C# de la version 3 à la version 7

Olivier Dahan

Collection
ALL DOT BLOG

Tome 1

Troisième Edition

e-naxos

www.e-naxos.com

Formation – Audit – Conseil – Développement
XAML, C#
Cross-plateforme Windows / Android / iOS / Mac OS
XAMARIN / Xamarin.Forms

ALL DOT.BLOG
3e édition - 2017
C#

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

Couverture - Conception & Réalisation : Oliver Dahan avec la collaboration de Valérie Pitard

Microsoft® MVP Most Valuable Professional
Olivier Dahan
odahan@gmail.com

Table des matières

Présentation

Bien qu'issu pour l'essentiel des billets et articles écrits sur Dot.Blog au fil du temps, le contenu de ce PDF a entièrement été réactualisé lors de la création du présent livre PDF en novembre 2013 puis en août 2015 et à nouveau en avril 2017. Il s'agit d'une version inédite corrigée et à jour, un énorme bonus par rapport au site Dot.Blog ! Un mois de travail a été à chaque fois consacré à la réactualisation du contenu. Corrections du texte mais aussi des graphiques, des pourcentages des parts de marché évoquées, contrôle des liens, et même ajouts plus ou moins longs, c'est une véritable édition spéciale différente des textes originaux toujours présents dans le Blog !

Toutefois les billets n'ont pas tous été réécrits totalement ils peuvent donc parfois présenter des anachronismes sans gravité. Tout ce qui est important et qui a changé de façon notable a soit été réécrit soit a fait l'objet d'une note, d'un aparté ou autre ajout.

C'est donc bien plus qu'un travail de collection déjà long des billets qui vous est proposé ici, c'est une relecture totale et une révision et une correction techniquement à jour au moins d'avril 2017. *Un vrai livre*. **Gratuit**. De surcroît il est évolutif et couvre jusqu'à C# 7 ! Quel éditeur vous offre un tel service ? !

Astuce : cliquez les titres pour aller lire sur Dot.Blog l'article original et ses commentaires ! Tous les liens Web de ce PDF sont fonctionnels, n'hésitez pas à les utiliser (même certaines images, alors essayez, au pire il ne se passe rien, au mieux vous obtiendrez l'image en 100 %) !

Avertissement : Ce livre n'est pas un cours C# pour débutant… Ici je traite de sujets particuliers, d'aspects moins bien connus du langage dans un ordre qui n'a rien d'une progression. C'est un livre pour qui connait déjà les rudiments de C#.

Lisez ce livre dans l'ordre que vous voulez, l'essentiel c'est d'y apprendre quelque chose et d'y prendre plaisir !

C# - Un tour d'horizon

Chacun d'entre nous connait les rudiments de C#, moins nombreux sont les experts qui jonglent avec toutes ses possibilités. Parmi celles-ci j'en ai relevé que vous ne connaissez peut-être pas ou que vous oublieriez d'utiliser, ou que vous ne sauriez pas utiliser sans consulter la documentation. Un peu comme une visite de nuit d'une ville fait découvrir des paysages nouveaux dans des rues qu'on connait si bien de jour, C# by night c'est ça, alors suivez le guide !

C# un langage riche

Je pense qu'aucun langage de programmation n'a jamais connu autant de modifications et d'améliorations, de changements de paradigmes que C# depuis sa création.

C# est un langage à la fois simple, défini par une poignée de mots clés, et à la fois d'une grande subtilité. Il est à la fois fortement typé tout en laissant beaucoup de liberté. Il peut être utilisé de façon simple ou on peut s'adonner à des constructions sophistiquées qui tiennent en quelques instructions pourtant (Linq par exemple).

Même s'il ne s'agit pas de troller sur le thème rabâché de la guerre des langages il faut convenir que C#, à défaut d'être "le meilleur" langage est vraisemblablement ce qui se fait de mieux en la matière.

Dans le foisonnement des langages de Php à JScript en passant par Java ou C++, C# apparait tel un soleil levant au-dessus d'un nuage de pollution.

C# est la cerise sur le gâteau, la chantilly sur le banana split. Bref un truc super chouette. Si je pense vous faire découvrir prochainement F#, C# reste et restera pour moi encore longtemps "le" langage de référence et de préférence.

Toutefois il reste difficile d'en mesurer toutes les finesses. J'ai déjà écrit sur ce thème, mais pour aujourd'hui je vous ai cuisiné un mélange de saveurs qui balayeront un peu tous les domaines du langage, des attributs à la syntaxe pure et dure, des mots-clés au méthodes et propriétés, rien de très savant à attendre (quoi que !), une sorte de révision, un devoir de vacances "cool" (vous comprendrez plus loin)…

Syntaxe

C'est un peu le B.A.BA d'un langage, ce qu'il offre vraiment. C# a été conçu après l'échec du Java Microsoft, écrit par Hejlsberg qui avait justement été débauché de

chez Borland, où il avait créé Delphi et son Pascal Objet. On sait les problèmes que Microsoft a rencontrés avec son Java, à mon sens fort injustement, ce qui a obligé à battre en retraite. Il fallait une nouvelle idée et sur les bases syntaxiques de Java et les restes de Delphi C# est né. Hejlsberg a certainement aussi été influencé par C++ (la surcharge des opérateurs par exemple) et par des langages fonctionnels puisque des extensions comme Linq ont fini de faire de C# un langage si unique. Historiquement le langage s'appelait au départ (vers 2000) "C-like Object Oriented Language", noté "Cool", le nom C# est venu plus tard et le # semble être une incrémentation de "++" répété deux fois graphiquement indiquant clairement l'ambition du langage… C# est le dernier grand langage "all purpose" (tout usage) créé et il n'a pas été égalé ni dépassé depuis sa sortie il y a 15 ans déjà (2002) !

Alors puisque la syntaxe est à la base du langage, regardons de plus près certains éléments de cette syntaxe qui ne sont pas forcément les plus connus ou les plus utilisés. Forcément ce n'est pas exhaustif, chaque billet de Dot.Blog ne pouvant être un livre !

Opérateur ??

Cet opérateur est appelé en anglais "*null-coalescing operator*" ce qu'on pourrait traduire par "opérateur d'union pour les valeurs nulles". Traduction pas fantastique je dois bien l'avouer.

Il est plus facile d'en comprendre le fonctionnement mais étrangement il reste trop peu utilisé.

Son rôle est de faire un OR logique entre deux valeurs possibles, choisissant la seconde systématiquement si la première est nulle. Cet opérateur est très utile quand on veut s'assurer d'obtenir une valeur non nulle sur une affectation sans avoir à écrire un test préliminaire de la valeur. l'opérateur a été introduit principalement pour simplifier l'utilisation des types "nullable", comme dans utilisé dans l'exemple ci-dessous.

```
int? x = null;
int y = x ?? -1;
// ici y vaut -1 et nul besoin de le déclarer nullable...
```

Facile donc. Il suffit d'y penser car l'opérateur s'avère fort utile dans beaucoup d'autres situations.

Marquage des nombres

Peu de code utilise des nombres correctement marqués (sauf quand le compilateur rouspète). Cela est pourtant souvent important même si les conversions implicites entre les types arrangent souvent les choses.

Ainsi on connait généralement

- M, m pour le type decimal, exemple: `var t=30.2m;`
- F, f pour le type float, exemple: `var f=30.2f;`
- D, d pour le type double, exemple: `var d=30.2D;`

C'est volontairement que j'ai utilisé la même constante dans les exemples et le mot clé var pour définir les champs. Lorsqu'on utilise ce dernier il est très important de bien marquer les nombres car sinon le compilateur ne peut le deviner et choisi un type standard (double) qui n'est pas forcément adapté au code. C'est le cas aussi dans le typage implicite des paramètres d'une méthode générique (voir plus loin dans ce billet).

Mais ces marqueurs sont assez connus. Il en existe de plus exotiques :

- U,u pour Unsigned Int
- UL, ul, Ul, uL … pour Unsigned Long
- L, l pour Long

On notera que les minuscules et les majuscules sont acceptées, même dans UL où toutes les combinaisons sont légales.

Pour les Long l'utilisation du L minuscule ("l") est déconseillée car ce caractère est dans la majorité des fontes source de confusion avec le chiffre un "1". Mais l'usage de la minuscule est légale à défaut d'être lisible.

Bien entendu, au-delà des marqueurs eux-mêmes mon intention est d'attirer votre attention sur la richesse des types disponibles. En dehors des `int` et des `double` (souvent par défaut) je ne vois pas souvent les autres types utilisés autant qu'ils le devraient. Bien choisir les types numériques a pourtant une influence assez grande sur l'occupation mémoire et les temps de calculs sans parler de la précision (utiliser un double au lieu d'un decimallorsqu'on traite des sommes d'argent est par exemple une erreur qui peut avoir des conséquences importantes).

Le double double point ::

Le "double double-point" (plus loin je noterai DDP) est tellement utilisé rarement que je ne l'ai jamais vu lors de mes audits. Certains doivent bien s'en servir mais peu. Et forcément lorsqu'on utilise peu voire jamais un élément du langage on oublie même qu'il existe le jour où il pourrait être utile !

A quoi sert le DDP ? Il s'utilise en réalité avec une autre possibilité du langage : la définition d'alias pour les espaces de noms.

Ainsi on peut écrire :

```
using web = System.Web.UI.WebControls;
using win = System.Windows.Forms;

web::Control aWebControl = new web::Control();
var aFormControl = new win::Control();
```

Dans un tel contexte on pourrait parfaitement utiliser un point de ponctuation simple après les alias "web" et "win", cela fonctionnerait parfaitement.

Alors pourquoi le DDP ?

Il sert à forcer l'interprétation de ce qui le précède comme étant un alias d'espace de noms. Et en quoi cela est-il important dans certains contextes ? Tout simplement parce que si demain nous ajoutons à cette application un espace de noms "web" et que nous y définissons une classe "Control", l'écrire avec un seul point fera que le compilateur ne saura plus de quoi on parle, il y aura collision de noms… Choisir d'utiliser le DDP évite toute confusion possible dans l'immédiat mais aussi dans l'avenir… A utiliser systématiquement lorsqu'on met des alias de namespace donc.

Portée des accesseurs des propriétés

Un peu mieux connu mais pas très souvent utilisée, la possibilité existe de définir une portée différente pour les accesseurs d'une propriété. Par exemple l'utilisation la plus fréquente est d'avoir un getter public et un setter privé.

Bien entendu de nombreuses autres possibilités sont autorisées comme l'utilisation de `internal` ou de `protected`. Choses plus subtiles mais qui peuvent bien entendu tout changer…

Ce qui me fait penser à un joke américain que j'ai lu sur G+ dernièrement et que je vais traduire de mémoire :

"Ecrire une application c'est la même chose qu'écrire un livre. Sauf que si vous oublier un mot page 246 c'est tout le livre qui n'a plus sens et n'est plus lisible."

Génériques implicites

Ce n'est pas grand chose mais tout de même cela allège la lecture du code. Et tout ce qui rend le code plus simple à lire le rend plus facile à maintenir. C'est donc finalement plus important qu'il n'y parait !

```
void GenericMethod<T>( T input ) { ... }
```

```
GenericMethod<int>(23);    // <> n'est pas utile
GenericMethod(23);         // inférence du type par C#
```

L'inférence de type de C# rend l'indication du type inutile. On est toujours dans les petites choses comme "var" qui rend l'écriture du code C# moins lourde, presque comme un langage non typé alors qu'il reste très fortement typé. Nous verrons dans de prochains billets que ce dénuement est justement l'avantage de F# tout en restant typé. Mais restons en C# pour le moment !

On revient ici à ce que je disais sur les marqueurs de types numériques. Si vous n'utilisez pas de marqueur vous êtes dans l'obligation de spécifier le type sinon "23" sera interprété comme un *integer* et sera ensuite traité comme tel par tout le code de la méthode "GenericMethod" (exemple ci-dessus). Si cette dernière fait des calculs, comme des divisions par exemple, c'est l'arithmétique des entiers qui s'appliquera avec un effet de bord pas forcément désiré. Si dans un tel cas on souhaite s'assurer que le numérique sera traité comme un double (simple exemple) il faudra soit ajouter le type à l'appel et ne pas utiliser l'inférence, soit utiliser un marqueur ("d" ici).

Si on souhaite définir des guidelines pour une équipe de développement on s'aperçoit que les subtilités de C# obligent à spécifier des règles très précises comme toujours utiliser des marqueurs pour les constantes numériques afin d'éviter une mauvaise inférence lors d'un appel de méthode générique sans le type… Et puis toujours et encore : *écrire un bon code c'est écrire un code dont l'intention apparait évidente*. Une telle stylistique évite les commentaires (qu'on oublie de mettre à jour…) et allège la lecture. Ecrire `23d` au lieu de `23` marque de façon évidente l'intention d'utiliser un double par exemple. Pensez-y !

Le mode verbatim

Cette possibilité est à la limite d'être présentée ici, mais je me suis aperçu qu'elle n'était pas utilisée tout le temps, alors juste un rappel rapide :

```
// au lieu d'écrire:
var s = "c:\\program files\\oldway"
// il est préférable d'écrire:
var s =@"c:\program file\newway"
```

L'arobas est utilisé comme marqueur pour la chaîne de caractères définies juste derrière. Les caractères d'échappement ne sont plus interprétés ce qui simplifie l'écriture (et la lisibilité) notamment des chemins Windows… On notera qu'on peut toutefois utiliser certains caractères comme le double guillemet à condition de le doubler.

Attention, l'arobas a deux utilisations très différentes, celle dont je viens de parler et la suivante :

Arobas

Utilisé pour la définition des chaînes de caractères en mode verbatim, l'arobas a aussi une seconde signification : cela permet de définir et d'utiliser des noms de variables identiques à des mots clés du langage.

Ce genre de mélange *est plutôt à éviter*, mais parfois définir une variable "return" ou "interface" peut avoir une cohérence sémantique avec le code qui l'impose plutôt que d'utiliser d'autres noms qui auront moins de sens. L'*intention visible* étant essentielle cela peut se justifier ponctuellement. Mais dans ce cas il faudra écrire "@return" et "@interface". Il s'agit bien sûr de simples exemples, cela fonctionne pour tous les mots clés de C#. Once more, à n'utiliser que dans des cas particuliers. Un truffé d'arabas et de variables identiques aux mots clés serait illisible et non maintenable.

Valeurs spécifiques pour les énumérations

Celle-ci est facile mais comme tout le reste, quand on sait c'est enfantin mais sinon…

Il est possible de spécifier des valeurs particulières aux items d'une énumération au lieu de laisser le compilateur utiliser généralement une suite de 0 à la valeur maximale autorisée.

Fixer les valeurs soi-même est très pratique lorsqu'on souhaite avoir une énumération qui limite et encadre les choix dans l'application tout en pouvant être "mappés" sur des valeurs utilisée par une API externe à l'application. Imaginons un librairie de reconnaissance OCR dont l'une des méthodes possède un paramètre qui peut prendre les valeurs 1, 5 ou 8. De telles curiosités existent j'en ai rencontrées… Imaginons cette API et notre application qui elle est écrite proprement. Nous avons intérêt à définir une énumération de trois valeurs mais avec des noms très explicites comme "Manuscrit","Photo","Livre" qui pourraient être la signification des trois valeurs évoquées plus haut.

En faisant ainsi l'application pourra utiliser une notation claire et lisible, comme "var a = ModeReconnaissance.Photo;" ce qui est tout de même plus clair que "var a = 5;".

Au moment où l'appel à l'API un peu rustique sera effectué il suffira de caster la valeur en integer et l'affaire sera dans le sac `: ApiBizarre((int)maVarEnum);` !

Pour résumer, on peut donc écrire quelque chose comme cela :

```
public enum ModeReconnaissance
{
   Manuscrit = 1,
```

```
    Photo = 5,
    Livre = 8
}
```

On n'oublie pas au passage qu'une énumération est définie par défaut comme un type int mais qu'il est possible de forcer ce type à tout type intégral comme byte, short, ushort, int, uint, long ou ulong. Il suffit pour cela de faire suivre le nom du type par deux points ":" suivi du nom du type intégral.

Ici encore il ne s'agit pas forcément d'élément du langage qui serait exotique, c'est juste qu'on l'a parfois lu une fois ou deux mais qu'on ne l'a jamais utilisé. Du coup on ne pense pas à s'en servir quand cela serait utile pour clarifier le code.

Event == Delegate

On l'oublie parfois mais la gestion des évènements n'est qu'une façon d'utiliser des delegates… Et tout delegate peut recevoir plusieurs méthodes attachées en utilisant += (et son contraire –= pour se désabonner).

Les évènements ne sont pas réservés aux seules gestions des clics des boutons ou autres contrôles. Souvent le développeur débutant ne comprends pas qu'il peut lui aussi utiliser ce mécanisme pour ces propres classes qu'elles soient d'UI ou des POCO's agissant au niveau du DAL ou du BOL de l'application, voire ailleurs (ViewModels ou services par exemple).

C'est une façon de programmer très puissante. Sous Delphi (créé par Hejlsberg) les évènements existaient aussi mais un seul et unique abonnement était possible ce qui affaiblissait l'intérêt du mécanisme même s'il s'agissait déjà d'un grand progrès. C# a amené la souplesse à ce procédé en autorisant les abonnés multiples (pattern*Observer* dans les guides de design patterns).

Mieux on peut écrire totalement le code de son évènement pour obtenir un contrôle très fin (automatiser un processus dès lors qu'un abonnement est créé par exemple). il est ainsi possible de définir l'évènementSelectionEvent de la façon suivante :

```
public event EventHandler SelectionEvent(object sender, EventArgs args)
  {
     add
        {
       if (value.Target == null) throw new
                          Exception("No static handlers!");
      // faire d'autres choses ici pourquoi pas...
           _SelectiveEvent += value;
    }
    remove
        {
      // ici aussi il est possible d'insérer du code...
      _SelectiveEvent -= value;
          // ici aussi...
```

```
    }
  }

...
private EventHandler _SelectiveEvent;
```

Bien entendu dans de nombreux cas une telle écriture n'est pas nécessaire puisque C# nous offre des moyens plus commodes et modernes pour définir de nouveaux évènements. Mais les "anciennes pratiques" existent toujours dans le langage et elles ont l'avantage d'offrir plus de souplesse.

Utiliser les techniques les plus récentes est une bonne chose, mais cela donne l'impression qu'implicitement on est forcé d'oublier les anciennes constructions et leurs avantages. C'est une mauvaise approche du langage. Chaque construction possède ses avantages et ses inconvénients. Bien maitriser un langage pour écrire du bon code implique d'avoir le choix des constructions les mieux adaptées. Il est donc essentiel de se souvenir de toutes celles que le langage considéré offre aux développeurs, sinon il n'y a plus de choix éclairé. Les langages informatiques sont comme les autres, tombent facilement dans le dogme ou la pensée unique. Y échapper réclame un effort…

Parenthèses américaines et Chaînes de format

Un rappel hyper rapide : dans une chaîne de format les parenthèses américaines de type "curly", c'est à dire `{` et `}` servent à insérer des variables passées à la fonction de formatage (comme `String.Format` par exemple.

Si on veut utiliser ces caractères dans la chaîne de sortie il suffit de les doubler dans la chaîne de format :

`String.Format("{{coucou}} M. {0}","Albert")` donnera en sortie `"{coucou} M. Albert"`

Le français utilise peu ces caractères raison pour laquelle certainement peut de gens s'intéressent à cette possibilité… mais parfois cela peut être pratique !

Checked et Unchecked

Maitriser quand un calcul doit "planter" ou non est une attention qui se perd… On espère certainement que tout ira bien et qui si ça va mal il se passera bien quelque chose et qu'il "suffira" de déboguer. Enfin j'imagine. Car en effet l'utilisation de Checked et Unchecked est vraiment rare dans les codes que je peux auditer !

Pourtant c'est utile.

```
short x = 32767;   // rappel de la valeur max pour un short
```

```
short y = 32767;

int a1 =  checked((short)(x + y));      // OverflowException

int a2 =  unchecked((short)(x + y));    // retournera en silence -2

int a3 =  (short)(x + y);               // retournera en silence -2
```

Détecter un calcul qui génère un overflow (débordement) est souvent crucial, alors pourquoi ne voit-on pas plus souvent Checked utilisé ? Mystère !

#error et #warning

En utilisant ces deux indicateurs on peut déclencher à volonté soit des erreurs de compilations (directive #error) ou des warnings (directive #warning).

Cela peut être très utile et plus efficace qu'un TODO car dépendant directement du code et du compilateur et non de l'EDI qui va présenter plus ou moins clairement les fameux TODO. Ensuite on peut utiliser ces directives dans des constructions #if ce qui leur donne encore plus d'intérêt.

Par exemple si votre code est conçu et optimisé pour le mode x64 vous pouvez déclencher une erreur de compilation uniquement si le développeur utilise "any CPU" à la compilation ou des choses encore plus spécifiques en jouant sur des #define.

L'intérêt de #error est de bloquer la compilation avec un message personnalisé. L'intérêt de #warning est de faire apparaitre un message personnalisé sous la forme d'un avertissement seulement.

Le petit GIF suivant montre tout cela en mouvement (sous LinqPad à voir sur Dot.Blog).

Encore une fois rien de bien époustouflant, juste une fonctionnalité de C# rarement utilisée alors qu'elle peut rendre de grands services parfois…

Les attributs

La plateforme .NET définie de nombreux attributs, chaque espace de noms ou presque ayant les siens. Les attributs servent à tout. C'est autant une feature du langage qu'un style de programmation. Certains frameworks MVVM en font par exemple usage pour marquer des classes : comportement, navigation… L'Entity Framework s'en sert pour reconnaitre les clés primaires ou les identifiants (et d'autres possibilités) alors que l'EDI utilise certains attributs pour afficher le nom des propriétés d'un Control. Bref, les attributs sont une possibilité du langage très utilisée par les frameworks mais relativement peu par les développeurs eux-mêmes alors qu'il

est si facile d'en définir… Mais regardons ici quelques attributs touchant plus le langage et son interprétation.

DefaultValueAttribute

Cet attribut est surtout utile lorsqu'on développe un Control ou User Control. Il permet d'indiquer la valeur par défaut d'une propriété. Visual Studio ne mettra pas en gras la valeur si elle est égale à la valeur par défaut déclarée ce qui facilite la lecture de toutes les propriétés par l'utilisateur de la classe… Les valeurs par défaut sont ignorées à la sérialisation ce qui confère aussi un avantage sur la taille des instances sérialisées et du code XAML s'il y en a.

Curieusement l'attribut ne positionne pas de valeur. Il est juste informatif et la valeur par défaut doit tout de même être spécifiée ailleurs dans le code d'initialisation de l'instance (ou la déclaration du champ privé pour un backing field de propriété par exemple).

En utilisant la Réflexion VS sait aussi rétablir la valeur par défaut lorsqu'on demande un Reset d'une propriété dans l'EDI.

Il est possible d'utiliser la même astuce dans son propre code :

```
foreach (PropertyInfo p in this.GetType().GetProperties())
{
    foreach (Attribute attr in p.GetCustomAttributes(true))
    {
        if (attr is DefaultValueAttribute)
        {
            DefaultValueAttribute dv = (DefaultValueAttribute)attr;
            p.SetValue(this, dv.Value);
        }
    }
}
```

L'attribut lui-même se pose de la façon la plus standard qu'il soit. Imaginons une propriété Color dont la valeur par défaut est vert :

```
[DefaultValue(typeof(Color), "Green")]

public Color MaCouleur { get; set; }
```

Ne pas oublier d'ajouter dans le constructeur ou ailleurs dans le circuit d'initialisation de l'instance une affectation du type MaCouleur=Color.Green; sinon l'attribut n'aura pas l'effet escompté…

ObsoleteAttribute

Cet attribut est très pratique et peu utilisé aussi. Il ne sert pas seulement dans les frameworks à marquer les méthodes, membres ou classes qui deviennent obsolètes (et à préciser une description contenant généralement le nom de la classe, membre ou méthode à utiliser à la place), il peut aussi rendre de grands services dans une application tout à fait classique.

Par exemple supposons qu'on désire créer une méthode B qui prend en charge une opération de façon plus complète que la méthode A originale. En marquant obsolete cette dernière et en compilant on obtiendra immédiatement tous les endroits de l'application qui font usage de cette méthode de façon plus fiable qu'avec un Search… Pratique.

