.WriteAsync(buffer, 0, readTask.Result);
18:     readTask = responseStream.ReadAsync(buffer, 0, buffer.Length);
19:     await readTask;
20:   }
21:   responseStream.Close();
22: };
23:
24: task("http://www.microsoft.com").Wait();
25:
26: Console.WriteLine("Done");
27: Console.ReadLine();
```

Ici tout est asynchrone, la lecture et l’écriture du résultat, mais le code est court, lisible, l’intention est plus flagrante :

- Une requête HTTP GET est créée
- On attend la réponse asynchrone
- On obtient le flux réponse qui est écrit de façon asynchrone à la console

- On boucle de façon entre la lecture et l'écriture par paquet de 24 octets (la taille du buffer déclaré)

Aync CTP c'est cela : une amélioration très nette de C# afin de prendre en compte l'asynchronisme de façon naturelle.

Il s'agit d'une étape aussi essentielle que l'ajout de LINQ qui intégrait au langage la gestion des données.

L'informatique moderne gère essentiellement des données et doit aujourd'hui prendre en compte le parallélisme. C# s'inscrit au fil du temps dans une modernité constante, intégrant naturellement des éléments qui dépassent de loin les traditionnels "if then else" des langages classiques qui laissent au développeur toute la responsabilité de trier ou filtrer des données et d'orchestrer manuellement le ballet fragile du multitâche.

Ici aussi le but n'est pas de faire un cours sur Async CTP mais juste de vous faire comprendre à quoi cela peut servir et dans quel contexte. J'y reviendrai forcément, c'est une partie importante des nouveautés de C# 5.

Les Rx

Avec Async CTP, TPL et Parallel LINQ, on se demande quelle place peut bien rester vide pour qu'une autre librairie puisse venir s'y loger...

Asynchronisme, parallélisme, multitâche, traitement des données, tout cela peut se tisser en une trame si complexe et si différente d'une application à l'autre qu'il existe encore beaucoup de place pour autre chose. Les Reactive extensions.

Les Rx ne se concentrent pas forcément sur le parallélisme ou l'asynchronisme mais plutôt sur la façon de gérer simplement des séquences de valeurs qui sont générées dans le temps, peu importe les délais entre les moments où ces valeurs sont créées.

Bien entendu derrière tout cela on entend bien le son de l'asynchronisme comme on entend celui des timbales scander le rythme derrière un orchestre symphonique. Les Rx suppose une gestion fine de l'asynchronisme, transparente. Tellement transparente qu'elle n'est plus l'objectif premier, la gestion de l'asynchronisme n'est plus que l'assise permettant la gestion de flux de données.

Les Rx sont bâties sur toutes les notions que nous avons vues plus haut. Mais ce ne sont que des briques de construction. La finalité des Rx n'est pas de gérer directement du parallélisme

ou de l'asynchronisme. Elles permettent juste de les prendre en compte de façon transparente pour accomplir quelque chose de plus sophistiqué.

La composition de séquences de valeurs est a entendre au sens le plus large avec les Rx, des séquences de prix d'articles sont tout aussi bien utilisables que des séquences de nouveaux items arrivant dans une collection ou même qu'une séquence d'évènements souris ou clavier...

Les Rx proposent un ensemble d'opérateurs qu'on peut voir comme une sorte de DSL pour traiter des séquences de valeurs.

(un DSL est un Domain Specific Language, un langage spécifiquement adapté à un type de tâche bien précis, à la différence d'un langage classique se voulant générique).
Les Rx permettent ainsi des opérations de type :

- Création ou génération de séquences
- Combinaison de séquences
- Requêtage, projection et filtrage de séquences
- Groupage et tris de séquences
- Altération de la nature temporelle des séquences en y intégrant des attentes, des délais, des bufferisations...

Le cœur des Rx est IObservable<T>, une collection bien particulière qui permet de gérer les séquences de valeurs.

Partant de cette collection particulière et avec l'ajout d'opérateurs Linq spéciaux, il est possible de créer des requêtes de type Linq jouant non plus sur des listes pré-existantes de valeurs mais sur des flots asynchrones de données qui n'existent pas encore.

Imaginons trois méthodes dont l'exécution est assez longue et pouvant même dépendre de données totalement asynchrones (des clics souris, des données en mode push...) :

```
1: int TaskA()
2: {
3:     Thread.Sleep(200);
4:     return 42;
5: }
6:
7: string TaskB()
8: {
```

```
 9:     Thread.Sleep(500);
10:     return "La réponse est {0} ! {1}";
11: }
12:
13: string TaskC(){ return "Incroyable !";}
```

le but du jeu est d'exécuter ces trois méthodes indépendamment, de collecter les résultats et d'en produire une information finale qui sera écrite à l'écran quels que soient les délais d'attente qui peuvent fort bien être très différents de l'ordre dans lequel il faut obtenir les résultats pour accomplir le travail.

(l'exemple utilise des Thread.Sleep() pour simplifier mais ce n'est pas à reproduire, ne prenez pas cela au pied de la lettre)

Avec du code .NET classique cela donnerait ça :