C'est aussi une bonne habitude de faire évoluer son code sans "tout casser", créer de nouvelles interfaces, de nouvelles API's le tout sans rendre l'ancien code totalement inutilisable. Et si c'est le cas raison de plus pour prévenir le développeur avec des messages de compilation.

L'une des causes de bogue est souvent le changement de sens, de qualité, ou de fonctionnalité pour une propriété ou une méthode. Le nouveau code marche très bien mais on a oublié que d'anciennes parties de code utilisent encore la méthode ou la propriété dans son ancienne signification. Et Boom! un jour ça casse et c'est difficile à déboguer ! Dans un tel cas il est bien préférable de créer une nouvelle propriété, une nouvelle classe ou nouvelle méthode, quitte à le faire par copier/coller et à changer juste ce qu'il faut dans la nouvelle copie. On utilisera un nom similaire avec une version par exemple Class Auto, devient Class Auto2 ou Auto2014… Bien entendu dans le même temps la classe Auto sera marquée obsolète. Tout code utilisant Auto sera assuré de fonctionner comme avant mais recevra un warning d'obsolescence, tout code utilisant Auto2 sait qu'il choisit la nouvelle version et assume les différences sans recevoir de warning.

Bref, marquer son code (classe, méthode, propriété, interface) avec l'attribut obsolete n'est pas réservé aux concepteurs de frameworks, c'est aussi une bonne pratique pour toute application !

La définition est évidente :

```
[Obsolete("Utiliser la méthode B à la place.")]

static void MethodA()
```

DebuggerDisplay

Cet attribut (défini dans System.Diagnostics) permet de contrôler comment une classe ou un champ sont affichés dans la fenêtre des variable du débogueur de VS.

L'attribut peut être utilisé avec : les classes, les structures, les delegates, les énumérations, les champs, les propriétés et les assemblages.

Cet attribut joue un peu le même rôle que la surcharge de ToString() qui est une guideline essentielle pour aider à déboguer une application. En effet, au lieu de <NomdeType>, un ToString() surchargé peut retourner les valeurs essentielles de l'instance. Parfois même cela évite d'avoir à créer des convertisseurs de valeurs ou à écrire du code spécifique dans les ViewModels ! Que du bonheur !

Mais si on doit déboguer un code dans lequel ToString() n'a pas été surchargé ou bien qui ne retourne pas les informations dont on a besoin, il est bien plus intelligent d'utiliser l'attribut DebuggerDisplay car toucher auToString() peut potentiellement casser du code existant…

Si les deux sont définis, pour la fenêtre du débogueur c'est l'attribut qui a la précédence.

Ce qu'on sait encore moins généralement sur cet attribut c'est qu'il est défini avec une chaine de caractères et que celle-ci peut contenir des accolades { } qui enchâssent un nom qui sera évalué dynamiquement. Ce nom peut être celui d'un champ, d'une propriété ou même d'une méthode !

Exemples :

```
// affichera x = 22 y = 2 par exemple et il sera utilisé sur les
// deux propriétés x et y
[DebuggerDisplay("x = {x} y = {y}")]

// plus subtile en faisant appel à une méthode
// notamment getString()
[DebuggerDisplay("String value is {getString()}")]
// sortie possible : String value is [4,2,9,12]

// autre utilisation du même type
[DebuggerDisplay("{value}", Name = "{key}")]
public class KeyValuePairs { ... public object Key {get; set;} }
```

La souplesse de cet attribut est vraiment un plus qui peut faire gagner des heures lors d'une session de débogue et sans utiliser d'outils complexes ou de code spécifique pour les tests qui eux-mêmes font parfois perdre plus de temps qu'autre chose…

DebuggerBrowsable et DebuggerStepThrough

Comme on vient de le voir certains attributs ne servent qu'à Visual Studio et son débogueur afin de simplifier le travail de test ou de débogue d'une application (ou de tout code). C'est aussi le cas des deux attributs présentés ici.

Le premier permet d'indiquer au débogueur si une propriété ou autre est visible dans la fenêtre de débogue. Cela est très pratique pour cache une propriété un peu *bidon*, intermédiaire, ou qu'on a ajouté juste pour les tests ou qui n'a pas grand sens pour la session de débogue. En évitant d'avoir une vision brouillée par des choses inutiles dans la fenêtre de débogue on s'assure un débogage plus rapide !

Le second attribut permet d'indiquer que le code qui le suit doit être sauté par le débogueur. Cela aussi est très pratique car parfois le jeu des "step through" (sauts internes) lors d'un débogue fini par devenir un jeu de piste harassant dans lequel on se perd au bout d'un moment. Faire en sorte que le débogueur saute les parties n'ayant pas d'intérêt (et sans bricoler le code !) peut s'avérer payant et faire gagner beaucoup de temps …

Exemple d'utilisation :

```
[DebuggerBrowsable(DebuggerBrowsableState.Never)]
private string nickName;
public string NickName    {
    [DebuggerStepThrough]
    get { return nickName; }
    [DebuggerStepThrough]
    set { this.nickName = value; }
}
```

ThreadStaticAttribute

Cet attribut permet d'indiquer qu'un champ statique sera unique pour chaque thread. De fait le champ a beau être statique il peut prendre plusieurs valeurs… Une par thread. Cela s'avère très intéressant pour conserver des données dans un code thread-safe comme une connexion à une base de données différente pour chaque thread.

ThreadStatic permet d'éviter l'utilisation de Lock en multithreading dans certains cas, notamment quand la valeur doit être protégée contre les accès concurrents mais qu'elle peut être différente dans chaque thread. Dans un tel cas l'attribut permet de créer autant de variables qu'il y a de threads supprimant toute collision éventuelle mais en conservant une écriture fortement typée du code et la sémantique de "static".

L'utilisation de cet attribut ayant du sens au sein de code multithreadé uniquement et ce type de code étant long à démontrer le plus souvent je laisse le lecteur faire des recherches sur Google pour trouver plus de documentation.

FlagsAttribute

Les énumérations sont souvent utilisées pour indiquer des options possibles, c'est presque leur unique utilisation. La plupart du temps il s'agit options simples : un objet peut être d'une couleur ou d'une autre, un employé peut être en activité ou non, une machine peut être dans un état ou un autre… Mais il existe des cas où les options doivent pouvoir se combiner : un objet peut avoir une couleur qui est une combinaison de celles définies, certains états d'une machine peuvent être superposables, des options de recherche par exemple peuvent être cumulables, etc.

Pour simplifier, si on doit afficher des choix avec ces énumérations visuellement les premières énumérations peuvent être affichées dans des ListBox ou des ComboBox, voire des boutons radio, alors que les secondes n'ont de sens qu'avec un ensemble de CheckBox.

Si le premier cas d'utilisation est disponible par défaut et s'il est bien compris et bien utilisé la plupart du temps, le second cas est souvent négligé. Je rencontre très peu de code qui en fait (bon) usage et c'est dommage.

Car il existe un attribut, FlagsAttribute, dont le but est justement de marquer une énumération de telle sorte à ce que ces différentes valeurs soient cumulables.

Il est vrai que cet attribut est plus informatif que fonctionnel. C'est à dire que son utilisation ne dispense pas de définir soi-même les valeurs correctes (puissances de 2) pour les constantes de chaque item de l'énumération et qu'attribut ou pas attribut si on respecte cette règle les choses marcheront de la même façon…

Toutefois FlagsAttribute confère au moins deux avantages au code : *la clarté de l'intention* (les options peuvent être combinées avec un OU) et l'accès à certaines méthodes comme `HasFlag`() dont l'effet n'est pas garanti sinon.

Alors lorsque vos énumérations représentent des options cumulables par un OU, n'oubliez pas de les définir comme des flags en veillant à ce que les options soient bien des puissances de 2.

Comme on le voit dans le code ci-dessous on peut définir dans l'énumération des valeurs qui sont déjà des combinaisons numériques de plusieurs valeurs, ce qui facilite l'utilisation de l'énumération pour les combinaisons les plus fréquentes.

```
void Main()
{
    var none = TestEnum.None;
    var a = TestEnum.Read;
    var b = TestEnum.Write;
    var c = TestEnum.ReadWrite;
    var d = TestEnum.Locked;
    Console.WriteLine("None; Valeurs : {0}; {1}; {2}; {3}; {4}",<br>
none,a,b,c,d);
    Console.WriteLine("None; (short)Valeurs : {0}; {1}; {2}; {3}; {4}",<br>
(short)none,(short)a,(short)b,(short)c,(short)d);
```

```
    var e = TestEnum.WriteLocked;
    var f = TestEnum.Write | TestEnum.Locked;
    Console.WriteLine("e= {0}; f={1}; e==f? {2}",<br>                e,f,
e==f?"OUI":"NON");
    Console.WriteLine("Has Flag 'Locked' ? {0}",<br>
e.HasFlag(TestEnum.Locked));
}

// Define other methods and classes here
[FlagsAttribute]
public enum TestEnum : short
{
    None = 0,
    Read = 1,
    Write = 2,
    Locked = 8,
    ReadWrite = 3,
    WriteLocked = 10
}
```

Ce qui produira la sortie suivante :

```
None; Valeurs : None; Read; Write; ReadWrite; Locked
None; (short)Valeurs : 0; 1; 2; 3; 8
e= WriteLocked; f=WriteLocked; e==f? OUI
Has Flag 'Locked' ? True
```

ConditionalAttribute

Ce dernier attribut que je vous présente est vraiment un truc à connaitre. En effet il permet de marquer une méthode afin que son exécution soit lié à une définition d'identificateur (#define).

Certes il est toujours possible d'écrire du code que celui-ci :

```
#if DEBUG

void ConditionalMethod()

{ ... }

#endif
```

Mais un tel code est à la fois lourd et n'est pas sans poser de petits problèmes… Notamment parce que dans le cas ici ou DEBUG n'est pas défini la méthode "ConditionalMethod" n'existera tout simplement pas ! Ce qui implique que la compilation plantera si elle est appelée dans le reste du code…

Pas très pratique. Il y a mieux, l'attribut Conditional. Il permet à la fois une écriture plus propre (pas de#if/#endif) et plus lisible mais surtout il supprime l'exécution de la

méthode même là où elle est appelée et cela sans erreur de compilation ni besoin d'intervenir !

On écrira alors plutôt :

```
[Conditional("DEBUG")]

static void DebugMethod()

{ ... }
```

Bien entendu n'importe quelle condition peut être testée et l'intérêt de Conditional va bien au-delà du débogue où elle est certes d'une aide précieuse (instrumentalisation d'un code qui apparait ou disparait automatiquement dès lors qu'on est dans le mode de débogue ou non). Des méthodes peuvent ainsi être appelée et exécutée par un code commun uniquement sous certaines conditions (Windows 8 ou Windows Phone par exemple) sans écrire un seul #if…

Les mots clés

C# est un langage assez concis et généralement on en connait tous les mots clés, mais comme la section "syntaxe" nous l'a fait voir cette connaissance n'est pas forcément profonde… Il y a des mots clés qu'on sait exister sans jamais les avoir utilisés. Certes on aura l'impression de ne rien apprendre de nouveau, mais en réalité on ne penserait pas à les utiliser quand le bon moment se présentera…

C'est particulièrement le cas du premier mot clé de cette section, yield.

Yield

Yield fait partie de ces mots clés dont on a forcément entendu parlé mais qu'on ne saurait pas forcément utiliser correctement.

Yield sert à retourner des valeurs, une par une (ainsi que la fin de la liste) à l'intérieur de bloc de code de type itérateur.

La syntaxe est la plus simple du monde puisqu'on écrit soit yield return <expression>; soit yield break; pour stopper l'itération (ou énumération).

Ce n'est donc pas un problème de syntaxe qui m'amène à vous parler de yield, sinon il serait apparu plus haut dans la section consacrée à cet aspect du langage. C'est logique.

Avec yield le problème c'est plutôt de bien comprendre comment l'utiliser. Pourtant ce n'est pas bien compliqué. Comme je le disais yield s'utilise dans un itérateur. Mais qu'est-ce que c'est qu'un itérateur ?

Un itérateur c'est une section de code qui retourne une séquence ordonnée de valeurs du même type (définition tirée de la documentation MSDN).

Un itérateur peut être utilisé comme corps d'une méthode, d'un opérateur ou d'un accesseur Get. Il utilise yield dont je viens de parler soit pour retourner une valeur soit pour indiquer la fin de la séquence.

Une classe peut implémenter plusieurs itérateurs. La contrainte étant, ce qui tombe sous le sens, qu'ils aient des noms différents et qu'ils puissent être invoqués par un Foreach de la façon suivante : foreach (var v in Voiture.Modèles) où Voiture est une instance de la classe imaginaire Véhicule et où Modèles est l'itérateur qui retourne la liste des modèles de la voiture en question. Ceci est bien entendu un exemple fictif servant à situer le morceau de code.

Le type de retour d'un itérateur doit être choisi parmi les suivants : IEnumerable, IEnumerator, IEnumerable<T> ou IEnumerator<T>.

Je n'entrerai pas dans les détails de ces types, MSDN sera comme d'habitude un ami fidèle pour ce genre de recherches (même si en général il est préférable de chercher sous Google pour accéder à ce qu'on désire dans MSDN ce qui est un comble…).

Pour mieux comprendre prenons un exemple d'itérateur implémenté au niveau de la classe. Ici nous définirons une classe DayOfTheWeek (jour de la semaine) que nous pourrons balayer avec un foreach (l'une de ses instances, pas la classe elle-même bien sûr).

```
void Main()
{
    // Création de l'instance
    DaysOfTheWeek week = new DaysOfTheWeek();

    // Itération
    foreach (string day in week) Console.Write(day + "; ");
}

public class DaysOfTheWeek : System.Collections.IEnumerable
{
    string[] m_Days = { "Lun", "Mar", "Mer", "Jeu", "Ven", "Sam", "Dim" };

    public System.Collections.IEnumerator GetEnumerator()
    {
        for (int i = 0; i < m_Days.Length; i++)
            yield return m_Days[i];
    }
}
```

Ce qui retournera à la console : Lun; Mar; Mer; Jeu; Ven; Sam; Dim;

Comme on le voit sur cet exemple la classe DayOfTheWeek implémente l'interface IEnumerable. De fait toute instance de cette classe se comporte comme un

fournisseur de liste. Cela impose de déclarer `GetEnumerator`()qui est la méthode imposée par l'interface. Son type est IEnumerator.

Cet énumérateur se contente de faire une boucle qui retourne à un à un les éléments de la liste m_Days. Pour ce faire la méthode utilise un boucle for calée sur le nombre d'éléments de la liste, chaque itération utilisant yield return <expression> (l'expression étant ici un simple élément de la liste).

L'avantage des itérateurs est évident puisque la méthode appelante n'est pas obligée de traiter une liste entière, uniquement les éléments qu'elle reçoit et qu'elle peut effectuer un travail sur chaque élément puis demander le suivant ou non. Une liste se retourne d'un bloc, pas élément par élément.

Avec IEnumerable et l'implémentation d'une méthode retournant un IEnumerator on peut ainsi fournir des valeurs les unes après les autres. Ces valeurs sont soit issues d'une liste interne comme dans l'exemple ci-dessus soit calculées à la volée ce qui devient plus intéressant en ouvrant des horizons nouveaux…

Revenons à yield. Il s'utilise donc dans un bloc de code pour fourni un objet énumérateur ou signaler la fin d'une séquence. Comme on le remarque au passage sur notre exemple la fin de séquence est optionnelle, ici nous avons choisi de balayer une liste interne par for et nul besoin d'un signal de fin, la boucle for s'arrête d'elle-même.

On notera que yield se comporte comme une sorte de callback puisque dans un foreach chaque élément est retourné un à un, ce qui impose un aller retour entre l'intérieur de la boucle de l'énumérateur et le code appelant. Dans notre exemple cela signifie que la boucle for interne à `GetEnumerator`() est saucissonnée élément par élément et que le foreach appelant obtient une réponse à la fois qu'il peut traiter comme il le veut avant de réclamer la suivante. Ce mécanisme est en lui même intéressant à comprendre car il est à la base de nombreuses utilisations "inventives" qui peuvent être faites de yield…

Si nous avons vu dans l'exemple de code comment faire pour qu'une classe soit d'emblée énumérable, le code suivant montre comment implémenter un énumérateur nommé à l'intérieur d'une classe n'étant pas elle-même énumérable.

Ici la classe Calcul est une classe statique de service, elle fourni des opérations mathématiques un peu comme l'espace de noms Math du framework. Parmi les opérations disponibles on trouve Power qui élève un nombre à une puissance donnée. Toutefois ici ce n'est pas seulement le résultat du calcul qui nous intéresse mais aussi les "valeurs intermédiaires" de celui-ci. De fait la méthode `Power`() va être implémentée sous la forme d'unIEnumerable(type de retour) qui renverra chaque

valeur intermédiaire jusqu'à la valeur finale. Power étant obtenu par des itérations successives c'est une opération qui se prête bien à l'utilisation de yield. Voici ce code :

```
void Main()
{
    foreach (int i in Calcul.Power(3, 9)) Console.Write("{0}; ", i);
}

public static class Calcul
{
    public static IEnumerable Power(int number, int exponent)
    {
        int counter = 0;
        int result = 1;
        while (counter++ < exponent)
        {
            result = result * number;
            yield return result;
        }
    }
}
```

Ce qui retournera à la console : `3; 9; 27; 81; 243; 729; 2187; 6561; 19683;`

Yield peut aussi être utilisé pour fabriquer des listes filtrées depuis d'autres listes… regardez le code très simple ci-dessous :

```
static IEnumerable<int> FilterWithYield()

{
            foreach (int i in MyList)
            {
                if (i > 10) yield return i;
            }
}
```

Partant d'une liste (MyList) supposée contenir des entiers, la méthode FilterWithYield est définie comme un IEnumerable d'entiers et son code utilise un foreach sur la liste originale en conjonction avec un yield uniquement quand la valeur obtenue est supérieure à 10. De fait on obtient une sorte de vue vivante sur MyList filtrée selon des critères qui peuvent être complexes. La valeur retournée peut même être calculée en fonction de la valeur originale. Ce n'est qu'un exemple parmi des centaines…

Comme on le voit yield est un mot clé plein de ressources ! Il est directement connecté à la notion de bloc de code de type énumérateur et aux interfaces IEnumerable que nous avons vu à l'œuvre et IEnumerator qui n'est que la classe de base dans le framework .NET pour tous les énumérateurs non génériques.

Il n'y a rien de compliqué à comprendre dans yield lui-même mais la difficulté semble être plutôt d'y penser et d'imaginer du code qui en tire partie. Le présent rappel vous donnera peut-être envie de vous en servir un peu plus souvent dorénavant …

Default

Même si toutes les variables de types par référence peuvent se voir mettre à null sans soucis "null" n'est pas forcément "null", notamment pour les types par valeur. C'est le cas pour les booléens (false), les entiers (0) et tous les types par valeur en général.

Lorsqu'on écrit du code générique il y a des circonstances où connaitre à l'avance la "bonne" valeur "null" à indiquer est impossible.

C'est là qu'intervient default(T). Plutôt que d'utiliser une constante comme "null" qui ne marchera pas à tous les coups, mieux vaut utiliser default(T) qui a l'avantage de fonctionner dans tous les cas…

```
var a = default(int); // donnera int a = 0
var b = default(bool) // donnera bool b = false
```

Volatile

En voilà une si vous me dîtes que vous le connaissez déjà j'aurai quelques doutes et si en plus vous m'assurez que vous l'utilisez tous les jours j'aurai vraiment du mal à vous croire sans preuves solides ! 

Nous avons vu dans un récent billet l'utilisation de Interlocked qui permet d'incrémenter ou décrémenter des variables de façon fiable en multithreading évitant souvent l'utilisation de `lock`().

Volatile rempli un peu un rôle similaire au niveau d'un champ. S'il est déclaré volatile les lectures et écritures concurrentes sont assurées de travailler sur la dernière valeur disponible (au lieu d'une valeur en cache dans un registre du CPU par exemple). Volatile obtient des "*aquire-fence*" pour les read ou des *release-fence* pour les write.

Volatile ne signifie pas seulement "*s'assurer que le compilateur et le jitter ne feront aucune réorganisation de code ou du caching dans les registres du CPU ni aucune autre optimisation*", ce mot clé signifie aussi "*demander aux processeurs de faire le nécessaire pour que quoi qu'ils fassent ils s'assurent de délivrer la dernière valeur disponible, même si cela signifie d'arrêter tous les processeurs pour qu'ils synchronisent leurs caches entre eux et avec la mémoire centrale*".

Toutefois il faut se méfier un peu de cette "garantie" car le compilateur C# s'offre quelques libertés pour optimiser le code final. Par exemple il peut swapper des reads et des writes. Cela est rare puisque un read/read, read/write ou un write/write ne

seront pas swappés, mais un write/read peut l'être… Et dans ce cas Volatile ne sert plus à grand chose.

Ceci expliquant cela, le multithreading étant déjà bien assez subtile comme cela et réservant suffisamment de surprises, il est probable que Volatile ne soit par trop biscornu pour être devenu populaire…

Il en va de même de `Threading.MemoryBarrier()` que vous ne connaissez certainement pas et qui permet pourtant en multithread d'obtenir le même type de barrière pour une portion de code.

Volatile ne peut agir sur les variables passées par référence, c'est pourquoi le framework offre `VolatileRead()` et `VolatileWrite()` qui sont construits en utilisant `MemoryBarrier();`

Voici un bout de code qui met en évidence le problème (à compiler en mode Release avec les optimisations pour avoir une chance de voir le blocage) :

```
class Test
{
    private bool _loop = true;

    public static void Main()
    {
        Test test1 = new Test();

        // _loop est mis à false dans un autre thread
        new Thread(() => { test1._loop = false;}).Start();

        // normalement la lecteur doit être ok… mais…
        while (test1._loop == true) ;

        // ...la boucle ne termine jamais…
    }
}
```

En ajoutant volatile à la définition de _loop le problème cesse.

Mais rassurez-vous, si tout cela ne vous dit rien c'est que vous n'êtes pas un expert en multithreading et que tant que vous savez utiliser `lock`() à bon escient ou des objets immuables vous pourrez tout de même écrire du code multithread tout à fait correct !

Extern Alias

Encore une possibilité du langage qui n'est pas souvent utilisée pourtant dans certains gros projets qui connaissent des améliorations dans le temps cela pourrait l'être avec plus de bonheur que de trafiquer les espaces de noms existants…

Car de quoi s'agit-il ici ? Le problème est simple, un code peut très bien vouloir accéder à deux versions différentes d'un même espace de nom. Supposons que dans la première version de notre application nous ayons créé un objet Grille qui a donné le binaire grid.dll. Supposons maintenant que nous ayons au fil du temps décidé de créer une amélioration de cette grille compilée dans le binaire grid2.dll. L'assemblage est exactement le même, les espaces de noms sont identiques, les classes aussi. C'est juste le code qui diffère.

Imaginons maintenant que notre application doit utiliser la nouvelle grille dans une page où l'ancienne est aussi utilisée et que pour des raisons variées on ne puisse pas ou ne veuille pas remplacer l'ancienne en place. Il faudra donc dans ce code accéder à deux classes Grille définies dans le même espace de noms… Et là ce n'est pas possible. Il faut que l'arborescence de noms possède une petite différence au moins sinon le compilateur ne sait pas quelle classe choisir.

La solution vue le plus souvent est le "bricolage" de l'espace de nom de la nouvelle classe ou bien le changement de nom de cette dernière. Cela n'est pas optimal. Créer une différence fictive entre deux espaces de noms qui sont en réalité identiques est la porte ouverte à toutes les confusions. Quant à attribuer un nouveau nom à une même classe rendant les mêmes services à quelques nuances près, là encore la porte s'ouvre sur les bogues à venir tout en rendant difficile l'échange là où il est possible (puisqu'il faudra changer le nom de classe utilisé dans le code existant).

Il y a mieux. Il y a Extern Alias. Cela fonctionne comme la définition d'un alias sur un espace de noms sauf qu'ici on indique le nom de l'assemblage binaire.

Sous VS, dans les références, on peut voir que par défaut les assemblages ont une propriété "Alias" qui est positionnée à Global. C'est en donnant un nom différent ici qu'on peut utiliser ce dernier avec "extern alias" dans son code.

Si nous définissions Grille1 et Grille2 comme alias pour les références aux deux assemblages, dans le code on peut maintenant utiliser Grille1::Grille pour accéder à la classe Grille dans version 1 et Grille2::Grille pour sa version 2, le tout sans changer le code existant, sans bricoler les espaces de noms etc.

Bien entendu il ne s'agit pas ici d'une guidline, on doit éviter de se retrouver dans de telles situations, mais elles existent et alors il faut savoir utiliser ce que le langage propose pour y répondre de la façon la plus appropriée…

On notera pour terminer que "extern" est aussi utilisé pour désigner un code externe non managé, ce qui n'a rien à voir avec le présent topic…

Les autres possibilités de C#

Il existe d'autres possibilités du langage qui restent peu utilisées. En tout cas on rencontre soit des gens qui s'en servent sans problème soit d'autres qui les ignorent totalement. La situation médiane se retrouvant bizarrement assez rarement sur le terrain.

Nullable

Il en va ainsi des types "nullables", c'est à dire ces types par valeur qui normalement ne peuvent pas être nuls, comme un booléen par exemple. A l'initialisation de la mémoire un booléen est rempli de "0" sa valeur numérique vaut zéro, et cela se traduit par "false". Il peut être positionné à "true", remis à "false", etc, mais jamais il ne sera "null".

Les types "nullables" exploitent un artifice de notation lors de la déclaration d'un champ, le point d'interrogation derrière le nom du type, pour autoriser la valeur nulle dans de tels cas.

Nous ne discuterons pas de l'utilité ni du bon usage des nullables, ils existent, c'est une fonctionnalité du langage, et cela rend service à qui sait s'en servir !

La notation est par exemple pour définir une variable "a" de type entier et nullable (et lui affecter la valeur nulle au passage) : int? a = null;

C'est d'ailleurs dans ces cas là que l'opérateur ?? que je vous ai présenté plus haut prend son sens. Car maintenant notre "a" peut très bien être nul, ce qui impose de l'utiliser en prenant en compte cette éventualité b = a ?? 0; permet d'attribuer à "b" la valeur de "a" s'il n'est pas nulle et la valeur zéro dans le cas contraire, en une instruction et sans "if".

Les types nullables sont très utiles pour des statistiques ou des questionnaires. Ils permettent par la valeur nulle de matérialiser la "non réponse". Une question "nombre d'enfants" si elle est stockée dans un entier (ce qui permettra ensuite des calculs comme la somme, la moyenne, la médiane…) n'est pas à même de faire la différence en "zéro enfant" comme réponse et "zéro enfant" parce que la réponse n'a pas été donnée. En utilisant un int? pour conserver la valeur on peut attribuer à zéro sa véritable signification (pas d'enfant) car en cas de non réponse la valeur sera nulle.

Il y a d'autres domaines où les nullables peuvent jouer un rôle important, mais cela dépasse le sujet traité !

Conclusion

Je vais conclure ici car ce n'est qu'un billet pas un livre sur C#… Les subtilités, grandes ou petites, sont nombreuses et il n'est pas possible de faire le tour de la question même dans un grand billet.

Déjà ici tracer une ligne claire est difficile. Certains lecteurs trouveront pour certains passages qu'il s'agit du B.A.BA du langage alors que d'autres y découvriront une possibilité qu'ils n'ont jamais utilisée, et ce même si globalement il s'agit de développeurs de même niveau, chacun n'ayant pas les mêmes "lacunes".

Le but à peine caché de ce billet n'est donc pas d'être exhaustif mais bien d'attirer votre attention sur C# qu'on pense connaitre alors qu'il réserve bien des surprises à celui qui farfouille un peu.

Développer est un plaisir, cela doit l'être en tout cas. Un langage est un ami de tous les jours. Bien connaitre ses amis permet de passer de meilleurs moments ensembles…

Les nouveautés de C# 3

Débuter par la version 3 peut sembler un choix étrange, mais c'est à partir de cette version que j'ai écrit beaucoup d'articles sur C# d'une part et, d'autre part, exposer les versions 1 et 2 serait transformer ce livre déjà long en un cours complet sur C# ce qui n'est pas le but. Pour cela je suis à votre disposition pour toute formation adaptée à vos besoins, et si vous vous sentez d'aborder ce langage seul le Web est aujourd'hui largement pourvu en articles sur la question. Dot.Blog commence à parler véritablement de C# avec la version 3 sortie en 2007 comme Dot.Blog, elle marque un tournant dans l'évolution du langage. Tout commence là donc !

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, 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.

Avec le recul lorsqu'on sait ce que va être Entity Framework 7, lorsqu'on sait ce qu'est devenu C# jusqu'à sa version 6 on mesure encore plus la chance que nous avons d'avoir fait le bon choix !

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 !

Les nouveautés de C# 4

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…

Les nouveautés de 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...

Les nouveautés de C# 6

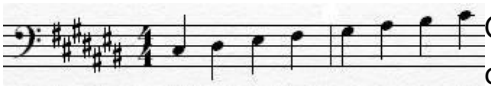
C# surprendra toujours. Chaque version est l'occasion de bouleversements tranquilles, d'ajouts qui semblent parfois mineurs et qui se révèlent plus tard des pièces essentiels d'un puzzle complexe où l'OS et la plateforme, ses API, forment un paysage toujours différents, s'adaptant aux besoins changeant d'un monde en mouvement.

Tous les ajouts ne sont pas aussi énormes que LINQ par son impact sur les esprits et les possibilités incoryables dont il a doté C#, mais des petites choses comme async/await trasnforment du tout au tout la façon de programmer, en profondeur. Il n'y a pas de modifications mineures avec C#.

C# 6 et ses petites nouveautés

C# va arriver en version 6. Chacune de ses versions a été l'occasion de petites ou de grandes avancées. Que nous réserve la 6ème cuvée ?

C# et Roslyn



Comme je vous en ai déjà parlé je ferai court: Roslyn est un ensemble de compilateurs et d'API d'analyse Open Source releasé par Microsoft principalement autour de C# et de VB, donc sous .NET.

C# 6 est désormais releasé et reprend ce que Roslyn aura apporté. Connaître l'un c'est donc savoir ce que contiend l'autre…

Roslyn n'est pas forcément une nouveauté, la première CTP datant d'octobre 2011… Le temps passe vite ! D'autres ont suivi. La dernière qui nous a rapprochés de la version finale date du Build de San Francisco en avril 2014.

On sait d'ailleurs que Xamarin a annoncé à cette occasion que le nouveau compilateur et ses outils seront intégrés à Xamarin Studio.

Il a même existé un package Nuget pour VS 2013 qui permettait de tester Roslyn. Toutes les fonctionnalités n'étaient pas encore implémentées toutefois. Roslyn a permi de définir des features qui sont désormais intégrées à C# 6 et à VB 13.

Quelques exemples

En annonçant C# 6 on peut se poser la question de savoir que peut-on *encore* ajouter à C# ?

Et répondre est assez facile : pas grand chose…

Le langage a tellement évolué avec les expressions Lambda, Linq, et tout un tas de choses dont j'ai parlé en long et en large qu'il semble bien peu probable que C# 6 contienne des avancées fantastiques car C# est *déjà* fantastique.

C'est donc plutôt du côté des "syntactic sugar", ces petites gâteries syntaxiques qui facilitent l'écriture qu'il faut regarder. Les grandes nouveautés qui cassent la baraque ne font pas partie du lot des nouveautés, elles sont déjà dans C# 5 et ses prédécesseurs…

Toutefois le principe même de Roslyn et de ses API, son code Open Source, tout cela peut être vu comme un *énorme changement* pour C# et VB. Toutes les grandes avancées ne se traduisent pas forcément par l'ajout de nouveaux mots clé dans le langage. Et comme je le disais plus, aucune modification de C# n'est mineure. Celles qui nous paraissent moins extraordinaires ne traduisent que notre ignorance et notre manque d'expérience à les pratiquer…

Les propriétés

Les propriétés automatiques de C# sont très pratiques. J'avoue ne pas trop les utiliser car en MVVM on s'oblige plutôt à propager les changements de valeurs avec INPC et cela n'est pas possible avec des propriétés automatiques. Et c'est bien dommage.

Mais ce n'est pas cet ajout que nous réserve C# 6 mais plus simplement la possibilité d'initialiser la valeur d'une propriété automatique et read-only (cela limite un peu les cas) sans passer par du code dans le constructeur de la classe.

Ainsi le code suivant :

```
public int MaPropriété { get; } = 256;
```

permet de déclarer la propriété automatique "`MaPropriété`", de type entier, possédant un getter et pas de setter (elle est donc bien read-only) tout en l'initialisant à la valeur `256`.

Cela peut servir mais ce n'est pas "killer" je vous l'accorde.

Les initialisations

Initialiser des valeurs est souvent fastidieux. Alors autant en simplifier la syntaxe. Ainsi tout objet qui supporte un indexeur peut être initialisé en référençant directement les index. Par exemple un dictionnaire :

```
var d = new Dictionary<string,int>{ ["A"]=22, ["B"]=100, ["robert"]=0 };
```

Les entrées dans le dictionnaire pour les index "A", "B" et "robert" sont créées et les valeurs indiquées sont affectées à ces nouvelles entrées. Pas besoin de passer une fois encore par le code du constructeur ou tout autre code d'initialisation.

L'accès aux propriétés indexées peut aussi se faire en utilisant plutôt une syntaxe de type propriété que index, le code suivant clarifiera cette phrase confuse … :

```
MonObjetIndexé[“toto”]=22;
```

s'écrira aussi `MonObjetIndexé.$toto = 22;`

La valeur de l'index précédée du signe dollar forme comme un nouveau nom de propriété et s'écrit précédée du point comme toute propriété.

Plus exotique est la classe définit par un code qui va créer automatiquement les propriétés et les initialiser. Ainsi :

```
public class UnPoint(int x, int y) { }
```

est un raccourci plutôt amusant pour le code suivant :

```
public class UnPoint
{
  private int x,y;
  public UnPoint(int x, int y)
  {
   this.x = x;
   this.y = y;
  }
}
```

Amusant non ?

Bon comme je l'annonçais ça ne casse pas la baraque, c'est vrai.

Et que dire que de :

```
var b = new Button { OnClick+=monGestionnaireDeClic};
```

Ce code créée une nouvelle instance de `Button` et attribue en même temps l'adresse de "`monGestionnaireDeClic`" (qu'on suppose définit ailleurs bien entendu) à l'évènement OnClick… On dirait presque du JavaScript, ce qui en ravira certains et fera faire la moue à d'autres comme moi.

Expressions et littéraux

Si les astuces précédentes ne vous feront peut-être pas sauter de joie, celle qui vient saura peut-être vous convaincre de l'intérêt de ces nouvelles syntaxes de C# 6 :

```
var x = (var y=rnd(10);y*y);
```

Bizarre non ? C'est l'utilisation du point virgule qui fait tout ici. Il permet de chaîner des expressions, la valeur finale étant celle de la dernière.

Dans le code exemple cela attribuera à x la valeur du carré d'un nombre aléatoire (en supposant `rnd()` une fonction retournant un tel nombre).

L'intérêt ? Comme on le voit dans cet exemple il n'est pas toujours possible d'attribuer à une variable le résultat d'une suite d'opérations sans passer par un code intermédiaire. Ici effectuer le carré dans `rnd()` n'aurait pas la même signification, de même faire un `rnd(10*rnd(10)` effectuerait deux tirages au sort ce qui est très différent.

En chaînant les opérations par un point virgule on peut définir des valeurs complexes en faisant l'économie de plusieurs lignes de code "intermédiaire".

Pas fantastique mais pouvant ajouter de la clarté au code.

Dans un même esprit :

```
var y =  (int x = 3) * 3 ;
```

Ce code attribuera la valeur "9" à y en définissant "x" à 3…

C# 6 ajoute comme Roslyn les littéraux binaires. Cela est parfois utile. On pourra donc écrire
`var b = 0b0100;` pour définir b à "100" binaire, soit 4 en décimal.

Comme le binaire ou l'hexadécimal sont généralement des valeurs qu'on peut découper en unités ayant un sens (par demi octet, octet, double, etc) il sera possible d'ajouter des soulignés pour séparer les groupes sans perdre le caractère numérique du littéral : `int x = 0xFF_00_0A_A0 ;` est plus lisible sans erreur que `0XFF000AA0;`

On notera enfin que les méthodes statiques seront plus faciles à utiliser ce qui sera une vraie simplification. Prenons le code suivant :

```
var b = Math.Sin(Math.Sqrt(5));
```

Il sera plus agréable d'écrire (et plus facile à lire) :

```
using System.Maths;
…
var b = Sin(Sqrt(5));
```

Exceptions

Normalement C# 6 introduit la possibilité d'appeler du code asynchrone dans les parties `Catch` et `Finally` des blocs `Try` :

```
try
{
  … code …
}
catch       { await cleanUp(yyy); }
```

```
finally { await cleanUp(xxx); }
```

Une autre possibilité intéressante consistera à pouvoir utiliser des filtres dans les `Catch` :

```
try { code… } catch(Exception e) if (e.xxx=22) { …bloc catch… }
```

Ou en est-on ?

Il ne s'agit pour l'instant que de simples exemples de ce que contient C# 6. Pour savoir où en est l'implémentation de Roslyn et ce qui est surement - ou pas - intégré à C# et VB on peut consulter la page "*language feature implementation status*" – le statut de l'implémentation des possibilités du langage – en accédant sur CodePlex via une URL très longue que je vous propose en format raccourci : http://goo.gl/pi0jIA

Conclusion

C# 6 est une cuvée sage, avec quelques petits su-sucres syntaxiques pas forcément tous utiles mais dont on deviendra certainement addict de l'un ou de l'autre…

C# est un langage qui est arrivé à une maturité étonnante, il a su grandir sans exploser, mais il arrive maintenant à un plateau : faire beaucoup mieux que la version précédente va devenir de plus en plus difficile.

En général c'est là qu'un produit meurt en informatique. Il y a une certaine connerie partagée, n'ayons pas peur des mots, qui veut que si une plateforme ou un langage n'apporte pas de nouvelles fonctionnalités "awesome" c'est qu'il "stagne", donc qu'il est mort.

C'est une connerie disais-je parce que bien entendu cela signifie juste que le produit est mature et peut enfin être pleinement utilisé en toute sérénité pour faire des choses intelligentes avec…

C# arrive à ce point.

L'intelligence qui consistera à s'en servir pour faire de beaux logiciels l'emportera-t-elle sur la connerie qui voudra y voir un langage "en panne", "hasbeen" ?

Je connais trop mon métier pour me risquer à une prévision ! Et vous ?

PS: Pour les curieux, la portée qu'on découvre en début d'article est la gamme de do dièse majeur, donc C# en notation américaine…

C# 6 : amélioration de la gestion des exceptions

Roslyn qu'on peut appeler maintenant C# 6 va bientôt être une réalité, en attendant on peut en découvrir certains aspects, par exemple du côté des exceptions…

Roslyn / C# 6

J'ai déjà abordé en avant première certains des aspects de C# 6 dans un billet du 15 mai dernier, je renvoie ainsi le lecteur intéressé à ce dernier qui y trouvera d'autres annonces et quelques mots sur les nouveautés touchant la gestion des exceptions.

Les exceptions de C# 6

Justement il ne s'agissait que de quelques mots. Aujourd'hui on peut présenter ces avancées de façon plus complète. Allons-y !

Les améliorations de la gestion des exceptions ne sont pas faramineuses disons-le tout de suite, mais elles sont assez importantes. Il y a deux avancées principales. La première est une amélioration de async/await et la seconde est le support du filtrage des exceptions.

C# 5 a introduit async/await dont je ne parlerais pas ici renvoyant le lecteur à mes billets sur le sujet (dont celui du 23 juin dernier qui est assez précis). Avec cet ajout les développeurs ont gagné beaucoup dans la simplification de l'écriture du code asynchrone laissant au compilateur la charge fastidieuse de générer les callbacks sous-jacents. De même grâce à ces mots clés contextuels il a été possible d'aborder le pattern TAP (Task-based Asynchronous Pattern) de façon plus simple et naturelle.

Hélas, l'équipe en charge de ces améliorations n'a pas eu le temps de finir le travail dans les blocs try/catch/finally. Ainsi, si await était utilisable dans le corps d'un Try, il n'était pas possible de s'en servir dans le Catch ou le Finally.

Le besoin a été un peu sous-estimé et la priorité n'a pas été donnée à une implémentation complète de async/await dans les gestionnaires d'exception. Le succès de ces derniers et la recherche d'un code toujours plus asynchrone ont d'un autre côté tenté de plus en plus de développeurs d'utiliser une gestion asynchrone des exceptions, impossible en C# 5 donc. Il y avait là un trou à boucher pour terminer l'édifice.

Traitement asynchrone du Catch

Heureusement les choses ont avancé et C# 6 ne possède plus cette limite ! Il est donc possible de gérer de façon asynchrone le code d'un Catch ou d'un Finally comme le montre le code ci-dessous (sur la partie Catch uniquement) :

```
try
{
    var webRequest =  WebRequest.Create("http://www.e-naxos.com");
    var response =  await webRequest.GetResponseAsync();
    // ...
}
catch (WebException exception)
{
    await LogError(exception);
}
```

Filtrage des exceptions

Une autre avancée dans la gestion des exceptions de C# 6 est le filtrage ou plus précisément le support des filtres d'exception. Pour une fois c'est C# qui était en retard et il ne fait donc ici que rattraper VB et F#.

Le filtrage des exceptions existait déjà sur le type de ces dernières depuis les débuts de C#, il s'agit donc ici de spécialiser ce filtre pour affiner les conditions de ce filtrage. Dans la pratique on peut ainsi ajouter un "if" après le Catch pour cibler certaines valeurs de l'exception qui ne dépendent pas seulement du nom de sa classe mais des informations que l'instance véhicule. Un bout de code rendra cela plus parlant :

```
public void Test()
{
    try
    {
        throw new Win32Exception(Marshal.GetLastWin32Error());
    }
    catch (Win32Exception exception) if
                    (exception.NativeErrorCode == 0x00042)
    {
        // traitement de l'erreur native 42 de l'OS
    }
}
```

Tout ce trouve donc dans ce "if" ajouté après le "catch". Ce dernier filtre sur la classe de l'exception et le "if" permet d'affiner sur le contenu réel de l'exception qui arrive. Ici on ne souhaite que gérer les exceptions de type Win32Exception (natives donc) et uniquement celle dont le code d'erreur natif est 42.

Bien qu'assez simple cette nouveauté est intéressante à plus d'un titre. Car beaucoup de nouveautés de C# 6 pourraient s'écrire en C# 5 avec plus de code mais celle-ci est totalement impossible à simuler proprement. Il faut en C# 5 attraper toutes les exceptions d'une classe donnée puis dans le bloc Catch vérifier si on désire ou non traiter l'erreur, dans la négative il faut faire un re-throw de l'exception. Mécaniquement c'est une émulation assez proche mais techniquement elle ne l'est pas notamment dans le suivi du StackTrace par exemple.

Conclusion

Rien d'énorme en soi, mais deux ajouts qui rendent C# encore plus fin, plus subtile et précis. Il ne faut plus s'attendre à des ajouts aussi énormes que LINQ par exemple, même si async/await me semble d'une importance capitale tout comme Parallel-LINQ mais cela a été fait dans C# 5, la version 6 arrive avec son nouveau moteur, en open source, et quelques ajouts intéressants. C'est la version 7 qui sera certainement plus marquante pour le développeur mais il est trop tôt pour en parler, la 6 n'étant pour l'instant que simplement testable dans une preview disponible ici

: https://roslyn.codeplex.com/. Le passage du projet Roselyn à celui de C# 6 en open source est déjà en soi quelque chose d'énorme ne l'oublions pas !

C# 6 – Tester le null plus facilement

La série des mini-tips sur les nouveautés de C# 6.0 continue. Aujourd'hui testons plus facilement les nulls pour l'écriture d'un code plus clair…

Null-Conditional Operators

Il s'agit d'une nouvelle feature de C# 6.0 qui va définitivement améliorer l'écriture du code et votre productivité ! Comme toute clarification du code l'effet de bord est aussi la diminution du risque de bogues ce qui est toujours bon à prendre…

L'astuce est d'une grande simplicité puisqu'il suffit d'ajouter un point d'interrogation après le nom d'une instance d'un type "nullable" avant d'indiquer le nom de la propriété à atteindre. Regardons cela de plus près :

```
#region Simple Condition
// if employee is NOT Null return Address, else return NULL
address = emp != null ? emp.Address : null; // old way

address = emp?.Address; // new way

#endregion Simple Condition
```

La première ligne montre le code habituel utilisant l'opérateur "?:" qui était déjà une simplification de syntaxe lorsqu'il fut ajouter à C#. Mais pour tester un null sur l'accès à une propriété c'est encore un peu verbeux.

Le rectangle rouge entoure l'utilisation de cette nouvelle syntaxe dans la seconde ligne qui fait exactement la même chose : si "emp" est nul alors on retourne "nul" sinon on accède à la propriété "Address". Un seul point d'interrogation suffit dans ce cas.

Cela fonctionne bien entendu en mode "nested" (imbriqué) et on voit alors toute la clarification qu'apporte cette syntaxe :

```
#region Nested Conditions
// if employee is NOT Null and also Address is NOT Null return City, else return NULL
if (emp != null && emp.Address != null) // old way
{
    city = emp.Address.City;
}
else
{
    city = null;
}
city = emp != null ? (emp.Address != null ? emp.Address.City : null) : null; // old way

city = emp?.Address?.City; // new way

#endregion Nested Conditions
```

Ici on cherche à accéder à la propriété "City" de la propriété "Address" de l'instance "emp" (qu'on suppose être de classe "Employee" par exemple).

Toute la première partie code montre l'utilisation de la syntaxe classique : celle sans l'opérateur "`?:`", celle avec.

La dernière petite ligne minuscule nous montre l'équivalent avec la nouvelle syntaxe… Incroyablement plus concis, propre, lisible, maintenable et forcément moins sujet aux bogues.

On peut voir ci-dessous comment utiliser le nouvel opérateur en conjonction avec "`??`" pour retourner facilement une valeur par défaut différente de "null" :

```
#region Default Values in Condition
// if employee is NOT Null and MemberOfGroups is NOT Null, return groups count, else return -1
groupsCount = (emp != null && emp.MemberOfGroups != null) ? emp.MemberOfGroups.Count : -1; // old way

groupsCount = emp?.MemberOfGroups?.Count ?? -1; // new way

#endregion Default Values in Condition
```

La première ligne montre l'ancienne façon habituelle d'écrire le code, la ligne avec le rectangle rouge la nouvelle syntaxe. Dans cet exemple un "null" sera retourné sans erreur d'exécution si '""emp" est nul, si "emp.MemberOfGroups" est nul. De fait Count ne sera pas accéder si ces conditions de null se présentent, sinon c'est bien le Count qui sera retourné. Mais si l'une des conditions de null est rencontrée l'opérateur "??" retournera alors la valeur "-1".

Conclusion

C# 6.0 n'apporte pas toujours des fonctions aussi puissantes que ne le furent par exemple l'apport de Linq ou du parallélisme, mais en améliorant sans cesse sa syntaxe, sa lisibilité, sa concision, Microsoft nous offre encore et toujours un langage au top de la technologie, agréable et puissant.

J'ai toujours aimé C# (mon premier titre MVP il y 8 ans l'était pour ce langage) et j'avoue que plus le temps passe plus je l'aime et moins je pourrais passer à autre chose car tout le reste me semble vraiment vieux, vide ou bricolé, voire les trois…

C# 6 – les "Expression-bodied function members"

Encore une nouveauté qui va faciliter l'écriture du code… Sous ce nom un peu complexe d'*Expression-bodied function members* se cache quelque chose d'assez simple mais de très puissant…

Expression-bodied function members
Un nom à rallonge qu'on pourrait traduire par "*membres de fonction dont le corps est transformé en expression*"…

Qu'est-ce à dire ?

Tout simplement que le corps de certaines constructions du langage C# peut désormais être directement remplacé par une expression (type expression Lambda). A noter que pour tester cette syntaxe il vous faudra un VS 2015 et un framework .NET 4.6 (au minimum, mais lorsque j'écris ces lignes c'est le maximum !).

Code plus court (beaucoup plus court), donc plus lisible, plus maintenable, etc, vous connaissez la chanson 

Regardons un premier exemple :

```
0 references
public string NewName => "Kunal Chowdhury"; // new way
0 references
public string OldName // old way
{
    get
    {
        return "Kunal Chowdhury";
    }
}
```

La déclaration de la propriété OldName montre l'ancienne façon de procéder. Beaucoup de code pour retourner une constante (le nom du bloggeur a qui j'ai emprunté cette capture écran, rendons à césar…) . Mais cette constante nous la voulons sous la forme d'une propriété pour des tas de raisons (peu importe lesquelles ici). Nous sommes donc obligés de suivre cette syntaxe.