```
 1: var waitHandles = new List<WaitHandle>();
 2: int ARet = 0;
 3: Func<int> A = TaskA;
 4: var ARes = A.BeginInvoke(res => { ARet = A.EndInvoke(res); },null);
 5: waitHandles.Add(ARes.AsyncWaitHandle);
 6: string BRet = "";
 7: Func<string> B = TaskB;
 8: var BRes = B.BeginInvoke(res => { BRet = B.EndInvoke(res); }, null);
 9: waitHandles.Add(BRes.AsyncWaitHandle);
10: string CRet = "";
11: Func<string> C = TaskC;
12: var CRes = C.BeginInvoke(res => { CRet = C.EndInvoke(res); }, null);
13: waitHandles.Add(CRes.AsyncWaitHandle);
14: WaitHandle.WaitAll(waitHandles.ToArray());
15: Console.Out.WriteLine(ARet, BRet, CRet);
```

Les méthodes sont exécutées dans un ordre précis, avec une attente de chaque résultat avant de passer à l'exécution de la méthode suivante.

C'est assez indigeste, pas vraiment agile, l'intention initiale est noyée dans la technique pour exécuter la tâche au lieu que cette dernière soit clairement identifiable.

Avec les Rx on peut écrire le même code de la façon suivante :

```
1: Observable.Join(
2:     Observable.ToAsync<int>(TaskA)()
3:         .And(Observable.ToAsync<string>(TaskB)())
4:         .And(Observable.ToAsync<string>(TaskC)())
5:         .Then((a, b, b) =>
6:             new { A = a, B = b, C = c })
7:     ).Subscribe(
8:         o => Console.WriteLine(o.A, o.B, o.C),
9:         e => Console.WriteLine("Exception: {0}", e));
```

Mais on pourrait vouloir exécuter tout cela de façon réellement asynchrone (exécution parallèle des méthodes). Avec du code classique cela serait très pénible. Avec les Rx il suffit d'écrire :