La première ligne montre la nouvelle façon de faire. C'est bien une propriété qui est encore déclarée et non une constante mais le corps est remplacé par une fonction (suivant "=>"). Une petite ligne pour dire la même chose.

Là où cela devient encore plus intéressant c'est qu'il est possible de faire la même chose pour les méthodes…

```
0 references
public int NewSum(int a, int b) => a + b; // new way
0 references
public int OldSum(int a, int b) // old way
{
    return a + b;
}
```

OldSum() déclare une méthode qui prend deux paramètres entiers et qui effectue un traitement simple, ici une addition.

NewSum() déclare exactement la même chose en une seule ligne en remplaçant le corps de la méthode par une fonction Lambda. Le gain de lisibilité et de productivité est évident.

Mais ce n'est pas tout ! On peut aussi utiliser l'astuce syntaxique pour des méthodes qui retournent Void ou des Tasks. Dans ce cas l'expression doit être une instruction simple qui généralement ne retourne rien non plus :

```
0 references
public void NewLog(string str) => Console.WriteLine(str); // new way
0 references
public void OldLog(string str) // old way
{
    Console.WriteLine(str);
}
```

OldLog() permet d'écrire une chaîne passée en argument sur la console (ou un système de log quelconque).

NewLog() fait de même mais en une seule ligne bien plus claire et débarrassée des "{ }" directement accessibles sur une clavier américain mais un peu enquiquinants à obtenir sur un clavier Azerty. Pour les français le gain est donc encore plus grand !

Conclusion

Ces nouvelles constructions de C# 6.0 apportent un vrai bénéfice en termes de lisibilité du code avec tout ce que cela implique de productivité, maintenabilité et le reste. Un langage toujours moderne même dix ans après.

C# 6 le mot clé nameof

C# 6.0 continue de nous surprendre… aujourd'hui je vous présente le nouveau mot clé "nameof"…

Les littéraux

Les chaînes de caractères sont l'ennemi d'un langage fortement typé comme C#, et pourtant en de nombreuses occasions il faut passer des chaînes qui, par force, ne sont pas contrôlées. Faute d'orthographe, étourderie, refactoring, etc, tout cela peut engendrer des situations à risque où le fameux littéral ne correspond plus du tout à un nom réel dans l'application. Bogue sournois s'il en est par exemple dans un gestionnaire de `PropertyChanged`.

A l'ancienne

Même si pour ce cas particulier il existe depuis quelques temps une solution élégante permettant de récupérer le nom de la propriété appelante (l'attribut `CallerMemberName`) il n'est pas rare qu'on souhaite activer INPC depuis une autre propriété ou bien qu'on ait tout simplement besoin du nom d'une propriété ou d'une classe dans plein d'autres circonstances (sous XAML l'usage des littéraux est parfois gênant pour une programmation sure).

Donc dans un code INPC classique on trouvera quelque chose comme ça :

```
private Personne personne = new Personne();

public Pesonne Directeur
{
    get { return personne; }
    set {
        personne = value;<pre class="brush: csharp; auto-links: true;
collapse: false; first-line: 1; gutter: true; html-script: false; light:
true; ruler: false; smart-tabs: true; tab-size: 4; toolbar: true;">
doChanged("Directeur");
         }
}

privade void doChanged(string propertyName)
{ ... INPC ... }</pre>
```

Ce qui gêne ici c'est bien entendu l'utilisation de "Directeur" sous la forme d'un littéral alors qu'il s'agit du nom d'une propriété.

Comme je le disais, dans ce cas précis l'attribut [CallerMemberName] qu'on placerait devant la définition de la chaîne dan la méthode "doChanged" règlerait avec élégance le problème.

Mais si nous supposons une propriété "NomComplet" qui doit être mise à jour à chaque fois que les propriétés "Nom " et "Prénom" seront modifiées on retombe sur le même problème. En effet, si le "doChanged" pour "Nom" ou "Prénom" peut être automatique, il n'en va pas de même pour "NomComplet", et il faudra dans les deux

propriétés principales écrire un “doChanged(“NomComplet”)” si on veut que cette dernière soit rafraichie aussi…

Alors comment se débrouiller pour supprimer définitivement le passage d’un littéral qui pourtant représente un nom déclaré dans le code ?

nameof

Les idées les plus simples sont souvent les meilleures… Donc C# 6.0 ajoute le mot clé “nameof” qui fait exactement ce qu’on attend de lui, c’est à dire remplacer un nom ayant un sens dans le code par une chaîne de caractères qu’on peut ensuite utiliser sans prendre le risque qu’une fracture se créée entre cette chaîne et le nom interne à l’application.

Ce qui donne ceci :

```
doChanged(nameof(Directeur));



//Second exemple:



.... doChange(nameof(NomComplet)); ...
```

Plus de littéral bien que des chaînes sont toujours utilisées. L’une de mes hantises prend fin !

Conclusion

Encore une simplification qui ici ne permet pas seulement de gagner du temps ou d’alléger le code mais bien d’éviter de nombreux bogues généralement difficile à trouver. Merci C# 6.0 !

C# 6 – Concaténation de chaînes

Dans la série des petites nouveautés sympathiques de C# 6, voyons de plus prêt celles qui concernent la concaténation de chaînes de caractères…

Concaténer

La concaténation est une opération simple et largement utilisée. Elle peut s’opérer de trois façons :

1. Par l’addition (+) comme s’il s’agissait de nombres
2. Par la classe StringBuilder plus rapide mais plus verbeuse
3. Par le string.Format et ces différentes options

Ces trois méthodes ont toutes leurs avantages et leurs inconvénients, je ne reviendrai pas la dessus ici.

Concaténer façon C# 6

C# 6.0 nous offre une façon encore plus rapide et plus efficace pour concaténer des chaînes de caractères. Plus exactement cette nouvelle façon de faire remplacera souvent à merveille l'addition ou le string.Format mais n'est pas conçue pour concurrencer StringBuilder (dans le cas de nombreuses manipulations à effectuer cette classe reste la plus performante).

L'astuce consiste à insérer directement les champs dans le texte à traiter ! Et on peut même fixer des règles d'alignement, des spécificateurs de format et plus encore des conditions !

Cela va plus vite à taper et éviter un code trop lourd donc permet un debug plus aisé. Voici à quoi cela ressemble :

```
Console.WriteLine("Nom complet : \{personne.Prénom] \{personne.Nom}\nAge : {personne.age}");
```

Ici on formate de la façon la plus directe le nom complet d'une instance "personne" supposée pour l'exemple contenir les informations d'un individu (salarié, élève, membre d'un groupe…) sur une seule ligne puis on affiche l'âge de cette personne sur la ligne suivante. Tout l'intérêt est dans la nouvelle syntaxe qui permet d'utiliser l'anti-slash suivi d'une expression entre accolades américaines sera interprétée au runtime.

Mais on peut aller plus loin avec du padding pour cadrer le texte et l'indication de chaînes de format pour les nombres par exemple :

```
Console.WriteLine("Nom : \{personne.Nom, 15}\nCrédit : \{personne.Crédit : C2}");
```

Le nom de la personne est cadrée à gauche dans un espace de 15 caractères comblé si nécessaire par des caractères 32, un saut de ligne est effectué et le crédit du compte de la personne est formaté comme un numérique utilisant la chaîne de format "C2".

C'est vraiment très pratique. Mais allons encore un cran plus loin avec du code conditionnel :

```
Console.WriteLine(<br> "Nom: \{p.Nom, 15}\nCrédit: \{p.Crédit : C2} \{(p.Crédit<p.Débit
? "Découvert!" : "Ok}");
```

Pour la mise en page du blog cela devient difficile puisque le jeu consiste à tout mettre dans une seule chaîne de caractères difficile à couper en deux ou trois…

Dans cet exemple on ajoute au précédent l'affichage d'une mention "Découvert!" ou "Ok" selon que le compte possède un crédit inférieur ou non à son crédit.

Conclusion

Voici encore une simplification qui évitera des lignes de code juste pour cadrer une chaîne, ajouter un mot dans certaines conditions ou cadrer un numérique correctement…

C# 6 – Initialisation des propriétés automatiques

C# 6.0 propose son lot de nouveautés qui rendent la programmation encore plus rapide, plus fiable. Parmi celles-ci on trouve l'initialisation des propriétés automatique.

Les propriétés automatiques

Depuis quelques versions déjà C# permet de déclarer des propriétés dites automatiques c'est à dire sans avoir à déclarer un "backing field", un champ sous-jacent qui contient réellement la valeur. Le compilateur s'en charge.

Une telle déclaration ressemble à :

```
public int Duration {get; set;}
```

C'est simple et efficace. Oui mais…

L'initialisation d'une propriété automatique

Comparativement à une propriété classique, la propriété automatique ne possède pas de champ explicite qu'on peut initialiser facilement.

C'est bête parce que cela oblige à initialiser la valeur dans le constructeur de la classe le plus souvent. C'est un code fastidieux qu'il ne faut pas oublier et qui, il est vrai, fait perdre de son charme aux propriétés automatiques.

C# 6 à la rescousse

C'est tellement bête, tellement simple qu'on se demande pourquoi il aura fallu attendre la 6ème version de C# pour voir apparaitre cette syntaxe !

Désormais pour initialiser une propriété automatique nul besoin d'ajouter du code dans le constructeur de la classe ou de déclarer un backing field. Il suffit de placer l'initialisation à la suite logique de la déclaration de la propriété :

```
public int Duration {get; set;] = 42;
```

```
public string FirstName {get; private set;} = "Frank";

public DateTime StartTime {get; set;} = DateTime.Now;
```

Etc…

Conclusion

Une astuce bien pratique dont on ne pourra bientôt plus se passer !

C# est un langage qui a connu de grandes évolutions majeures comme LINQ, il ne peut guère aller beaucoup plus loin mais malgré tout la nouvelle mouture apporte son lot d'innovations, qu'on aimera, adorera, ou détestera !

Les nouveautés de C# 7

Microsoft ayant connu des difficultés avec son Java maison a demandé à Anders de sauver la mise. Mélangeant ce qu'il avait fait avec Delphi (comme la notion de propriété) chez Borland, un peu du Java MS mis au placard et un soupçon de C++ il a pondu C#. On connait généralement tous l'histoire (mais ce n'est pas certain, ça remonte un peu !). C# aurait ainsi pu n'être qu'une tentative désespérée de sauver la face sans grand avenir. Mais il s'avère que ce langage a connu un franc succès… Et que le talent de Anders Hejlsberg au nom aussi imprononçable qu'il s'écrit a permis des évolutions incroyables dont la plus marquante pour moi restera pour toujours Linq (l'introduction du fonctionnel dans C#) dont je ne saurai plus me passer pour écrire du code.

Lentement mais surement, C# a franchi les outrages du temps (même s'il reste un langage assez récent comparé à C++ ou même Java) sans prendre une ride, sans devenir ringard ou pire, dépassé.

La 7ème version de ce langage est là. Il y a fort à parier que jamais plus nous ne connaitrons de bonds aussi spectaculaires que Linq, mais tout de même, chaque version a amené ses petits perfectionnements qui au final en font un grand langage toujours aussi agréable à utiliser. La version 7 n'échappe pas à la règle même si comme à chaque fois désormais certaines "améliorations" feront hérisser le poil de certains.

Mais voyons de quoi il s'agit.

Les variables OUT

Un petit truc qui me plait bien car franchement cela m'agaçait… Au lieu d'écrire :

```
1. string result;
2. if (dictionary.TryGetValue(key, out result))
```

On peut écrire :

```
1. if (dictionary.TryGetValue(key, out string result))
```

C'est quand même moins lourd. Cette variable a déclarer avant m'énervait. Soulagement donc mais pas une révolution c'est évident.

Tests étendus "pattern matching"

Voici quelque chose qui va permettre pour les uns d'écrire un code plus concis et plus lisible et pour d'autres qui signifie toujours plus d'obfuscation qui rend le langage de moins en moins lisible notamment pour les débutants. Les deux positions sont vraies, sans que cela soit paradoxal.

C# 7 amène dans son sac de nouveautés la notion de "patterns". Le *pattern matching* se concrétise par des éléments syntaxiques qui peuvent tester si une valeur a une certaine « forme », et extraire des informations à partir de la valeur quand cela est nécessaire.

Pour l'instant les motifs ou patterns que C# 7.0 propose sont de type Constante (tester si C est égal à la valeur testée), des motifs de Type sous la forme T x (on teste si une valeur est de type T et dans ce cas elle est placée dans la valeur x), et les motifs de variables sous la forme Var x (le test passe toujours et la valeur est transférée dans x).

Ce n'est bien sûr qu'un début et cela évoluera avec d'autres types de motifs.

En pratique il faut un peu de code pour voir de quoi il s'agit je l'admets.

Expression "is" avec pattern matching

Prenons le cas de "is", lorsqu'on y ajoute le pattern matching cela devient encore plus puissant :

```
1. public void PrintStars(object o)
2. {
3.     if (o is null) return;      // constant pattern "null"
4.     if (!(o is int i)) return; // type pattern "int i"
5.     WriteLine(new string('*', i));
6. }
```

Comme on le voit la première ligne est assez classique et correspond au "constant pattern "null"", au motif de type Constante dont la valeur de test est "null".

La seconde ligne devient plus intéressante et moins classique. Le "is" est utilisé avec le "type pattern", quand on teste un type tout en transférant dans une valeur si le test passe. Ici on note le "i" placé derrière "int". Sans ce "i" le test ressemble à du C# 6 ou précédent. Avec le "i" on demande en pratique de créer une variable "i" de type "int" si le "is" passe… Du coup la dernière ligne de la méthode n'est pas exécutée si le "is" ne passe pas (test assez classique de la ligne précédente) mais si elle est exécutée on pourra utiliser la variable entière "i" qui aura été créée !

C'est simple mais très efficace.

Le Switch et le pattern matching

Autre possibilité d'utilisation du pattern matching, avec le switch et le case.

Il devient d'abord possible de faire un switch sur n'importe quel type et plus seulement sur un type primaire, ce qui est génial, et les patterns peuvent être utilisées dans les "case", ce qui est fantastique, les "case" peuvent avoir des conditions additionnelles, ce qui est la cerise sur le gâteau !

```
1. switch(shape)
2. {
3.     case Circle c:
4.         WriteLine($"circle with radius {c.Radius}");
5.         break;
6.     case Rectangle s when (s.Length == s.Height):
7.         WriteLine($"{s.Length} x {s.Height} square");
8.         break;
9.       case Rectangle r:
10.             WriteLine($"{r.Length} x {r.Height} rectangle");
11.             break;
12.      default:
13.             WriteLine("<unknown shape>");
14.             break;
15.      case null:
16.             throw new ArgumentNullException(nameof(shape));
17. }
```

Attention embrouille !

Avec l'introduction du pattern matching l'ordre des cas dans le swtich devient désormais important !

Le premier test qui matche fera exécuter le cas. Par exemple on note que le cas du carré (longueur égale à hauteur) est testé avant celui du rectangle… Inverser le test donnera un résultat différent.

Autre détail à noter, le cas "default" est toujours évalué en dernier. Même si dans le code ci-dessus le test à "null" semble suivre le test par défaut, ce dernier sera toujours exécuté à la fin (donc après le test à null ici, ce qui est ainsi différent de l'ordre d'écriture et peut troubler les développeurs peu attentifs !).

Plus étonnant le cas "null" ne sera en fait jamais exécuté c'est un code mort à supprimé car le switch version pattern matching fonctionne comme le "is", c'est à dire

que le “null” n’est jamais testé comme ok. Si on veut tester la valeur nulle il faut laisser cela au cas “default” ou le faire autrement…

Comme je le disais il y aura les gars qui seront pour et les gars qui seront contre ! (et ça marche pareil pour les nanas, je veux rassurer les féministes !).

Les tuples

Les tuples ça existe déjà et il y a déjà un clivage entre les deux camps. Personnellement je trouve ça très pratique dans l’idée mais à chaque fois que j’ai écrit un code qui les utilisait j’ai tout refait en déclarant un type car je trouvais imbuvable le coup des “item1”, “item2” etc qui n’ont aucun sens et font donc perdre celui du code (avec à la clé plus d’erreurs possibles).

Mais c’est là que C# 7 vient ajouter le petit truc en plus qui rend tout cela plus utilisable. Avec les types de tuple et les littéraux de tuple.

Avec cet ajout on trouve donc désormais des méthodes capables de retourner plusieurs valeurs typées et nommées. Plus d’obligation de créer une classe juste pour rendre lisible une méthode qui retourne plusieurs valeurs, c’est un truc que j’adore et qui s’ajoute aux autres améliorations des tuples :

```
1. (string, string, string) LookupName(long id) // tuple return type
2. {
3.     ... // retrieve first, middle and last from data storage
4.     return (first, middle, last); // tuple literal
5. }
```

Ici le “return” passe simplement trois valeurs entre parenthèses. C’est possible car la méthode prend comme type de retour (string, string, string) ce qui est automatiquement traduit en tuple sans avoir à l’écrire.

Mais ce n’est toujours qu’un tuple “normal” qui est créé. Ainsi on utilisera la méthode comme si elle était écrite avec Tuple<string,string,string>, c’est à dire avec les “Item1”, “Item2”, ceux qui me chagrinent justement…

```
1. var names = LookupName(id);
2. WriteLine($"found {names.Item1} {names.Item3}.");
```

C’est mieux au niveau de la déclaration de la méthode mais c’est toujours aussi fouillis au niveau de l’utilisation. Mais heureusement C# 7 nous permet d’aller un cran plus loin en écrivant la déclaration de la méthode ainsi :

```
1. (string first, string middle, string last) LookupName(long id)…
```

Maintenant chaque élément du tuple en plus d'être typé, en plus de ne pas demander l'indication Tuple<…> devient un champ nommé !

L'utilisation de la méthode devient alors véritablement clair :

```
1. var names = LookupName(id);
2. WriteLine($"found {names.first} {names.last}.");
```

Le type Tuple<…>, sa déclaration habituelle, son côté figé, tout cela disparait. C'est en réalité un type Dynamic qui est créé, mais de façon à ce que cela soit plus performant que les dynamiques habituels. Ces contournements font que pour un novice qui apprendra C# 7 il sera plus facile de lui dire que les méthodes peuvent retourner plusieurs valeurs sans même parler de la classe Tuple.

Cela va même plus loin avec la "**déconstruction**", c'est à dire la segmentation des éléments d'un Tuple dans plusieurs variables :

```
1. (string first, string middle, string last) = LookupName(id1);
                                        // deconstructing declaration
2. WriteLine($"found {first} {last}.");
```

L'appelant récupère ici les trois valeurs du tuple dans trois variables, peu importe les noms des champs du tuple.

On notera qu'on peut utiliser "var" au lieu de nommer le type de chaque variable. Il est bien sûr possible de déconstruire un tuple dans des variables existantes :

```
1. (first, middle, last) = LookupName(id2);
```

On suppose ici que first, middle et last sont déjà déclarés plus haut dans le code. Aucune nouvelle variable ne sera créée par la déconstruction.

Déconstruction partout !

Ce que je viens de dire pour les tuples marche en réalité pour TOUT type. On peut désormais *déconstruire* n'importe quel type, à une condition toutefois : que ce type contienne une méthode spéciale qui s'appelle forcément **Deconstruct** :

```
1. public void Deconstruct(out T1 x1, ..., out Tn xn) { ... }
```

Pascal ou C# ? : Les fonctions locales

Et oui tout vient à temps pour qui sait attendre… J'ai quitté il y a longtemps le Pascal Objet pour C# mais parfois encore malgré les années des choses me manquent cruellement, comme les sous-fonctions.

Avec l'arrivée des expressions Lambda j'avais trouvé un moyen de déclarer des *delagate* dans les méthodes pour créer des sous-fonctions à la Pascal. Ca marche plutôt bien.

Je sais, là aussi il y a le camp de ceux qui adorent et de ceux qui détestent. Un Pascalien reste un Pascalien je n'y peux rien, moi j'adore.

Mais ce n'était qu'une astuce et seulement depuis l'arrivée des Lambda… Grâce à C# 7 on retrouve ce que Anders avait déjà fait dans Turbo Pascal et Delphi… Tant d'années pour y revenir… mais c'est là, ça s'écrit comme une sous-fonction Pascal sans bricoler avec les lambda :

```
1. public int Fibonacci(int x)
2. {
3.     if (x < 0) throw new ArgumentException("Less negativity please!",
    nameof(x));
4.     return Fib(x).current;
5.
6.     (int current, int previous) Fib(int i)
7.     {
8.         if (i == 0) return (1, 0);
9.         var (p, pp) = Fib(i - 1);
10.             return (p + pp, p);
11.         }
12.     }
```

L'exemple (puisé de la documentation MS) est un peu tiré par les cheveux car la vraie fonction est celle qui est déclarée à l'intérieur, elle ne devrait pas être là mais à part. Mais ça se discute… La méthode qui déclare la sous-fonction (en C#7 on dit méthode locale) n'est qu'une enveloppe qui teste si la valeur est négative, c'est totalement artificiel et pourrait être contenu dans la fonction elle-même. Mais les exemples sont ce qu'ils sont, souvent réducteurs… Et tout le monde s'enflamme sur l'exemple trop simple qui ne sert pas la cause de ce qui est montré !

En tout cas vous avez compris le principe, et s'il est bien utilisé je suis totalement pour. Pas seulement par nostalgie de pascalien mais tout simplement parce qu'il arrive souvent qu'une méthode soit plus claire avec des sous-fonctions. Certains diront que si le code d'une méthode devient trop "compliqué" (ce qui est un

jugement 'moral' plus que technique, aucune norme n'existant) il faut éclater la méthode en plusieurs méthodes.

Parfois c'est en effet ce qu'il faut faire.

Mais parfois pas du tout, cela pollue la classe avec des tas de petites méthodes qui en réalité ne servent à rien en dehors des méthodes où elles auraient dû être déclarées.

En Pascal il existe par exemple la notion de variable globale, déclarée hors de toute classe (qui n'existent pas en Pascal standard), ce que Delphi qui est objet avait repris mais avec une astuce, ces variables étaient de toute façon englobées en réalité dans une classe non visible. Mais peu importe, tout le monde considère, moi le premier, que des variables globales comme le fait le Pascal non objet est une catastrophe aujourd'hui. C'est plus clair si les variables sont contenues dans les classes où elles sont utilisées. Et les classes dans des namespaces… Vous serez certainement de cet avis.

Si les sous-fonctions vous gênent dîtes-vous alors qu'il ne s'agit jamais que d'attribuer un namespace local à une méthode privée (ce namespace étant la méthode mère), et ce pour les mêmes bonnes raisons qu'on n'utilise plus de variables "globales" et qu'on place les variables dans des classes et les classes dans des namespaces…

Les littéraux le binaire et le underscore

Certainement moins clivant, les littéraux sont autorisés à utiliser le signe souligné (le "tiret du 8") pour séparer les chiffres ce qui permet de les rendre plus lisibles :

```
1. var d = 123_456;
2. var x = 0xAB_CD_EF;
3. var b = 0b1010_1011_1100_1101_1110_1111;
```

Au passage on découvre les littéraux binaires… déclarés avec le préfixe "0b". Le souligné prend ici tout son intérêt pour séparer les octets.

Le retour d'une Référence

Alors là c'est chaud, très chaud. Niveau polémique je suis certain que ça va bouillir avec des équipes qui interdiront carrément ce genre de trucs comme c'est le cas pour la surcharge des opérateurs. On replonge dans les heures sombres du développement avec les histoires de pointeurs retournés à la place de valeurs ce qui fichait un beau bazar et était l'un des points rendant les programmes en C si prompts à planter !

Alors certes il ne s'agit pas de manipuler ici de vrais pointeurs mais c'est tout comme techniquement, une référence c'est un pointeur. Et ça possède les mêmes effets pervers. Effets qui est justement ce qui est recherché ici, ce qui est un peu vicieux je trouve.

Il y a néanmoins des cas où cette nouveauté sera certainement très appréciée, comme toutes les features un peu discutables.

Concrètement ? Voici un bout de code :

```
1. public ref int Find(int number, int[] numbers)
2. {
3.     for (int i = 0; i < numbers.Length; i++)
4.     {
5.         if (numbers[i] == number)
6.         {
7.             return ref numbers[i];
               // return the storage location, not the value
8.         }
9.     }
10.          throw new IndexOutOfRangeException($"{nameof(number)} not f
   ound");
11.      }
12.
13.      int[] array = { 1, 15, -39, 0, 7, 14, -12 };
14.      ref int place = ref Find(7, array);
        // aliases 7's place in the array
15.      place = 9; // replaces 7 with 9 in the array
16.      WriteLine(array[4]); // prints 9
```

La méthode "Find" ne retourne plus une valeur mais une *référence*. De fait son appel est farci de "ref" pour le faire comprendre. Mais lorsqu'on modifie la variable "place" on modifie aussi la valeur dans l'array originale ! Attention danger donc…

Les corps d'expression

C'est un ajour de C# 6 que j'aime bien car cela simplifie le code pour le développeur expérimenté sans le rendre plus difficile à comprendre pour le débutant (et un code lisible par tous dans une équipe qui peut recevoir des novices est essentiel). On remplace une méthode par une expression. Il existait toutefois des limitations, C# 7 les fait sauter et on peut utiliser les expressions pour ainsi dire partout même dans les constructeurs ou destructeurs :

```
1. class Person
```

```
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int, string>();
    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors
    ~Person() => names.TryRemove(id, out *);              // destructors
    public string Name
    {
        get => names[id];                                 // getters
        set => names[id] = value;                         // setters
    }
}
```

Les expressions peuvent être utilisées aussi pour lever des exceptions, là encore cela simplifie beaucoup le code :

```
class Person
{
    public string Name { get; }
    public Person(string name) => Name = name ?? throw new ArgumentNullException(name);
    public string GetFirstName()
    {
        var parts = Name.Split(" ");
        return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");
    }
    public string GetLastName() => throw new NotImplementedException();
}
```

D'autres choses

Il existe d'autres améliorations notamment dans la généralisation des types de retour async. Je reviendrai certainement dans le futur sur ces nouveautés.

Conclusion

C# évolue, de plus en plus à la marge car il a tellement évolué qu'on voit mal ce qu'on pourrait encore lui ajouter, en tout cas de très significatif et sans le changer en profondeur. Pour les changements radicaux de paradigme il y a déjà F# par exemple. Inutile de trop bricoler C#.

Il n'empêche, C# n'est pas statique, il s'améliore sans cesse. Des features introduites dans une version sont améliorées dans une suivante.

Et c'est bien de travailler avec un langage qui se modernise tout le temps. Certes apprendre C# aujourd'hui est bien plus difficile qu'à l'époque de la version 1.0. C'est le prix à payer.

Mais grâce à cela C# reste un langage performant, d'une puissance incroyable, et toujours très agréable à utiliser au quotidien, et ça c'est très important !

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 subtil 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

    }
}
```

Asynchronisme & Parallélisme

Appels synchrones de services. Est-ce possible ou faut-il penser "autrement" ?

> ***Ces premiers articles sur l'asynchronisme ont déjà quelques années, mais j'ai décidé de les laisser dans cette nouvelle édition car de proche en proche ils permettent de mieux saisir comment l'édifice asynchrone s'est construit, pierre par pierre, et comment nous en sommes arrivés aux solutions proposées par C# 6.***
>
> ***Le lecteur passera outre quelques anachronismes ou le fait que j'évoque Silverlight ici. La problématique est ancienne mais les mécanismes étudiés permettent de comprendre les nouvelles façons de faire ! Lisez cette série à la suite puis abordez celle sur Task à la fin. Elle ne vous semblera que plus abordable !***

Silverlight ne gère que des appels asynchrones aux Ria Services et autres communications WCF [NdA - début de l'asynchronisme partout !]. Le Thread de l'UI ne doit jamais être bloqué assurant la fluidité des applications [NdA – début du Reactive Design !]. 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 ?

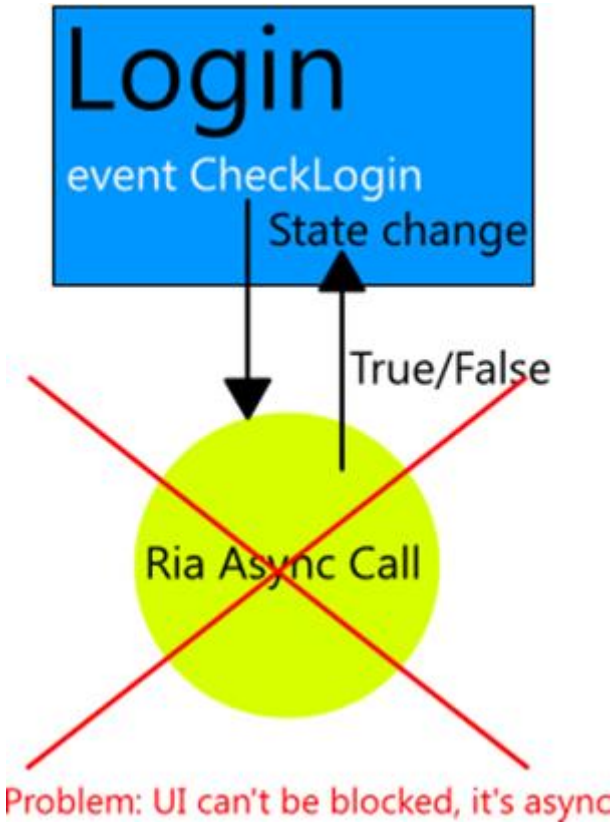
J'y viens... C'est vrai que le décor, pourtant simple, n'est pas si évident que cela à "raconter". Mais je pense que vous avez saisi l'affaire.

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.

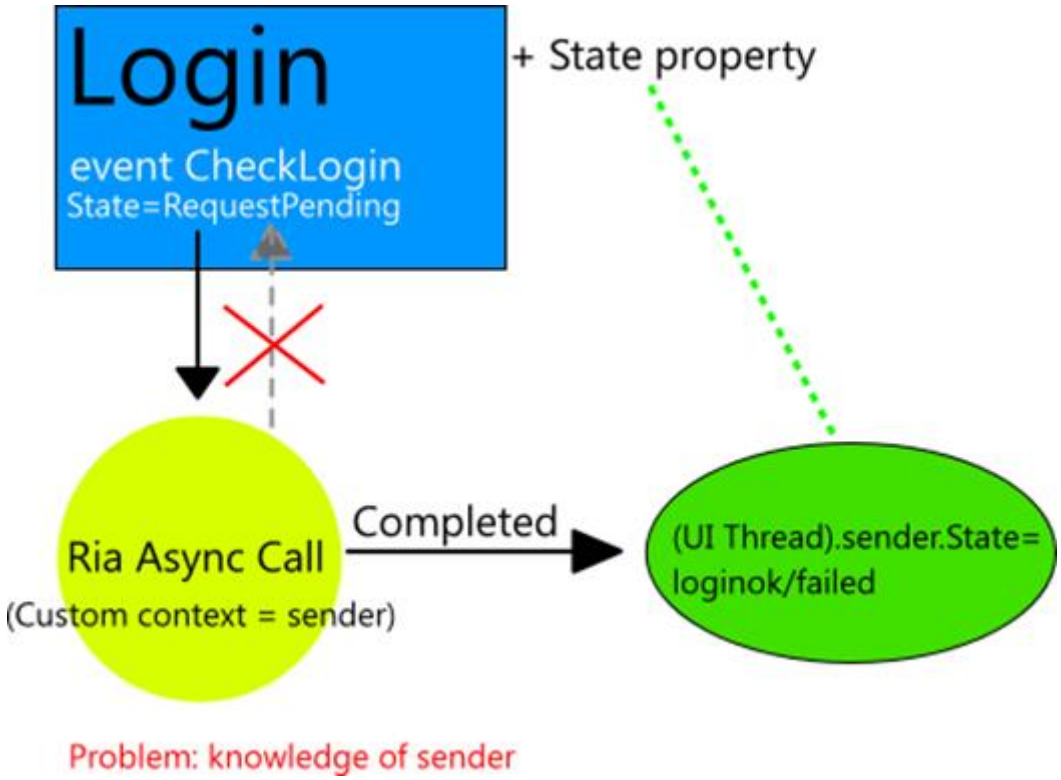
Alors ?

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...

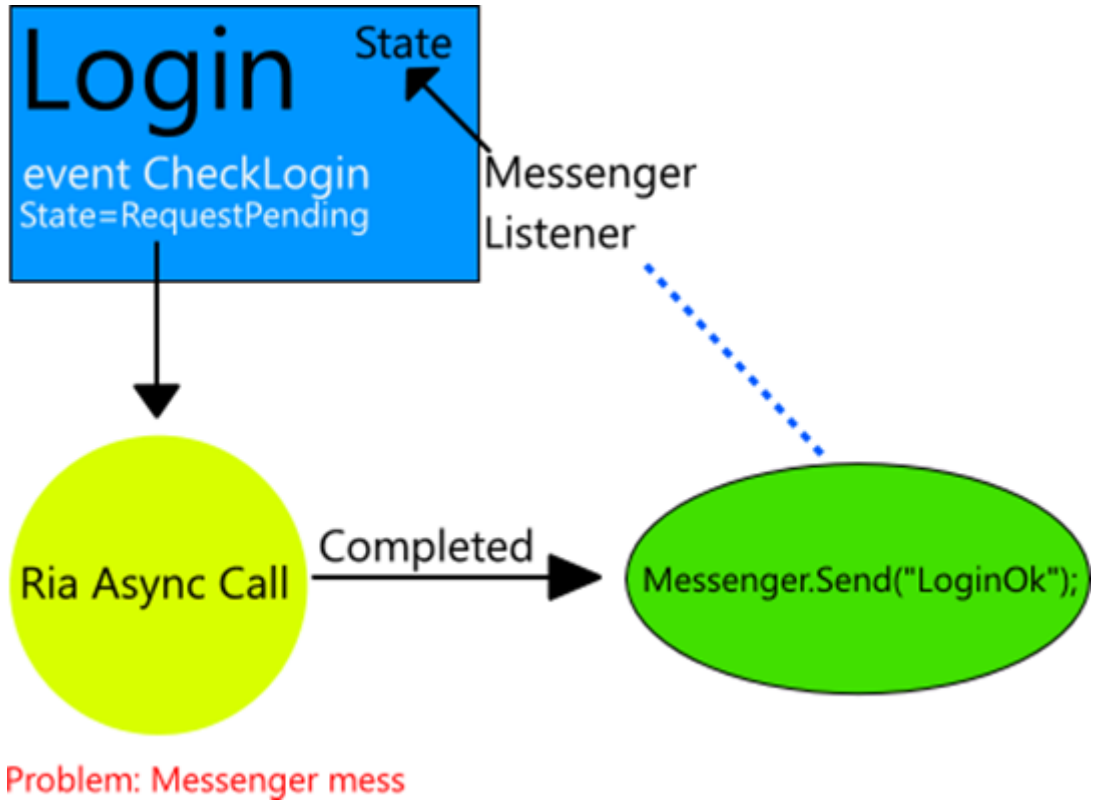
L’approche par messagerie

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.

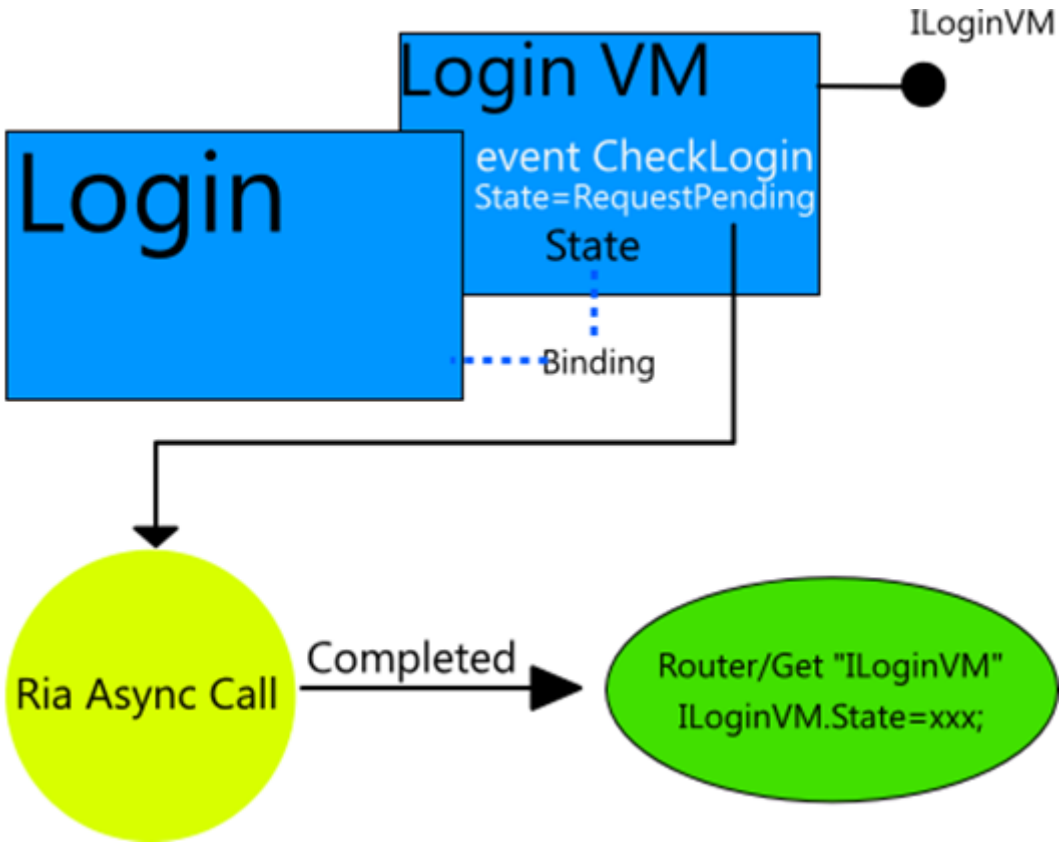
Mais la messagerie. Brrr. Ca me glace le sang. D'autant plus que le Contrôle de Login devra répondre à un message précis qui sera codé en dur. Il faudra bien documenté tout cela pour que les applications qui s'en serviront sachent quel message il faut utiliser, s'il s'agit d'une notification en chaine 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 la complète, les Reactive Extensions !

Parallélisme

J'avais abordé le sujet dans mon billet "La bombe Parallèle ! PLINQ, et PCP Parallel Computing Platform" il y a 3 ans presque jour pour jour… Pour ceux qui n'ont pas suivi je rappellerai donc juste quelques points essentiels développés dans ce billet :

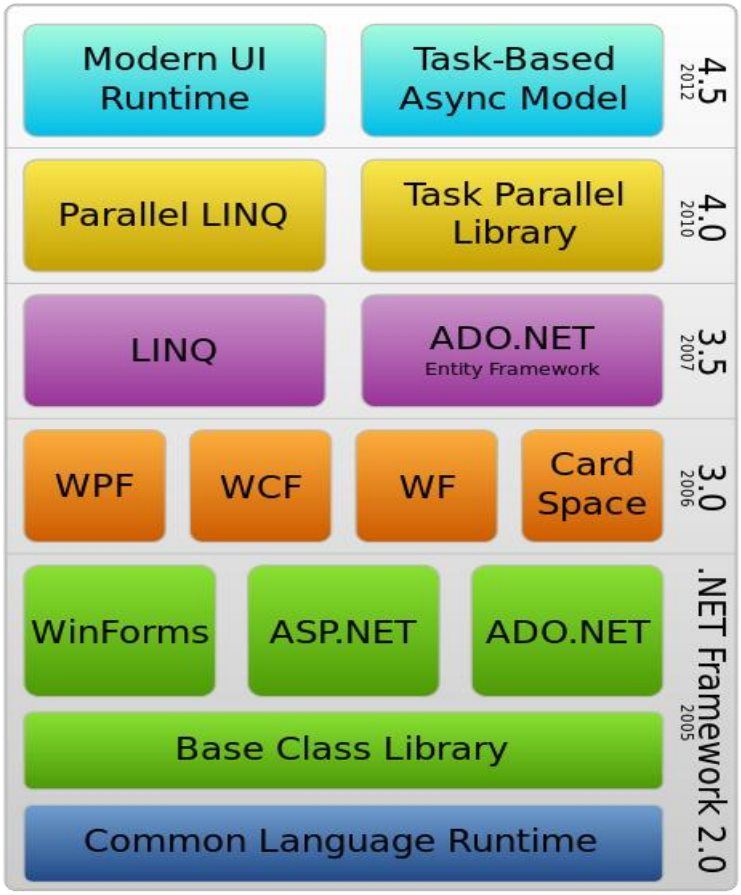
- La loi de Moore reste valable mais l'équivalence gain de puissance = fréquence du processeur plus élevée, elle, est morte…
- Pour continuer à faire évoluer la puissance des ordinateurs les fondeurs optimisent les puces mais surtout *multiplient les cœurs* au sein d'une même puce.
- Il y a deux ans, 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 parti 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 côté 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 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 seront intégrées au Framework mais elles évoluent, peut-être un peu comme un laboratoire expérimental indépendant. 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 compilateur 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…

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 parti 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 4.6 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 tort 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 trêve de mots, le plus simple c'est de jouer avec ce composant pour se rendre compte de son utilité. Pour faciliter la... Tâche...
Un peu de douceur multitâche dans ce monde de multicoeurs...

Les dangers de l'incrémentation / décrémentation en multithreading

Parfois il est bon de parler de petites choses dans un billet court... C'est le cas aujourd'hui pour les opérateurs d'incrémentation et de décrémentation qui posent de sérieux problèmes en mode multitâche...

Parallélisme, multithreading, asynchronisme...
Toutes ces technologies ou techniques ont pour finalité de faire tourner plusieurs bouts de code en même temps. Les machines modernes, même mobiles, ayant plusieurs cœurs il s'agit vraiment de faire exécuter du code par plusieurs "ordinateurs" physiquement différents à la fois, même si tout cela est rangé sagement dans un même boitier, dans une même puce.

Dans mon dernier billet je mettais l'accent sur l'utilisation correcte de async/await et la façon d'optimiser le code dans un contexte asynchrone.

Aujourd'hui je voudrais vous alerter du danger de l'utilisation de "++" et "--" dans de tels contextes multitâches.

++ et -- sont dans un bateau multitâche...
Non, la suite n'est pas de savoir lequel des deux tombe à l'eau et lequel reste à bord. Ils coulent tous les deux, tout simplement !

L'implémentation de ces opérateurs n'est pas "thread safe", c'est à dire que leur utilisation et surtout leur effet dans un contexte multithreadé n'est pas garanti.

Vous allez vite me croire en quelques lignes de C# sous LinqPad sous la forme d'un GIF animé (à voir sur Dot.Blog !).

Ce code est vraiment très simple : une méthode `Main()` en mode console qui utilise deux variables statiques, deux compteurs de type *integer*,
appelés `counter1` et `counter2` en toute logique.

Pour montrer les dangers de l'incrémentation (ce serait la même chose pour la décrémentation) dans un contexte multitâches
j'utilise `System.Threading.Tasks` et `Parallel.For` qui permet de paralléliser des traitements fournis dans une boucle de type *For*.

J'ai fixé un maximum de 1000 itérations pour la boucle parallélisée. Comme le contenu de la boucle ne fait pas grand chose j'ai ajouté un délai de 12 millisecondes.

Cela ralenti un peu les choses et permet de voir l'effet pervers que je cherche à mettre en évidence.

A chaque passage dans la boucle les deux compteurs sont incrémentés par "1". Le premier utilise l'opérateur d'incrémentation "++", l'autre utilise `Interlocked.Increment()`.

En fin de travail l'application récapitule le nombre d'itérations effectuées ainsi que la valeur des compteurs.

Et Ô désolante réalité… Si le compteur 2 affiche avec fierté "1000" qui correspond à nos attentes, le compteur 1 semble avoir connu quelques "trous" et montre une valeur de 898 totalement imprévue…

Conclusion

La conclusion s'impose d'elle-même : les opérateurs d'incrémentation et de décrémentation ne sont pas fiables en mode multitâche. Et comme ce mode devient de plus en plus fréquent en programmation moderne il y a beaucoup à parier que de sacrés bogues bien difficiles à trouver ne manqueront pas de causer bien des tracas à de pauvres développeurs mal informés…

Ce n'est pas le cas des lecteurs de Dot.Blog, bien heureusement !

De la bonne utilisation de Async/Await en C#

Async et Await simplifient l'écriture des applications qui doivent rester fluides et réactives. Mais cela suffit-il à rendre les applications multitâches ? Pas si simple… C'est ce que nous allons voir…

Async et Await

Pour ceux qui éventuellement auraient loupé un épisode important des aventures de C#, sa vie, son œuvre en 5 tomes, voici un bref résumé de ce que sont les mots clés *Async* et *Await* et ce à quoi ils servent.

C'est une question de méthode…

De nombreuses méthodes ne retournent pas immédiatement le trait, elles peuvent effectuer des tâches assez longues, comme accéder à des ressources extérieures (Web, réseau, appareil de mesure externe ou interne tel le compas, le GPS, l'accéléromètre…).

Toute méthode exécutée sur le thread principal bloque logiquement ce dernier or il est utilisé par l'OS pour gérer l'interface.

Toute méthode longue fige donc l'UI et confère à l'application une certaine rigidité d'un autre âge lui interdisant d'être réactive et fluide, deux points essentiels aujourd'hui.

Les threads ne sont pas des tâches

On pourrait se dire que toute méthode un peu longue à exécuter devrait être exécutée au sein d'un thread et que cette technique existe depuis longtemps. C'est tout à fait exact. Mais il faut convenir d'une part que cela rend l'écriture du code très lourde si toutes les méthodes un peu longues doivent ainsi être déplacées dans des threads – incluant les API de la plateforme – et que, d'autre part, il est assez difficile de contrôler avec précision plusieurs threads, leur début, leur fin, la gestion des erreurs, etc.

De fait, les threads ne sont pas des *tasks*…

async et await

Pour permettre à une méthode longue de retourner le trait immédiatement (ou en tout cas au plus vite), elle peut exécuter son code en créant un tâche (Task). Le mot clé await permet d'attendre la fin d'une telle tâche. Asyncpermet de marquer une méthode qui fera usage de await.

Fixons les choses

- Une méthode async ne peut retourner que void ou une Task.
- Une Task peut ne retourner aucune valeur (autre que son état).
- Une Task<montype> retourne des données de type "montype".

- La méthode Main ne peut pas être async. La raison est simple, async permet avec await de retourner le trait immédiatement, or sortir de Main cela revient à mettre fin à l'application…
- Elle ne peut donc pas utiliser await.
- Si elle en a besoin elle peut lancer une méthode async en utilisant directement la classe Task
- Elle peut aussi appeler une méthode qui elle sera async et pourra utiliser await.

On notera au passage un débat assez nourri sur l'utilisation de async avec une méthode retournant void. Le Web est plein de discussions savantes (ou simplement rasoirs, il y en a !) sur ce thème. En gros async void est de type "fire and forget", expression militaire assez claire. De fait les seules méthodes pouvant être de type

async void (donc qu'on ne peut pas attendre avec await) son des évènements (type Click par exemple). Dans tous les autres cas même si la méthode ne retourne rien elle sera de type async Task et non async void.

Une méthode marquée async sera exécutée de *façon* synchrone si elle ne contient pas le mot clé await au moins une fois. C'est finalement assez évident.

Techniquement async et await ne sont qu'un *syntactic sugar*, c'est à dire un moyen artificiel de simplifier l'écriture du code et l'utilisation des threads puisque, bien entendu, s'il y a exécutions parallèles, il a forcément des threads derrière pour le permettre. Mais l'asynchronisme ne veut pas dire multitâche L'asynchronisme est bien plus proche de la technique des callbacks que de celle des threads.

Le framework fournit de nouvelles API se terminant par le mot Async pour indiquer celles qui peuvent faire l'objet d'un await. Ce mécanisme remplace le précédent qui utilisait deux méthodes par API dont les noms se terminaient
par BeginAsync et EndAsync pour démarrer l'appel asynchrone et récupérer son résultat (et clore l'appel, fermer les ressources, récupérer le code erreur éventuel…).

asynchronisme != multithreading

C'est là qu'on en revient au sujet principal de ce billet, l'asynchronisme n'est pas égal à du multithreading même si sa mise en œuvre "interne" fait souvent (mais pas toujours) usage de threads.

Par défaut un code écrit avec async/await est *monotâche* (un seul thread). Avec la méthode `Task.Run()` on peut rendre ce code multithreadé.

Un code asynchrone retourne simplement le trait avant d'avoir terminer son travail. D'autres méthodes peuvent être lancées dès lors qu'une méthode asynchrone retourne le trait.

L'asynchronisme se place dans ce possible retour rapide du trait alors que le travail n'est pas terminé, et non dans l'exécution concurrente de plusieurs codes différents.

L'asynchronisme n'est donc pas du multithreading, au moins par défaut et dans son sens premier.

Asynchronisme != parallélisme

Maintenant que nous avons posé le décor et que nous savons tous ce à quoi servent Async et Await, regardons par l'exemple pourquoi l'asynchronisme autorisé par ces mots clé n'a rien à voir avec du parallélisme et que leur simple emploi ne transforme pas *ipso facto* un code non parallèle en fusée…

Programme exemple – Squelette

Je vais utiliser un programme console pour cette démonstration.

Il se compose d'une méthode `Main()` qui exécutera trois tests différents avec une sélection du test à lancer. Le programme attendra un Return clavier avant de s'arrêter car sortir de Main fait sortir du programme tout court… Or nous allons lancer des opérations asynchrones, c'est à dire qui retournent le trait aussitôt (ou presque). De fait le Main continuera à exécuter son propre code alors que les opérations asynchrones ne seront pas terminées. Comme ces dernières seront longues, alors que le code de Main est très court (en quantité et temps d'exécution) nous ne verrions jamais les tests s'exécuter jusqu'au bout !

C'est d'ailleurs une erreur de "débutant" avec Async et Await. On oublie que le trait d'une méthode Async retourne assez vite et on ne le prend pas en compte dans le flux des méthodes appelantes ce qui peut mener à des désastres ou des bogues aléatoires.

Bref ici nous avons un Main classique et court :

```
void Main()
{
    var menu = SelectTest();
    Run(menu);
    Console.ReadLine();
    Util.ClearResults();
}
```

La variable créée au début sert à recevoir le choix du numéro de test (1, 2 ou 3). Ce choix est géré par `SelectTest()` une méthode traditionnelle utilisant la console pour la saisie du choix. Je ne vous la présenterai pas car gérer des saisies en mode console n'est vraiment pas mon propos.

La méthode `Run()` sert à lancer le test choisi, comme elle sera asynchrone elle retournera le trait assez vite. D'où la nécessité d'un `ReadLine()` qui attendra une frappe clavier validée avant de nettoyer l'écran et de laisser Mainsortir, donc le programme s'arrêter. Le nettoyage de la console s'effectue par un `Console.Clear()`, toutefois ici j'utilise l'indispensable et fantastique LINQPAD pour faire joujou avec C#. Il ne s'agit donc pas de la vraie console puisque cette application gère sa propre fenêtre de sortie. Il faut ainsi utiliser une classe utilitaire (Util) fournie avec LINQPAD pour effacer la fenêtre servant de console.

Vous savez tout sur Main !

La méthode Run(int test)

Elle joue un rôle important mais ce n'est pas elle qui constitue la démonstration. Passons vite sur son code dont l'objet est d'exécuter le test dont le numéro (1, 2 ou 3) est passé en paramètre. Cette méthode est asynchrone et marquée par async. Elle

peut donc utiliser await, ce qu'elle fait pour attendre la fin d'exécution de chaque test possible. Les trois tests étant écrits sous la forme de trois méthodes asynchrones retournant un Task.