```
1: (from a in Observable.ToAsync<int>(TaskA)()
2:  from b in Observable.ToAsync<string>(TaskB)()
3:  from c in Observable.ToAsync<string>(TaskC)()
4:  select new { A = a, B = b, C = c })
5:  .Subscribe(o => Console.WriteLine(o.A, o.B, o.C));
```

C'est encore plus court !

Les Rx proposent ainsi une nouvelle façon de penser le traitement d'évènements asynchrones en les séquentialisant au sein d'une syntaxe logique, claire, déclarative, qui plus est réexploitant la puissance de LINQ.

Les Rx ont ainsi toute leur place à côté de Async CTP et de la TPL.

Conclusion

Il n'était pas question ici de faire un cours complet sur chaque des technologies présentées mais plutôt de vous aider à comprendre à quoi elles correspondent, qu'en attendre et vous faire découvrir ce nouveau monde parallèle et asynchrone. A la clé, vous donnez envie d'en savoir plus et de tester par vous-mêmes !

TPL avec Parallel LINQ est déjà intégré au Framework .NET 4.0. C'est dans la boite, vous pouvez vous en servir dès maintenant.

Async CTP est un CTP, un simple preview d'une technologie qui fait partie de la prochaine version 5 de C#. Vous pouvez installer la bêta de test depuis la galerie Visual Studio (Microsoft Visual Studio Async CTP).
Quant aux Rx elles existent en version 1.0 stable et peuvent être téléchargées sur le Data Developer Center de MS (Reactive Extensions).
Trois technologies qui semblent similaires mais qui, vous le voyez maintenant, offrent un angle de pénétration dans le monde de l'asynchronisme totalement différent. Chacune a son intérêt, sa place. Il faut juste s'y former, comprendre la nouvelle façon de concevoir le code.

Une autre histoire !

Je reviendrai sur ces technologies essentielles dans de prochains billets. Mais que cela ne vous empêche pas de vous y intéresser par vous même !

Programmation asynchrone : warnings à connaître…

La programmation asynchrone était déjà entrée depuis longtemps dans la panoplie du développeur, même si certains sont arrivés à faire l'autruche jusqu'à maintenant. Mais avec Windows 8 l'asynchrone est une obligation. Certains warnings du compilateurs matérialisent des erreurs fréquentes. Regardons les deux plus fréquents…

Code CS1998

Rien à voir avec l'année 98. Cela laisse juste présager qu'il y a1997 autres erreurs possibles et ça donne le vertige (mais la prochaine fera encore plus peur !).

Le code CS1998 affiche le message suivant (je travaille avec des versions US, je ne connais pas la traduction exacte en français) :

warning CS1998: This async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a background thread.

La cause est assez simple : vous avez spécifié le modificateur "async" dans l'entête de la méthode mais vous ne faites aucun usage de "await" dans cette dernière.

C'est un avertissement à prendre au sérieux. Soit vous avez placé cet "async" pour rien, et il faut le retirer… soit vous aviez prévu d'utiliser "await" et vous avez oublié de le faire ce qui risque de fausser "légèrement" le fonctionnement de la méthode. Situation à corriger immédiatement donc.

Code CS4014

Le vertige devient digne de celui d'un Baumgartner avant de sauter… imaginez 4012(*) autres warnings à découvrir ! :-)
(*) Les plus futés auront noté l'erreur… j'aurai du dire 4013 n'est-ce pas ? Ceux qui suivent vraiment auront compris qu'il n'y a pas d'erreur, puisque je vous ai déjà fait découvrir la 1998, il en reste bien 4012 à connaître !
Le message de celui-ci est tout aussi important à reconnaître :

warning CS4014: Because this call is not awaited, execution of the current method continues before the call is completed. Consider applying the 'await' operator to the result of the call.

La cause : Vous appelez une méthode qui possède le modificateur "async" et qui retourne une Task<> et votre code ne fait rien pour attendre le résultat de cette dernière.

Peut-être est-ce volontaire. Le cas peut se produire (si le code qui suit l'appel ne se sert pas du résultat par exemple).

Si ce n'est pas intentionnel il est urgent de vérifier le code appelant et le code appelé !

Si la situation est voulue, on peut supprimer le warning avec un #pragma. Cela est tout à fait acceptable car l'intention du développeur devient visible : il assume la situation. C'est mille fois préférable à un warning laissé en l'état.

Une autre astuce consiste à tout simplement déclarer un variable qui ne sert à rien et lui affecter le résultat de l'appel, de type "var dummy = appelDeCodeAsync;"

Ne vous inquiétez pas pour cette variable supplémentaire, le compilateur est assez malin pour la supprimer totalement du code compilé. Son avantage est d'éviter un #pragma (qui exige de spécifier deux fois le code du warning avec possibilité de se tromper). Comme cette solution ne coûte rien dans le code compilé elle est intéressante. Bien entendu elle sous-entend une parfaite compréhension du warning en question et ne doit pas servir à masquer une situation non maîtrisée !

Je préfère le #pragma qui marque l'intention plus clairement. Mais le risque de supprimer le warning dans tout le code (en cas d'erreur sur le second code) et de masquer des erreurs potentielles me chagrine tout autant.

A vous de voir…

Conclusion

La programmation asynchrone est pleine de surprise. Certains développeurs laissent parfois des tartines de warnings dans leurs projets. Pourtant il faut les traiter avec la même vigilance

que des erreurs de compilation car derrière chaque warning peut se cacher de potentiels bugs très sournois.

Le cas des warnings 1998 et 4014 est intéressant à ce titre.

Les warnings sont des erreurs comme les autres, mais de nature plus vicieuse, c'est un peu la leçon à retenir de ce billet…

Avertissements

L'ensemble des textes proposés ici sont issus du blog « Dot.Blog » écrit par Olivier Dahan et produit par la société E-Naxos.

Les billets ont été collectés fin septembre 2013 pour les regrouper par thème et les transformer en document PDF cela pour en rendre ainsi l'accès plus facile.

Les textes originaux ont été écrits entre 2007 et 2013, six longues années de présence de Dot.Blog sur le Web, lui-même suivant ses illustres prédécesseurs comme le Delphi Stargate qui était dédié au langage Delphi dans les années 90.

Ce reccueil peut parfois poser le problème de parler au futur de choses qui appartiennent au passé… Mais l'exactitude technique et l'à propos des informations véhiculées par tous ces billets n'a pas de temps, tant que C# existera…

Le lecteur excusera ces anachronismes de surface et prendra plaisir j'en suis certain à se concentrer sur le fond.

E-Naxos

E-Naxos est au départ une société éditrice de logiciels fondée par Olivier Dahan en 2001. Héritière de Object Based System et de E.D.I.G. créées plus tôt (1984 pour cette dernière) elle s'est d'abord consacrée à l'édition de logiciels tels que la suite Hippocrate (gestion de cabinet médical et de cabinet de radiologie) puis d'autres produits comme par exemple MK Query Builder (requêteur visuel SQL).

Peu de temps après sa création E-Naxos s'est orientée vers le Conseil et l'Audit puis s'est ouverte à la Formation et au Développement au forfait. Faisant bénéficier à ses clients de sa longue expérience dans la conception de logiciels robustes, de la relation client, de la connaissance des utilisateurs et de l'art, car finalement c'en est un, de concevoir des logiciels à la pointe mais maintenables dans le temps.

C#, Xaml ont été les piliers de cette nouvelle direction et Olivier a été récompensé par Microsoft pour son travail au sein de la communauté des développeurs WPF et Silverlight. Toutefois sa première distinction a été d'être nommé MVP C#. On ne construit pas de beaux logiciels sans bien connaître le langage…

Aujourd'hui E-Naxos continue à proposer ses services de Conseil, Audit, Formation et Développement, toutes ces activités étant centrées autour des outils et langages Microsoft, de WPF à WinRT (Windows Store) en passant par Silverlight et Windows Phone.

A l'écoute du marché et offrant toujours un conseil éclairé à ses client, E-Naxos s'est aussi spécialisée dans le développement Cross-Plateforme, notammant dans le mariage des OS Microsoft avec Android, les deux incontournables du marché d'aujourd'hui et de demain.

N'hésitez pas à faire appel à E-Naxos, la compétence et l'expérience sont des denrées rares !