```
static async void Run(int testId)
    {
        var start = DateTime.Now;
        Console.WriteLine("[{0}] DEBUT", start);
        string result=string.Empty;
        switch (testId) {
            case 1: result = await DoMyTasksV1("test1"); break;
            case 2: result = await DoMyTasksV2("test2"); break;
            case 3: result = await DoMyTasksV3("test2"); break;
        }
        var end = DateTime.Now;
        Console.WriteLine("[{0}] Sortie: {1}", end, result);
        Console.WriteLine(
    "[{0}] TOUTES LES TACHES SONT TERMINEES - Temps global: {1}", end,end-
start);
    }
```

La méthode `Run` mesure le temps passé dans le test exécuté. Elle indique l'heure de départ et de fin du test ainsi que le différentiel.

Les trois méthodes de test sont appelées `DoMyTasksV1`, V2 ou V3 comme on le voit dans le switch.

Regardons de plus près chacune de ces méthodes et leurs temps d'exécution…

Les tâches

Pas si vite ! Avant de regarder le détail des méthodes de test il nous mettre en place les "méthodes longues" qui seront appelées dans les tests. Ici il s'agira de simuler trois tâches longues, un envoi de mail, l'obtention d'un nombre aléatoire et celle d'une chaîne de caractères.

On supposera que chacune de ces opérations est "longue" – elles dépassent les 50 ms, limite utilisée par Microsoft pour la conception de WinRT et le choix de passer ou non une méthode en asynchrone. En réalité nous irons plus loin pour que l'effet soit bien visible. Ainsi chaque tâche (qui ne fait rien, ce sont des *fakes*) est assortie d'un délai de 2 secondes.

Voici le code très sommaire de ces trois tâches qui sont des méthodes de la classe DummyDelayResource, en français "ressource fictive retardée" (enfin à peu près).

```
public class DummyDelayResource
    {
        public Task SendEmailAsync()
        {
            Console.WriteLine("[{0}] SendMail (fake)", DateTime.Now);
```

```
            return Task.Delay(2000);
        }
        public async Task<int> GetRandomNumberAsync()
        {
            Console.WriteLine("[{0}] GetRandomNumber", DateTime.Now);
            await Task.Delay(2000);
            return (new Random()).Next();
        }
        public async Task<string> GetSpecialStringAsync(string message)
        {
            Console.WriteLine("[{0}] GetSpecialString", DateTime.Now);
            await Task.Delay(2000);
            return string.IsNullOrEmpty(message) ? "<RIEN>" :
message.ToUpper();
        }
    }
```

Nul besoin de s'appesantir sur ce bout de code : trois méthodes de test ne faisant absolument aucun travail autre que d'attendre 2 secondes. Elles ne font pas totalement rien, pour être précis elles écrivent leur nom sur la console, attendent – via un `Task.Delay()` – et pour les deux dernières elles retournent une valeur (nombre aléatoire ou chaîne de caractères passée en paramètre).

On remarquera que ces méthodes sont marquées async car elles utilisent await. Elles retournent des Taskmême celle qui ne retourne rien et qui serait codée par un void en mode synchrone. Await sert essentiellement à allonger la durée d'exécution de façon fictive en imposant une attente de 2 secondes. Attention, s'agissant de méthodes asynchrones le trait sera retourné dès que await sera rencontré. Await n'est finalement traduit que par un callback qui reviendra continuer le travail. C'est l'écriture fastidieuse de ce callback que await nous évite. Cela rend le code plus clair, plus lisible, plus séquentiel mais il ne l'est pas (séquentiel)! L'oublier c'est s'exposer à quelques soucis.

La première méthode de test retourne une instance de Task tout court, donc l'équivalent d'un void. Les deux suivantes retournent un Task<int> ou Task<string> ce qui seraient équivalents à des méthodes retournant directement int ou string. L'instance de Task (avec ou sans retour de valeur) sert à contrôler le code exécuté (notamment s'il est terminé ou non).

Premier test : résultat conforme mais temps d'exécution trop long

Pour tester le comportement de async/await le mieux est d'écrire maintenant notre première méthode de test qui en fera usage…

Voici ce code :

```
  static async Task<string> DoMyTasksV1(string message)
    {
        Console.WriteLine("[{0}] Entrée dans la méthode DoMyTasksV1...",
                          DateTime.Now);
```

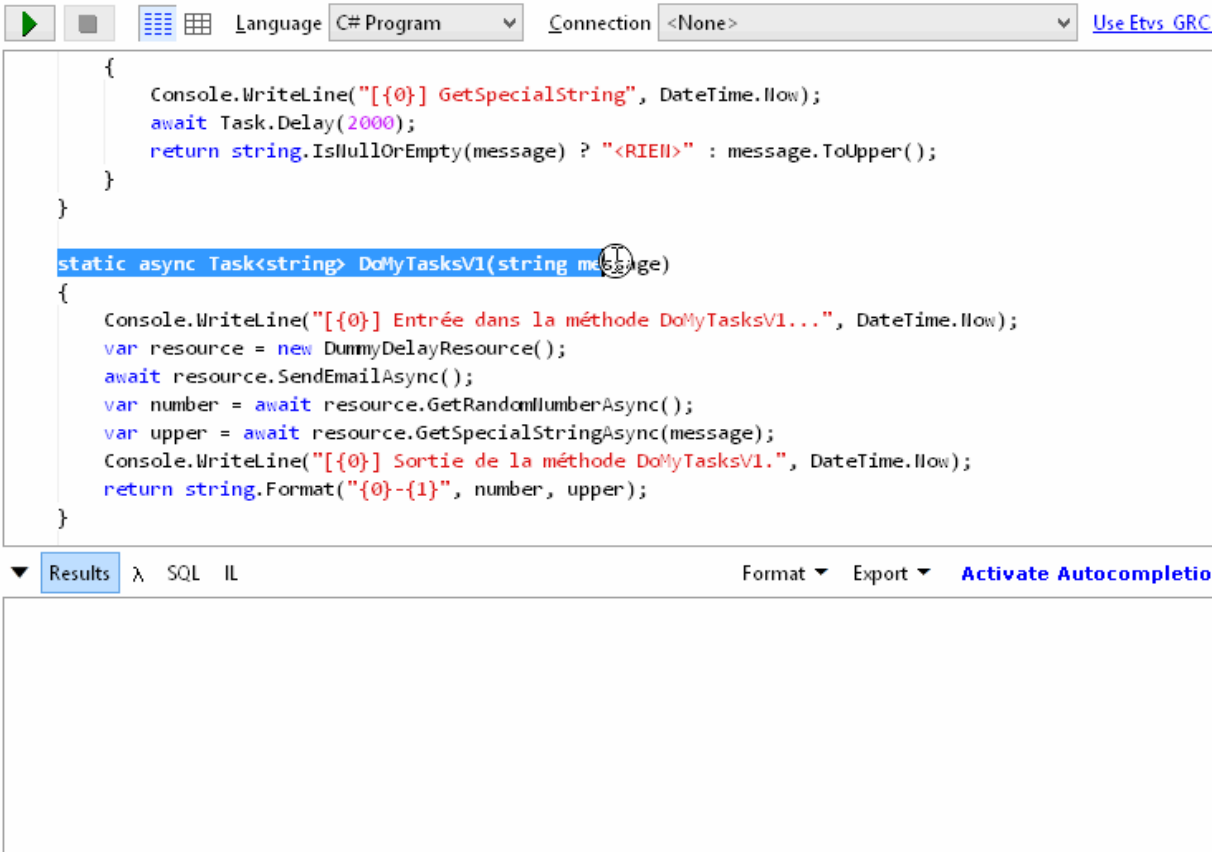
```
        var resource = new DummyDelayResource();
        await resource.SendEmailAsync();
        var number = await resource.GetRandomNumberAsync();
        var upper = await resource.GetSpecialStringAsync(message);
        Console.WriteLine("[{0}] Sortie de la méthode DoMyTasksV1.",
                          DateTime.Now);
        return string.Format("{0}-{1}", number, upper);
    }
```

Il s'agit bien entendu d'une méthode marquée par async. Elle retourne un Task<string> et elle accepte une paramètre de type string qu'elle utilisera pour fabriquer sa valeur de retour,

Après avoir indiqué son nom avec l'heure, elle créée une instance de la classe de test. Ensuite elle exécute les trois méthodes de test l'une après l'autre avec un await pour garantir la "*séquentialité* " du code. Chaque méthode de test est elle-même une méthode retournant un Task et étant marquée async. Il est donc important de faire unawait si on veut s'assurer de l'ordre d'exécution de la méthode de test.

Cela semble très raisonnable et parfaitement conforme à l'utilisation qui peut être faite de async/await.

Regardez le GIF ci-dessous qui montre l'exécution de cette méthode (forcément un PDF ne contient pas d'images animées, regardez l'original sur Dot.Blog !) :



En fin de travail nous obtenons la durée globale. Si on fait abstraction des millisecondes passées ici ou là dans notre code, le test complet dure 6 secondes.

Finalement cela est bien naturel : notre méthode de test exécute trois méthodes qui chacune simule une ressource lente avec un délai de 2 secondes. Même un neuneu non initié aux subtilités mathématiques de notre belle profession comprendra aisément que 3 fois 2 font 6 …

Et c'est bien ce que je veux vous montrer : Asynchronisme != Parallélisme !

En utilisant async/await on ne gagne absolument rien en temps d'exécution … notre application gagne en réactivité et en fluidité, son thread principal n'est plus bloqué par les méthodes longues qui sont exécutées dans des threads séparés, mais le simple fait de rendre séquentiel ce qui ne l'est pas forcément, par await, oblige notre application à attendre à chaque étape avant de passer à la suivante. Les temps se cumulent et chaque méthode de test prenant 2 secondes, l'ensemble des tests en prend 6.

C'est tout de même dommage d'avoir déployé autant de ruse et d'intelligence pour en arriver à une application, certes fluide, mais très lente !

L'asynchronisme ne devrait-il pas nous offrir un meilleur résultat ?

Non. Car l'asynchronisme s'occupe de régler un problème particulier qui n'a rien à voir avec le celui réglé par le parallélisme. Async/await permettent d'écrire un code lisible dont l'ordre d'exécution est maitrisable. Certes pour que les méthodes asynchrones s'exécutent il faut bien que des threads soient créés, mais c'est await qui casse le parallélisme, et c'est bien son rôle : attendre !

La véritable question qu'il faut se poser est "quand attendre et quand ne pas attendre".

Attendre mais pas trop…

En réfléchissant à cette question cruciale nous nous apercevons qu'il est un peu idiot d'attendre la fin deSendMail pour passer aux deux autres tests. En effet, SendMail n'a aucune dépendance avec les méthodes suivantes, elle ne dépend pas des valeurs qu'elles retourneront. Nous pouvons donc retourner le trait immédiatement, nul besoin d'attendre la fin d'exécution de SendMail.

Cela nous amène à une version un peu plus optimisée de notre code :

```
static async Task<string> DoMyTasksV2(string message)
{
    Console.WriteLine("[{0}] Entrée dans la méthode DoMyTasksV2...",
                      DateTime.Now);
    var resource = new DummyDelayResource();
    var emailTask = resource.SendEmailAsync();
    var number = await resource.GetRandomNumberAsync();
```

```
    var upper = await resource.GetSpecialStringAsync(message);
    await emailTask;
    Console.WriteLine("[{0}] Sortie de la méthode DoMyTasksV2.",
                      DateTime.Now);
    return string.Format("{0}-{1}", number, upper);
}
```

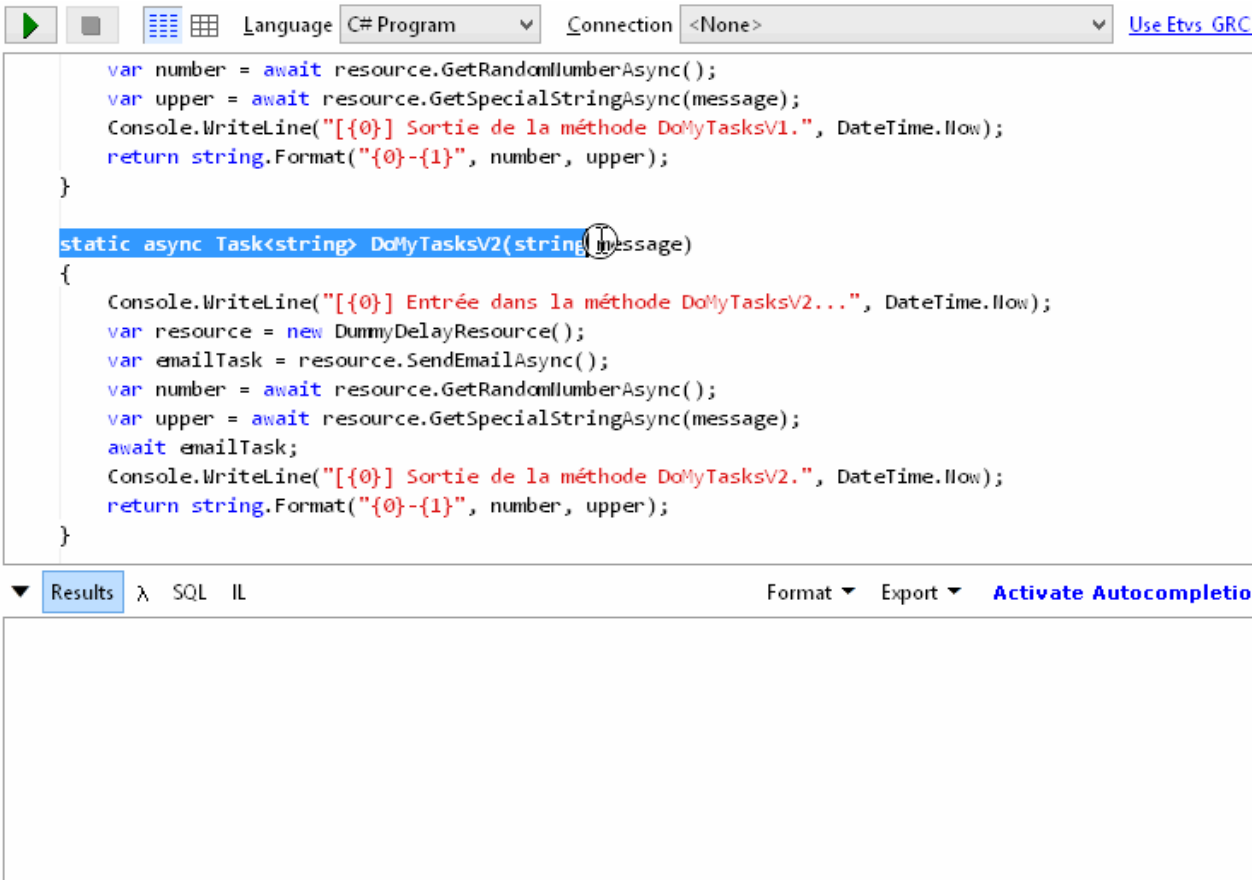
Comme on le voit la tâche envoyant le mail n'est pas attendue, il n'y a plus de await devant son appel. De fait le trait passera immédiatement à la suite qui consiste à obtenir le nombre aléatoire puis la chaîne de caractères.

Mais nous avons besoin d'être sûr que l'email est bien parti avant de sortir définitivement de notre méthode de test (disons que c'est une contrainte fonctionnelle de notre application). Avant que la méthode ne se termine nous nous assurons que la tâche d'envoi de mail est bien terminée en utilisant un await sur la Task retournée plus avant…

Sommes-nous pour autant certain que toutes les autres tâches sont terminées aussi ? Oui puisqu'elles sont exécutées avec un await…

Attendre pour SendMail est donc suffisant ici pour nous assurer que toutes les méthodes de test auront bien été exécutées avant que la méthode de test elle-même ne retourne définitivement le trait…

Ce qui donne le GIF suivant (n'oubliez pas que les images de Dot.Blog sont généralement cliquables pour les voir dans leur résolution complète, les GIF de ce billet n'échappant pas à cette règle, ce qui les rendra plus lisible. Un GIF animé n'a pas de sens dans un PDF donc regardez-le sur Dot.Blog !) :



Nous venons d'un seul coup d'optimiser notre application de 33,33 % !

Par rapport à la version précédente, la méthode de test ne prend plus que 4 secondes pour s'exécuter bien qu'elle nous garantisse toujours de ne se finir qu'une fois toutes les tâches terminées. C'est magique ?

Non, c'est juste que nous avons deux tâches en await soit 2 x 2 … oui, j'en vois un qui lève la main… Oui ! Bravo cela fait 4 secondes ! Mais où sont passées les 2 secondes "manquantes" ?

La tâche d'envoi de mail s'est juste exécutée en parallèle des 2 autres. Alors bien entendu il n'y a pas de miracles, s'il n'y avait qu'un seul cœur à notre PC et si le code faisant réellement quelque chose au lieu de créer un délai vide, il n'y aurait aucun gain sauf la fluidité. Mais sur toute machine récente, même un smartphone, globalement de 6 secondes nous passons à 4 secondes d'exécution sans rien sacrifier à nos exigences fonctionnelles ni à la linéarité et la lisibilité de notre code.

Mais c'est ici qu'on s'aperçoit aussi très vite que ces histoire de await ce n'est pas aussi simple qu'on l'envisageait au départ… Surtout si on transpose ce tout petit exemple fictif à une application de plusieurs dizaines de milliers de lignes de code ou plus !

Bon. Réfléchir, progresser, c'est l'un des attraits de notre profession, garder l'œil vif et les neurones en mouvement… Donc on veut pousser les choses plus loin car plus on pense plus on devient exigeant…

N'y aurait-il donc pas un moyen d'améliorer encore notre méthode de test ? Le mieux serait de n'attendre que le plus long des processus. Ici tous prennent 2 secondes mais cela n'est pas représentatif de la réalité ou une stricte égalité de temps de traitement serait même totalement incroyable.

Attendre le minimum c'est encore mieux !

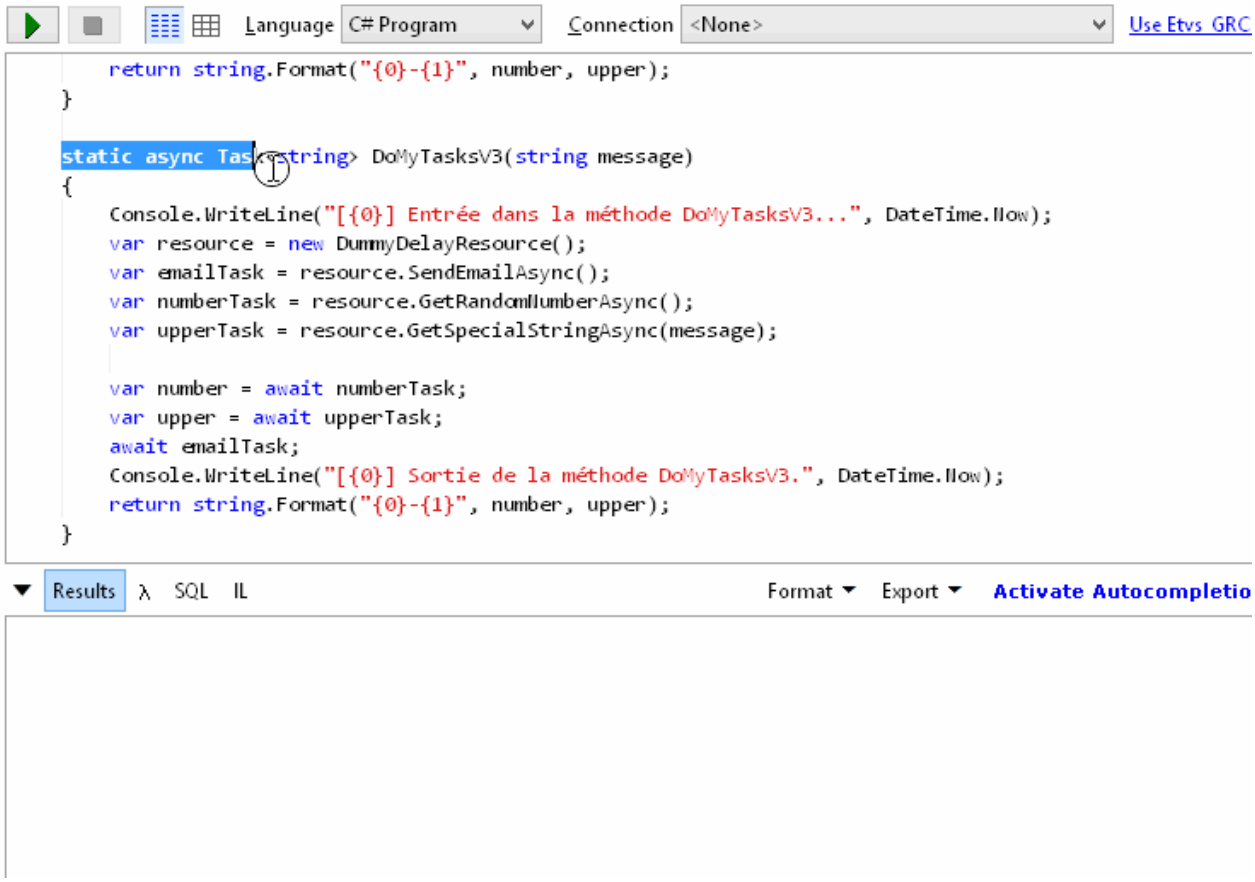
Voici le code de notre troisième méthode de test :

```
static async Task<string> DoMyTasksV3(string message)
    {
        Console.WriteLine("[{0}] Entrée dans la méthode DoMyTasksV3...",
                           DateTime.Now);
        var resource = new DummyDelayResource();
        var emailTask = resource.SendEmailAsync();
        var numberTask = resource.GetRandomNumberAsync();
        var upperTask = resource.GetSpecialStringAsync(message);

        var number = await numberTask;
        var upper = await upperTask;
        await emailTask;
        Console.WriteLine("[{0}] Sortie de la méthode DoMyTasksV3.",
                          DateTime.Now);
        return string.Format("{0}-{1}", number, upper);
    }
```

Puisqu'il n'existe aucune contrainte entre les trois méthodes de la ressource lente simulée il n'y a aucune raison d'attendre l'une plus que l'autre, les trois tâches peuvent être lancées de façon concurrente… C'est exactement ce que fait le code ci-dessus : l'envoi de mail, le calcul aléatoire et le retour d'une chaîne de caractères sont tous lancés à la suite sans aucune pause, sans aucun await.

De fait tous vont s'exécuter en parallèle ce que montre le GIF suivant :



Temps global, 2 secondes (et quelques miettes).

De 6 secondes pour la première méthode nous avons réussi à réduire le temps d'exécution à 2 secondes, soit un gain de 66.66 % !

De la bonne façon d'attendre

Dans le dernier exemple ci-dessus les trois appels aux méthodes de la ressource sont lancés puis on a choisi d'attendre les tâches par un await.

Finalement ce code ne diffère pas vraiment du premier ! Mais au lieu de faire un await sur l'appel de chaque méthode on le fait sur les variables dans lesquelles nous avons stocké les Task retournés par le lancement des trois méthodes.

C'est peu de choses mais cela change tout. La preuve, de 6 secondes à 2 secondes. Pas besoin d'exagérer ou de balancer des superlatifs savants. Le gain est évident.

Toutefois l'écriture de tous ces await qui se suivent (et dans quel ordre si on veut être précis ?) n'est pas la plus gracieuse. Il serait plus agréable d'écrire un code plus intelligent, plus concis aussi.

C'est bien entendu dans la classe Task que se trouve la réponse avec les méthodes comme WaitAll() ou mieux WhenAll() qui gère correctement le parallélisme que nous recherchions.

Task : la porte des étoiles

Task est une classe très riche, à la fois par son importance dans la gestion du parallélisme, de l'asynchronisme mais aussi par les méthodes et propriétés qu'elle offre (toute une série suit sur ce sujet dans ce livre).

Etudier Task dépasse largement le cadre de ce billet. Notons toutefois pour allécher le lecteur qui n'aurait pas encore investiguer cet océan de possibilités qu'il est possible d'utiliser des choses comme ContinueWith() qui va permettre de fixer un ordre d'exécution dans une chaîne de tâches asynchrones. On peut aussi utiliser Start() qui lance la tâche mais dans le cadre d'une planification gérée par TaskScheduler. On retrouve aussi Wait() très proche de await mais cette fois-ci en tant que méthode de Task et non comme mot clé de C#. WaitAll(), WaitAny(), aident à gérer des groupes de tâches d'une façon bien plus simple que le multithreading d'il y a quelques années. Task est ainsi le cœur d'une nouvelle façon de programmer à la fois *asynchrone* et *parallèle* avec le framework .NET en C#.

Qu'il s'agisse d'une ou deux tâches ou d'une liste de tâches éventuellement observables via TaskObservableExtensions, Task est la pierre angulaire d'une programmation moderne, fluide et efficace, tant du point de vue du code lui-même que de l'application qui sera exécutée.

Je ne peux que vous conseiller vivement de vous intéresser à cette classe et tout ce qui s'y rattache tellement tout cela est essentiel.

Conclusion

Async et await ne sont pas grand chose, juste deux mots clés. Mais ils ouvrent la voie à une programmation plus claire, plus maintenable mais aussi à des applications plus fluides, plus réactives et plus puissantes. La classe Task est le pivot autour duquel gravitent des millions de possibilités. Encore faut-il avoir conscience des effets de bord de async/await et des optimisations importantes du code qu'on peut obtenir en les manipulant correctement.

C'est un sujet très vaste touchant à deux domaines incroyablement riches mais aussi parfois complexes à maitriser. L'asynchronisme et le parallélisme ouvrent au moins autant de questions qu'ils n'apportent de réponse… Un simple billet ne peut avoir la prétention d'avoir couvert ces univers. Tout juste peut-il prétendre avoir aiguisé votre curiosité et avoir attiré votre attention sur le sujet. Et je me satisferai sans problème d'avoir juste et par chance réussi à atteindre cet humble objectif ici…

Task, qui es-tu ? partie 1

J'ai souvent parlé de parallélisme, de PLINQ, de multitâche, d'async/await, mais je me rend compte que Task est la grande oubliée de tout ça. Corrigeons cela par une petite série d'articles !

Task la fuyante…

En faisant le tour de ce que j'avais dit sur la question je suis me suis aperçu que la classe Task (et Task<T>) était un peu les oubliées de tous ces brillants (soyons modeste!) exposés sur le parallélisme, le multitâche et l'asynchronisme (autant de notions différentes dont vous savez tout j'en suis certain, sinon relisez Dot.Blog).

Lors de cette prise de conscience je me suis aperçu que Task était souvent négligée. Peut-être en raison d'une disposition particulière liée à son apparition au milieu de plein d'autres choses qui semblent parler de la même chose.

Ainsi il existe chez les développeurs deux approches très tranchées :

- Ceux qui ont déjà utilisé Task et la TPL (dont j'ai parlé en son temps) – Task Parallel Library – depuis sont introduction dans .NET 4.0. Ces développeurs-là sont familiers avec la classe Task et son utilisation au sein de la mécanique subtile qu'est le parallélisme. Un danger : celui encouru par tous les early adopters… C'est qu'on croit savoir puisqu'on a même fait partie des premiers, et du coup on est à côté de ses pompes dans le présent car beaucoup de choses ont changée et qu'on ne s'est pas donné la peine de tout revoir… or Task de TPL est très différent du Task utilisé aujourd'hui avait await notamment.
- Ceux qui n'ont jamais entendu parlé de Task jusqu'à l'apparition de async dans C#. Un danger : pour eux Task n'est qu'une partie accessoire participant à la mise en œuvre de async/await. Un truc de plus, compliqué en plus, à apprendre. La prise de conscience d'une continuité est absente… Chaque méthode Task est considérée comme faisant partie de ce tout orienté async/await alors qu'il n'en est rien.

Bien entendu il y a le troisième groupe qui n'a jamais ni utilisé ni entendu parlé de TPL et encore moins de async/await, mais là ça doit être des développeurs C++ ou Java ou un truc comme ça, ça compte pas…

L'équipe de développement qui a introduit "async" et ses mécanismes a été tenté pendant un temps de créer sa propre classe mais réutiliser Task a fini par l'emporter pour de nombreuses raisons et certainement la plus importante : elle existait et elle pouvait faire le job sans multiplier les solutions proches et confusantes. Ainsi il fut

décidé d'étendre Task pour satisfaire les besoins de async. Créant de la confusion en voulant en éviter…

Ainsi la classe Task contient des choses qui ne servent à rien à la logique async et qui peuvent rendre perplexes les développeurs du second groupe évoqué plus haut, autant qu'elle peut sembler être la même qu'à l'époque de TPL et être utilisée de la même façon ce qui mène à des déconvenues et à du mauvais code chez les développeurs du premier groupe…

Les deux faces d'une tâche

Donc depuis "async" Task est un même noyau avec deux visages, à la façon de la tête d'un Terminator un peu esquinté…

On peut même dire qu'il y a deux types de tâches. Le premier type est celui des "delegate tasks". Ce sont des tâches qui proposent un code à faire tourner. Le second type est appelé "promise tasks", ce sont des tâches qui représentent des évènements ou des signaux. Elles sont souvent basées sur des entrées/sorties comme un évènement de fin de téléchargement HTTP par exemple. Mais elles peuvent servir à plein de choses très différentes comme la fin d'un timer de n secondes.

Dans le monde de la TPL les tâches étaient toutes, ou presque, de type "Delegate Task". C'est-à-dire des tâches possédant un code à faire tourner. La TPL offrait un support minimaliste des "promise tasks".

On notera que j'ai abandonné l'idée de traduire les termes "Delegate task" et "Promise task", le lecteur m'en excusera j'en suis certain d'autant que je pense que cela lui sera plus profitable et plus clair d'utiliser ces termes qu'il rencontrera ailleurs plutôt que d'inventer des traductions farfelues qui ne l'éclaireront pas dans ces futures lectures…

Quand il y a des traitements parallèles les Delegate tasks sont distribués sur plusieurs threads qui exécutent réellement le code de ces dernières. Dans le monde async la plupart des tâches sont des Promise task. Il n'y a aucun thread accroché à leur existence et qui attendrait que la tâche se termine…

Cette différence est tellement énorme qu'elle valait bien un billet d'introduction !

Vous pourrez parfois lire les termes de "code-based tasks" et de "event-based tasks" en place et lieu de "Delegate tasks" et "Promise tasks". On comprend que l'esprit est le même, la terminologie est juste plus ancienne. Les tâches "code-based" sont celles qui contiennent du code à exécuter, comme un Delegate, alors que les tâches "event-

based" sont conçues sur la notion d'évènement c'est à dire d'une promesse d'un quelque chose venir ("Promise task", des "tâches promesses").

Mais le plus essentiel de tout cela est bien de comprendre que les Delegate tasks contiennent du code à exécuter qui le sera dans un thread alors que les Promises tasks ne font qu'attendre un évènement et ne nécessitent aucun thread pour cela, en tout cas de façon obligatoire.

C'est en cela que "classe Task = multitâche" est une confusion horrible qui interdit de comprendre et d'utiliser convenablement les tâches …

Je l'avais déjà dit dans mon billet "De la bonne utilisation de Async/Await en C#" : "*les threads ne sont pas des taches*", mais tout cela était orienté async/await et je m'aperçois que le message n'avait peut-être pas été énoncé avec assez d'emphase pour qu'il passe assurément !

Conclusion

Bien comprendre Task c'est déjà saisir cette nuance énorme entre Delegate Task et Promise Task, comprendre l'histoire d'une fusion entre TPL et async/wait qui peut créer de la confusion.

Nous verrons dans la seconde partie comment une tâche se construit…

Task, qui es-tu ? partie 2

Continuons cette petite série sur Task. Après avoir vu le piège que la nature même de Task tendait, voyons à quoi ressemble cette classe et son instanciation.

Dis-moi comment tu te construis et je te dirais qui tu es !

Souvent le constructeur, ou les constructeurs, d'une classe en disent beaucoup sur son utilisation à venir. Task n'échappe pas à la règle mais avec une variante intéressante : on n'utilise pourrait ainsi dire jamais son constructeur !

Toutefois l'étude de la construction d'une instance nous apprend des choses intéressantes sur la finalité et les options qui permettent de donner vie à tâche même si ces constructions sont souvent cachées par d'autres mécanismes.

Les constructeurs

Task possède plusieurs constructeurs. Il s'avère que l'équipe en charge de la BCL, la librairie de base de .NET, n'aime pas les paramètres par défaut car ils pensent, avec raison, que ce type d'écriture ne s'accorde pas avec la nature d'une librairie comme .NET et ses évolutions. Dans un logiciel précis il est pratique d'ajouter des valeurs par défaut à certains paramètres, mais lorsqu'on doit gérer les évolutions dans le temps d'une énorme librairies de classes mieux vaut s'en tenir à des constructeurs

différenciés clairement. Il est toujours facile d'en ajouter un de plus que d'ajouter un paramètre sans valeur par défaut par exemple (car s'il y a des valeurs par défaut elles sont en fin de liste et on ne peut pas ajouter autre chose d'un paramètre avec valeur par défaut…).

C'est donc pourquoi Task propose 8 constructeurs :

```
Task(Action);
Task(Action, CancellationToken);
Task(Action, TaskCreationOptions);
Task(Action<Object>, Object);
Task(Action, CancellationToken, TaskCreationOptions);
Task(Action<Object>, Object, CancellationToken);
Task(Action<Object>, Object, TaskCreationOptions);
Task(Action<Object>, Object, CancellationToken, TaskCreationOptions);
```

On pourrait fort bien tous les regrouper en un seul constructeur logique qui possèderait des paramètres par défaut, cela donnerait :

```
Task(Action action, CancellationToken token = new CancellationToken(),
TaskCreationOptions options = TaskCreationOptions.None)
    : this(_ => action(), null, token, options) { }
Task(Action<Object>, Object, CancellationToken = new CancellationToken(),
TaskCreationOptions = TaskCreationOptions.None);
```

On notera que ce ne serait pas forcément plus clair d'ailleurs… D'où en parallèle, si je puis dire sur un tel sujet, la réflexion qui s'ouvre sur l'utilisation ou non des paramètres par défaut, les avantages, les inconvénients, etc… je vous laisse y méditer !

Task existe aussi en version générique, Task<T> qui propose la même déclinaison :

```
Task<TResult>(Func<TResult>);
Task<TResult>(Func<TResult>, CancellationToken);
Task<TResult>(Func<TResult>, TaskCreationOptions);
Task<TResult>(Func<Object, TResult>, Object);
Task<TResult>(Func<TResult>, CancellationToken, TaskCreationOptions);
Task<TResult>(Func<Object, TResult>, Object, CancellationToken);
Task<TResult>(Func<Object, TResult>, Object, TaskCreationOptions);
Task<TResult>(Func<Object, TResult>, Object, CancellationToken,
TaskCreationOptions);
```

Liste qu'on pourrait s'amuser à réduire de la même façon à un seul constructeur avec paramètres par défaut :

```
Task<TResult>(Func<TResult> action, CancellationToken token = new
CancellationToken(), TaskCreationOptions options =
TaskCreationOptions.None)
    : base(_ => action(), null, token, options) { }
Task<TResult>(Func<Object, TResult>, Object, CancellationToken,
TaskCreationOptions);
```

Ce qui n'est franchement pas plus clair, c'est évident !

Task et son double Task<T> arrivent donc avec 16 constructeurs qu'on peut réduire à 2 constructeurs bourrés de paramètres optionnels.

Le choix semble pléthorique … D'autant qu'on n'utilise jamais les constructeurs de Task ou Task<T> !

Pourquoi cette situation à première vue surprenante ?

Tout simplement parce que TPL ou async offrent peu de place à des cas d'utilisation où l'appel direct à un constructeur de Task serait utile…

Comme je l'ai expliqué dans la première partie il y a deux type de Tasks : les Delegate et les Promise Tasks. Les constructeurs de Task ne peuvent pas créer des Promise Tasks, uniquement des Delegate tasks. Ce qui déjà élimine une bonne partie des use cases où le constructeur serait utilisable.

Puisque la logique async fait que le constructeur n'est pas utile (pour le code que vous écrivez en tout cas) il ne peut en toute logique que trouver sa place dans une logique TPL, c'est à dire création de tâches parallèles.

Dans un tel contexte il existe deux grandes familles de tâches, les tâches qui parallélisent des données et celles qui parallélisent du code. Ces dernières peuvent encore se subdiviser en parallélisme statique et parallélisme dynamique. Dans le premier cas le nombre de tâches est fixe et connu à l'avance, dans le second le nombre de tâche peut évoluer en cours d'exécution.

La classe ParalleIn PLINQ et les types de la TPL offrent des constructions de haut niveau pour gérer le parallélisme des données et celui du code. La seule véritable occasion de créer un Task se limite donc à lan branche du parallélisme de code dynamique.

Mais même dans ce cas bien particulier il est rare de le faire… le constructeur de Task fabrique une tâche qui n'est pas prête à tourner car elle doit être planifiée avant. Séparer ces deux actions est rarement souhaitable car on désire le plus souvent planifier la tâche immédiatement. La seule raison qui pourrait faire qu'on désire créer une tâche sans la planifiée tout de suite serait de pouvoir laisser le choix d'attribuer (de planifiée) un thead particulier pour exécuter la tâche. Ce cas est un peu tordu et même là vaudrait-il encore mieux utiliser Func<Task> au lieu de retourner une tâche non planifiée.

Si on résume : si vous voulez faire du parallélisme dynamique et que vous avez besoin de créer des tâches qui peuvent être exécutées sur n'importe quel thread, et que le choix de ce thread est une décision déléguée à une autre partie du code, et

que pour une raison à déterminer vous ne pouvez pas utiliser Func<Task>, et bien là, et seulement là vous aurez besoin d'appeler l'un des constructeurs de Task !

Autant dire qu'il s'agit d'une situation tellement spéciale, spécifique, tordue, qu'à l'heure actuelle je n'en ai jamais vu le moindre exemple fonctionnel et n'en ai encore moins jamais écrit.

Vous comprenez pourquoi l'utilisation des constructeurs de Task n'est pas interdite en soi (sinon ils ne seraient pas accessibles, .NET est bien écrit) mais réservée à quelques cas purement hypothétiques que les concepteurs de la BCL n'ont pas voulu … interdire. Ils l'auraient fait que je doute que qui que ce soit aurait été lésé. Les parents finissent en général par le comprendre, parfois mieux vaut cacher que montrer et expliquer pourquoi il ne faut pas toucher ! 

Comment créer des Task alors ?

C'est fort simple ce n'est pas votre problème ! En programmation on dira de façon policée et plus technique "ce n'est pas de la responsabilité de votre code".

Soit vous écrivez du code async et c'est le mot clé async qui est le moyen le plus simple pour créer du code de type Promise Task. Soit vous englober une autre API asynchrone ou un évènement et vous utilisez Tasl.Factory.FromAsync ou TasCompletionSource<T>. Et si vous avez besoin de faire tourner une tâche en parallèle d'un autre code, vous utilisez Task.Run.

Nous verrons cela dans de prochaines parties. Mais pour du code parallèle commencez en général à regarder du côté de la classe Parallel ou de PLINQ. Et sauf si vous faites du parallélisme dynamique le mieux est d'utiliser Task.Run ou Task.Factory.StartNew.

Conclusion

Les constructeurs de Task et Task<T> nous apprennent des choses, assurément. On voit de quoi une tâches peut avoir besoin pour tourner et aussi ce qu'on peut attendre. La présence par exemple d'un CancellationToken nous laisse penser qu'il y a des moyens prévus pour annuler une tâches en cours. De même que le paramètre TaskCreationOptions laisse déviner que certains paramètres doivent autoriser un réglage fin de la tâche à créer.

Tout cela est vrai et nous donne une vision sur les entrailles de la bête.

Néanmoins la création directe d'une instance de Task est chose rarissime car il existe des moyens de plus haut niveau pour le faire, et ce sont ces moyens que nous utilisons dans notre code.

Il n'y a pas d'interdiction d'utiliser les constructeurs de Task comme on peut le lire parfois, il y a tout simplement un manque cruel de situations réelles où cela aurait la moindre utilité…

Passons plutôt à la suite, donc…

Task, qui es-tu ? partie 3

Avançons petits pas par petits pas dans cette meilleure connaissance de Task et Task<T>. Qu'est-ce que l'AsyncState ?

Résumé

Task et Task<T> présentent deux facettes, l'une proche de la TPL (parallélisme) et l'autre de async (asynchronisme). Qu'on en fasse usage dans un cas ou dans l'autre il n'y a jamais besoin d'utiliser son constructeur car les Task sont créées par des moyens de plus haut niveau.

AsyncState

Encore un étape nécessaire de dépoussiérage pour bien comprendre Task…

En effet la propriété AsyncState qui implémente l'interface IAsyncResult.AsyncState est un vieux reste de vieilles façons de faire de l'asynchronisme.

```
object AsyncState { get; } // implements IAsyncResult.AsyncState
```

Si vous aviez investi dans la compréhension des mécanismes sous-jacents faisant intervenir AsyncState sachez que cette connaissance n'est plus utile…

Cela remonte aux âges anciens de la programmation asynchrone lorsque .NET s'est paré de méthodes doublées par des sœurs dont le nom était préfixé de Begin et même triplées par une autre méthode préfixée End. Les méthode sans préfixes étaient synchrones (et généralement celles originellement présentent dans la librairie), celles avec Begin offraient la possibilité alors nouvelle de traiter l'appel de façon asynchrone. C'était le modèle appelé APM ou Asynchonous Programming Model.

Je ne reviendrais pas sur APM (vous pouvez suivre le lien qui mène à la documentation officielle) car c'est un modèle dépassé.

Du coup à quoi sert encore AsyncState.

Je serai tenté de dire, à rien. En réalité on peut toujours lui trouver une raison d'être, notamment dans la compatibilité entre code async et vieux code suivant APM. Là il peut être intéressant d'utiliser l'AsyncState.

Mais n'allons pas plus loin. Inutile de montrer des choses qui ne serviront pas.

Conclusion

Elément indispensable au modèle APM, AsyncState ne peut aujourd'hui jouer de rôle que dans des situations ou nouvelles et anciennes constructions doivent coexister. Maintenir du code dans le temps est une chose noble et utile. Mais de deux choses l'une : soit vous avez du vieux code APM et vous connaissez AsyncState, par force, soit vous n'avez pas de vieux code APM à maintenir ou à mélanger au votre, et dans ce cas, si vous ne savez pas à quoi sert AsyncState dormez tranquille !

Comme je le disais dans la partie 1 le fait de connaitre certaines techniques ou technologies avant les autres possède un pendant terrible : on a l'esprit encombré de choses qu'on pense indispensables alors que les jeunots qui viennent directement aux nouvelles façons de faire utilisent tout de suite correctement les nouvelles possibilités sans se faire de mouron…

Pour l'early adopter encore plus que pour les autres, apprendre à désapprendre est parfois plus utile qu'apprendre tout court.

Oubliez AsyncState, vous ne comprendrez que mieux ce qu'est aujourd'hui Task !

Task, qui es-tu ? partie 4

Poursuivons cette petite série sur Task. Passons rapidement sur les options de création, avant de nous lancer dans le vif du sujet.

CreationOptions

```
TaskCreationOptions CreationOptions { get; }
```

Voici une définition de propriété qui ne nous renseigne pas totalement… Sauf que nous savons maintenant que CreationOptions est en lecture seule.

Et de fait cette propriété sert à cela : à prendre connaissance des options de création utilisées lors de l'instanciation de la tâche. Comme la tâche existe déjà ces options ne sont plus modifiables.

Les options de création ne sont pas utilisables qu'avec les constructeurs de Task et heureusement puisque nous avons vu qu'ils étaient rarement utilisés… On peut ainsi préciser ces fameuses options lorsqu'on utilise Task.Factory.StartNew ou TasCompletionSource<T>.

Nous verrons plus en détail au moment de l'étude de ces méthodes à quoi servent réellement les options de création, une énumération qui permet d'indiquer si la tâche doit être exécuter rapidement, s'il s'agit d'une tâche longue (ce qui permet au scheduler d'autoriser une sorte de surbooking avec plus de tâches qu'il n'y a de CPU disponibles) ou bien si elle doit être attachée à son parent (principe des nested tasks).

En tout cas pour ce qui est de la propriété elle-même il n'y a généralement aucune raison de lire sa valeur car on voit mal quelle utilisation on en ferait…

Conclusion

Voilà, le ménage est fait nous allons pouvoir aborder des choses plus utiles dans la 5ème partie à venir…

Task, qui es-tu ? partie 5

Nous allons continuons à explorer les différentes propriétés de Task, aujourd'hui examinons le Status d'une tâche.

Status

Cette propriété doit être comprise comme l'état de la mini machine à états qu'est une Task. On pourra relire à ce sujet un article récent qui portait justement ce sujet (les machines à états finis).

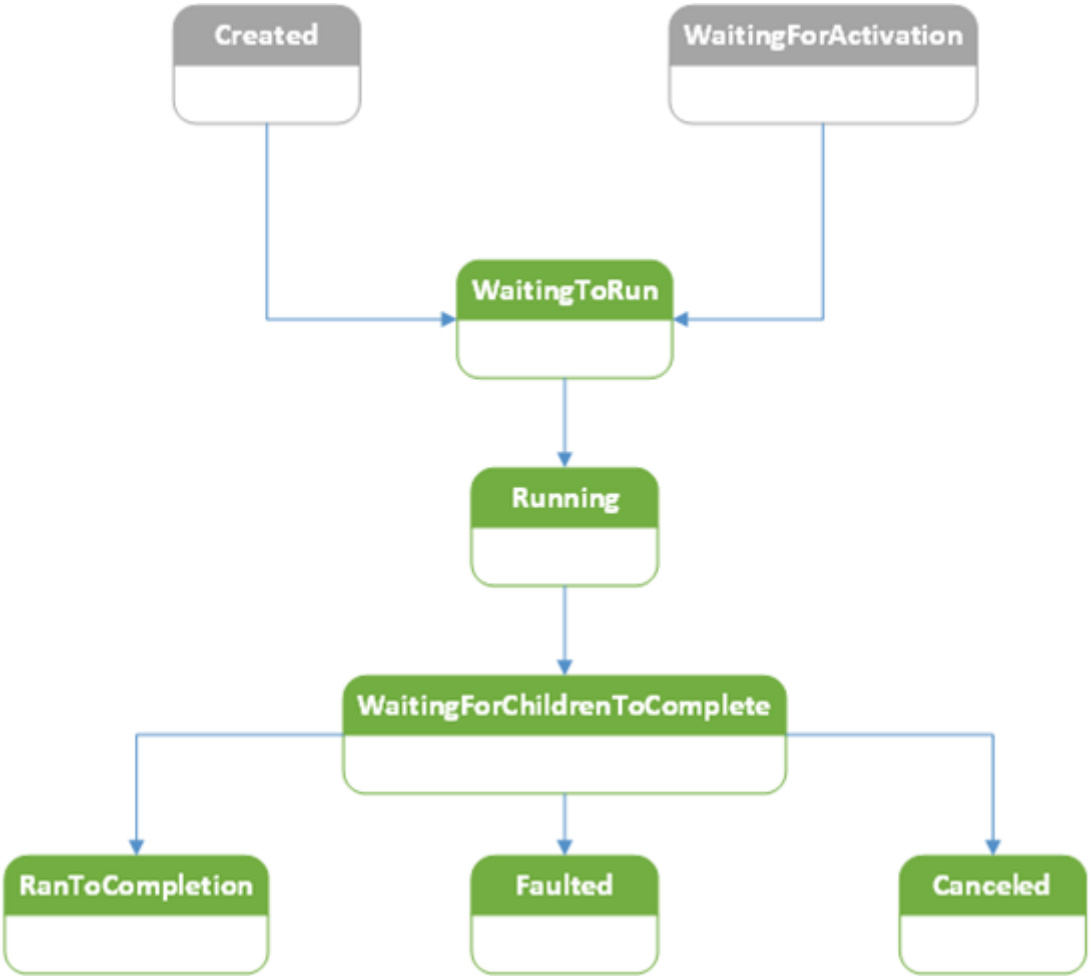
Savoir dans quel état se trouve une tâche est une information qui peut avoir son importance dans certains cas. Pour les systèmes de haut niveau que nous utilisons pour créer des Task il est certains que cela est utile, pour le code que nous écrivons l'importance est moindre.

```
TaskStatus Status { get; }
```

Toutefois comme l'état change de valeur dans des cas un peu différents selon qu'on utilise une Promise Task ou une Delegate Task et que cette nuance fondamentale doit être toujours présente à l'esprit quand on parle de Task, il est intéressant d'aller voir cela plus en profondeur…

Le cas des Delegate Tasks

Les différents états d'une tâche en mode Delegate Task sont représentés ci-dessous :



Le plus souvent les Delegate Tasks sont créées en utilisant Task.Run (ou Task.Factory.StartNew) et entrent immédiatement dans l'état WaitingToRun. Ce qui signifie que la tâche est créée et planifiée par le scheduler et qu'elle attend simplement son tour.

Les états en gris représentent des états qu'on ne voit que lorsqu'on utilise les constructeurs de Taks, ce qui est rarissime comme nous avons pu le voir. Par exemple après l'appel d'un constructeur l'état, en toute logique, devient Created. Une fois la tâche assignée à un scheduler son état passe à WaitingToRun (en utilisant Start ou RunSynchronously).

Si la tâche est la continuité d'une autre tâche alors elle démarre en mode WaitingForActivation et bascule automatiquement vers WaitingToRun quand la tâche qui la précède se termine.

L'état Running est évident : le code de la Delegate Task est en cours d'exécution.

Quand ce code a terminé son exécution la tâche passe si besoin par un état d'attente de ses tâches enfants, ce qui est matérialisé par l'état WaitingForChildrenToComplete.

On remarque au passage qu'un librairie qui utilise des noms assez longs et assez clairs se passent de documentation assez facilement…

A la fin de cycle la tâche peut prendre trois statuts différents : RanToCompletion qui indique que tout s'est bien passé, Faulted qu'on comprend comme l'indication qu'une erreur est survenue dans le mécanisme, et Canceled si la tâche a été volontairement stoppée.

Il est bien entendu assez rare de pouvoir suivre tous ces états en temps réel, tout va très vite et certains états sont franchis avant même qu'on puisse lire l'état de Status. De même une annulation de la tâche pourra la faire passer à Canceled avant même qu'elle passe par Running.

Donc lire Status peut avoir un intérêt dans quelques cas mais c'est surtout l'énoncé de ses valeurs possibles qui est intéressant pour comprendre le mécanisme des Task.

Le cas des Promise Tasks

Rappelez-vous que les Promise Task ne sont en réalité pas affectée à un thread et qu'il n'y a pas de code qui tourne, ce sont juste des évènements. Tout naturellement les états possibles reflète cette nature totalement différente bien qu'il s'agisse toujours de la classe Task…

Comme il ne s'agit pas de représenter tous les diagrammes UML de Task il y a par force des simplifications. Les Promise Tasks peuvent éventuellement passer par l'état WaitingForChildrenToComplete notamment. Mais cela est tellement bizarre que les tâches créées par async le sont avec le flag DenyChildAttach, rejetant ainsi la possibilité d'accrocher des tâches enfants.

On pourrait se demander malgré tout pourquoi il n'y a pas d'état Running. Après tout lorsqu'on attend qu'une requête HTTP soit "exécutée" il se passe bien quelque chose. Oui mais cette "exécution" se passe ailleurs ! Pour votre code il ne se passe rien d'autre qu'une attente. Aucun code ne tourne sur aucun cœur du CPU…

Cette dualité Delegate / Promise Task est déroutante comme j'ai eu l'occasion de le présenté dans l'une des premières parties de cette série. Mais justement, il faut la garder sans cesse présent à l'esprit pour comprendre et utiliser Task correctement.

Les autres propriétés d'état

Task réserve des surprises… Status est un état unique, celui d'une machine à états finis qui par définition ne peut avoir qu'un seul état à la fois. Mais pour des raisons que nous ne discuterons pas ici Task offre d'autres propriétés de type booléen qui se rapprochent de certaines valeurs de Status :

```
bool IsCompleted { get; }

bool IsCanceled { get; }

bool IsFaulted { get; }
```

On comprend facilement que lorsque IsCanceled passe à True cela correspond à l'état Canceled. Il en va de même pour IsFaulted et l'état Faulted.

Mais pour IsCompleted c'est plus subtile il n'y a pas de correspondance directe avec RanToCompletion comme on pourrait s'y attendre.

IsCompleted passe à True quand la tâche passe à n'importe lequel des états "terminaux", c'est à dire aussi bien à RanToCompletion qu'à Canceled ou Faulted !

Conclusion

L'étude des propriétés de Task n'est pas la partie la plus exaltante mais elle est instructive et réserve parfois des surprises comme IsCompleted. Des subtilités qui hélas n'apparaissent pas de façon évidente sans les étudier même rapidement comme nous le faisons ici.

Mais la visite n'est pas terminée, attendons la partie 6 qui nous présentera les propriétés Id et CurrentId.

Task, qui es-tu ? partie 6

L'étude des propriétés de Task nous permet progressivement d'en comprendre les subtilités de fonctionnement. Continuons la série.

L'ID de la tâche

La propriété Id de type integer permet de connaître l'identificateur qui a été attribué à la tâche. C'est pratique en débogue pour pister les messages émis par la tâches 3 et ceux de la tâche 22 en sachant quel code à émis quoi dans le fichier de Log par exemple.

Mais contrairement à ce qu'on croit et même à ce que la documentation semble indiquer, l'ID n'est pas unique. Il l'est "presque" mais pas tout à fait. Ce dont on est sur c'est qu'il ne peut pas prendre la valeur zéro.

Ce qu'il faut en retenir c'est qu'il ne faut pas utiliser cet ID autrement que pour le debug. On pourrait être tenté par exemple d'associer l'ID via un dictionnaire à des données relatives à une tâche. Ce serait une erreur car l'unicité n'est pas garantie et cela pourrait créer des conditions de bogues particulièrement redoutables à déminer ! Il existe d'autres moyens (un peu plus sophistiqués) pour ce genre de choses. Leur étude dépasse le cadre de cette série sur Task mais j'y reviendrais car c'est intéressant.

L'ID courant

Si l'ID de la tâche n'a d'intérêt qu'en débogue que dire de ce qui apparait comme un doublon qu'est CurrentId ?

Propriété de type integer aussi avec une variante, elle est nullable.

En réalité l'astuce est dans la définition : cette propriété retourne l'ID si la tâche existe et s'exécute et null dans les autres cas… Et cela ne fonctionne qu'avec les Task qui font tourner du code sinon il n'y a pas d'exécution de quoi que ce soit et pas d'ID ni de CurrentId (donc uniquement dans le cas des Delegate Tasks, pas les Promise Tasks).

Unicité ou pas ?

Le CurrentId pourrait être utilisé comme un ID unique alors que l'ID ne le peut pas alors même que le CurrentId retourne l'ID… Well… il y a un truc qui cloche !

Oui et non.

Il faut comprendre les ID sont générés "à la demande" par type. La première Task aura l'ID 1, le premier scheduler aura aussi un ID à 1. Et qui plus est la séquence qui attribue les ID semble avoir un petit problème. Car bien que la documentation dise que l'ID est unique, on peut trouver des codes de démo qui arrivent à générer des doublons au bout de quelques minutes.

J'ai testé l'un de ces codes (ci-dessous) et en moins de 3 minutes le programme avait trouvé un doublon. Je ne suis pas convaincu qu'il s'agisse d'une erreur dans le Framework. A mon sens il s'agit d'un problème d'interprétation quant à la notion d'unicité concernant une tâche.

Les tâches qui s'exécutent doivent avoir un ID unique. Celles qui ne sont plus exécutées n'ont plus d'intérêt et leur code peut être réutilisé. Dans ce cas la documentation serait exacte (les code sont unique, pour les tâches qui tournent) et

les contradicteurs auraient aussi raison (l'unicité n'est pas absolue dans le temps si on prend en compte les tâches terminées).

```
using System;
using System.Threading.Tasks;

class Program
{
    static void Main(string[] args)
    {
        var task = new TaskCompletionSource<object>(null).Task;
        var taskId = task.Id;

        Task other;
        do
        {
            other = new TaskCompletionSource<object>(null).Task;
            if (other.Id == 0)
                Console.WriteLine("Saw Id of 0!");
        } while (other.Id != task.Id);

       Console.WriteLine("Id collision!");
       Console.WriteLine("  task.Id == other.Id: " + (task.Id ==
other.Id));
       Console.WriteLine("  task == other: " + (task == other));
       Console.ReadKey();
    }
}
```

Conclusion

Les ID ne sont pas les propriétés les plus intéressantes de Task mais ils recèlent des particularités étranges et des spécificités qu'il est bon de connaitre pour ne pas se fourvoyer…

Dans la prochaine partie nous allons voir comment on attend qu'une tâche se termine…

Task, qui es-tu ? partie 7

La patience est une qualité parait-il, savoir attendre qu'une tâche se termine doit donc rendre le code meilleur… C'est ce que nous allons voir aujourd'hui avec l'art d'attendre la fin d'une Task !

Attendre…

L'art de l'asynchronisme tient bien plus dans la façon d'attendre que d'exécuter du code. Ou plutôt dans l'art d'attendre pour gagner du temps utilisable autrement…

L'art du parallélisme est au contraire de ne pas attendre ou plutôt l'art de donner l'impression qu'il n'y a pas d'attente en utilisant les attentes créées par l'asynchronisme…

Les deux mécanismes sont donc complémentaires et se retrouvent par un heureux hasard, au sein de la classe Task.

Attendre qu'une tâche soit terminée oblige le code qui attend à ne plus rien faire, c'est bloquant. Le thread du code qui attend est suspendu au retour du code appelé. C'est pour cela que l'attente d'une Task est réservée aux Delegate Tasks et qu'elle ne s'utilise jamais avec les Promise Tasks.

Wait

La méthode Wait est d'une grande simplicité. Elle force l'attente d'une tâche, elle crée donc un point de blocage dans le thread courant jusqu'à ce que le thread de la tâche se termine.

```
void Wait();

void Wait(CancellationToken);

bool Wait(int);

bool Wait(TimeSpan);

bool Wait(int, CancellationToken);
```

Les différentes variantes de Wait déclinent les quelques possibilités mise à disposition pour gérer l'attente. L'attente se poursuit jusqu'à ce la tâche soit terminée quel que soit la façon dont cela arrive : fin normale, annulation, erreur.

L'annulation de tâche lève une exception OperationCanceledException. En revanche si un timeout a été programmé et que celui-ci est atteint Wait retournera la valeur False.

Si l'arrêt est lié à des erreurs d'exécution elles sont rassemblée par Wait dans une AggregateException.

Wait pourrait sembler une bonne solution pour rendre synchrone un code qui ne l'est pas. Ce n'est pas le bon outil pour cela … Wait s'utilise en réalité rarement. L'occasion la plus évidente est l'exécution d'un code asynchrone par la méthode Main d'une application. Dans ce cas il est évident qu'un Wait est utile puisque sinon Main se terminera ce qui mettra fin à l'application immédiatement… C'est pour cela qu'une méthode Main ne peut pas être async on plus, cela n'aurait aucun sens. Et comme elle ne peut pas être async elle ne peut pas utiliser await. C'est donc tout naturellement qu'on utilisera Task.Wait dans ce contexte précis.

Mais sinon les occasions d'utiliser Wait sont rares car pour tout code qui en attend un autre la meilleure approche est celle de async/await. Une bonne raison de plus à cela : Wait peut créer des deadlocks.

Mais await aussi si on n'y prend garde ! La raison tient en mot : le contexte. C'est un sujet important et je vous en reparlerais plus tard car je veux garder le fil…

WaitAll

On retrouve ici les mêmes possibilités que pour Wait :

```
static void WaitAll(params Task[]);

static void WaitAll(Task[], CancellationToken);

static bool WaitAll(Task[], int);

static bool WaitAll(Task[], TimeSpan);

static bool WaitAll(Task[], int, CancellationToken);
```

WaitAll est identique à Wait donc son principe et dans son fonctionnement sauf, on l'a compris, il s'agit ici d'attendre de façon groupée la fin d'un ensemble de tâches passées en paramètre.

WaitAll reste d'une utilisation assez rare. Cela peut être utile dans certains cas avec plusieurs Delegate Tasks exécutées en parallèle, mais ce n'est pas un cas si fréquent que cela. Le traitement parallèle des données est plus efficace avec PLINQ notamment. Et dans les cas où cela n'est pas adapté il est préférable de créer des tâches enfants qui ont des liens sémantiques entre elles plutôt que d'attendre un groupe de tâches créé de façon plutôt artificielle.

WaitAny

WaitAny se présente avec les mêmes variantes que WaitAll. La différence se joue dans ce qui est attendu. Avec WaitAll on attend que toutes les tâches soient terminées, avec WaitAny on sort de l'attente dès qu'une tâche se termine.

Si les cas d'utilisation de WaitAll sont rares, ceux de WaitAny le sont encore plus !

AsyncWaitHandle

Le type Task implémente l'interface IAsyncResult pour des raisons de compatibilité avec l'ancien code asynchrone basé sur APM comme nous l'avons vu dans une partie précédente. De fait Task propose aussi une propriété WaitHandle.

A moins d'avoir une grande quantité de code existant utilisant WaitHandle et de vouloir le maintenir en conservant la même logique AsyncWaitHandle n'a aucun intérêt.

Conclusion

Attendre n'est donc pas si simple… Et à moins d'avoir de bonnes raisons de les utiliser finalement aucune des méthodes de type Wait ne servent dans un code récent et bien écrit. Tout juste Task.Wait peut-il trouver quelques rares utilisations justifiées.

La prochaine partie s'intéressa notamment au résultat de Task<T>.

Task, qui es-tu ? partie 8

Task<T> est utilisé dans le cas où une tâche retourne un résultat. Il y a plusieurs façons de récupérer ce dernier mais toutes ne se valent pas…

Task et Task<T>

Récupérer les informations retournée par une Task<T> est un passage obligé, soit parce que cela coule de source (demander un résultat et ne pas en ternir compte c'est écrire du code qui peut être mis en commentaire !), soit parce qu'il faut au minimum s'assurer que tout s'est bien passé ou traiter les éventuelles erreurs. Et dans ce cas on entend ici par "résultat" tous ces différents types de résultat (une exception lue via la propriété Exception est un résultat comme un autre).

Result

Cette propriété est du type passé à l'appel de la tâche et n'existe bien entendu que dans le cas de Task<T>.

De la même façon que Wait, lire Result va bloquer de façon synchrone le thread appelant jusqu'à temps que la tâche soit complétée. Ce n'est donc pas une bonne idée que de procéder de la sorte car comme pour Wait les deadlocks ne sont pas loin !

De plus si des erreurs se produisent pendant l'accès à Result elles seront agrégées dans une AggregateException ce qui en compliquera le traitement.

Exception

Comme son nom le laisse supposer cette propriété retourne les exceptions de la tâche sous une forme agrégée. A la différence de Result et Wait la lecture de cette propriété n'est pas bloquante. Elle retourne null en permanence sauf si la tâche est terminée et qu'elle a généré des erreurs. Cette propriété se retrouve aussi bien dans Task que Task<T> bien entendu.

GetAwaiter().GetResult()

Le membre GetAwaiter a été ajouté dans Task dans .NET 4.5 uniquement pour les besoins de await mais en théorie rien n'interdit de s'en servir directement. Il n'y a que peu de différence avec Result mais elle peut avoir son importance : les exceptions ne seront pas agrégées ce qui peut être plus pratique à traiter.

await

Ce n'est bien entendu pas un membre de Task mais comme il est question des résultats retournés par une tâches il semble essentiel de rappeler que await attendra de façon asynchrone un résultat en provenance de Task (ou Task<T>) et qu'il est donc non bloquant. Dans tous les cas sauf contrainte spécifique qui resterait à définir et justifier un résultat de Task s'obtient par await. C'est de loin la méthode la plus efficace. Elle est préférable à toutes les autres, que cela soit Wait, Result, Exception or GetAwaiter.

Conclusion

Finalement l'utilisation de Task c'est facile, il ne faut pratiquement rien utiliser sauf un await sur la tâche… On ne créée pas d'instances, on await juste les données.

Il est clair qu'on est parti de loin et il y a encore pas longtemps il fallait faire de drôles de détours réservés à des spécialistes pour un code difficile à lire et à maintenir. Avec les dernières versions du Framework, Parallel, la TPL, async/await et Task, tirer parti des multicoeurs est devenu un jeu d'enfant…

Mais ce n'est pas fini, puisque dans la prochaine partie nous parlerons justement des continuations !

Task, qui es-tu ? partie 9

Si l'utilisation de Task n'est finalement pas si compliquée il n'en reste pas moins vrai que de nombreux détails sont à connaitre pour en tirer pleinement partie. Au-delà les choses peuvent se sophistiquer mais toujours sans trop se compliquer, c'est le cas des Continuations.

Continuations

Nous avons vu comment obtenir les résultats d'une tâche et comment attendre ceux-ci. Souvent des résultats sont attendus pour être traités ou complétés et ce par d'autres tâches puisque la programmation asynchrone devient une norme et une obligation technique.

Dans un tel cas allons-nous écrire des séries de await ?

Non, il existe les continuations, c'est à dire une méthode permettant de lier une tâche à une autre de telle sorte qu'elle débute immédiatement après la fin de la précédente… Pas de blocage, pas de await, rien, c'est encore mieux, en tout cas sur le papier. La tâche à laquelle on attache une autre tâche est appelée l'antécédent.

Les avantage de cette technique sont nombreux tant du point de vue stylistique, de la lisibilité du code. Toutefois là aussi un await semble plus performant et évite les problèmes de scheduler que nous verrons plus loin.

ContinueWith

Le moyen le plus direct de continuer une tâche par une autre est d'enchainer un ContinueWith. Il existe de nombreuses surcharge de cette méthode que la documentation officielle vous détaillera et qui alourdirait ce billet. Le principe est simple : on utilise ConinueWith sur à une Task en passant en général un Delegate qui sera exécuté à la suite de cette dernière.

Si on ne tient pas compte de tous les paramètres de type object, si on fait abstraction des options de type jeton d'annulation, options de continuations etc, on peut en réalité résumer toutes les variantes de ContinueWith à ces deux duos :

```
Task ContinueWith(Action<Task>, CancellationToken, TaskContinuationOptions,
TaskScheduler);
Task<TResult> ContinueWith<TResult>(Func<Task, TResult>, CancellationToken,
TaskContinuationOptions, TaskScheduler);


Task ContinueWith(Action<Task<TResult>>, CancellationToken,
TaskContinuationOptions, TaskScheduler);
Task<TContinuationResult>
ContinueWith<TContinuationResult>(Func<Task<TResult>, TContinuationResult>,
CancellationToken, TaskContinuationOptions, TaskScheduler);
```

L'un s'applique à Task, l'autre à Task<T>.

On en conclue qu'il y a deux façons de continuer une tâche,en la faisant suivre d'une autre tâche qui soit retourne un résultat (Func<..>) soit n'en retourne pas (Action<..>). Le delegate de la continuation reçoit toujours en paramètre la tâche antécédent. ContinueWith retourne à son tour une Task ce qui signifie qu'une continuation peut à son tour être continuer par un ContinueWith qui retournera une Task etc…

A noter que le dernier paramètre est le TaskScheduler qui sera utilisé pour la continuation. Malheureusement la valeur par défaut n'est pas Taskscheduler.Default mais TaskScheduler.Current. Cela semble causer pas mal de problèmes et de nombreux utilisateurs conseillent de spécifier le scheduler systématiquement en précisant Taskscheduler.Default. Le même problème semble concerner aussi Task.Factory.StartNew. Un développeur averti en vaut deux !

Au final faut-il utiliser les continuations ?

Du point de vue de l'écriture c'est certain. L'intention du développeur est conservée et visible (je veux que la tâche B soit réalisée uniquement une fois que la tâche A sera terminée). Techniquement il apparait que le gain n'est pas décisif et les petites difficultés sur le scheduler incitent à se dire que si c'est pour compliquer le code autant utiliser une série de await…

TaskFactory.ContinueWhenAny

Le principe reste le même, on attache une tâche à la suite d'une autre. Sauf qu'ici ce n'est pas une autre tâche mais un liste de tâche. Et qu'il ne s'agit pas de continuer n'importe quand mais uniquement quand l'une des tâches s'arrête, peu importe laquelle.

Les problèmes de scheduler étant les mêmes que ceux évoqués plus haut on préfèrera utiliser Task.WhenAny(…).

TaskFactory.ContinueWhenAll

Même chose que la précédente avec une nuance : la continuation n'a lieu que lorsque que toutes les tâches de la liste se sont terminées. Les cas d'utilisations semblent plus nombreux. Mais comme les problèmes de scheduler sont les mêmes, ici aussi on préfèrera Task.WhenAny(…).

Task.WhenAll

Retourne une tâche qui se termine quand toutes les tâches passées en paramètre (liste) sont terminées. par exemple :

```
var client = new HttpClient();
string[] results = await Task.WhenAll(
    client.GetStringAsync("http://example.com"),
    client.GetStringAsync("http://microsoft.com"));
// results[0] est le HTML de example.com
// results[1] est le HTML de microsoft.com
```

Ici la tâche se terminera quand les deux GetStringAsync seront terminés. Puisqu'il y a plusieurs tâches retournant un résultat celui de la tâche globale est en toute logique une liste de résultats… A la fin de l'opération comme indiqué en commentaire du code l'élément 0 de results sera le code HTML de la page example.com et l'élément 1 sera le code HTML de la page microsoft.com.

La liste de tâches étant un IEnumerable on peut passer le résultat d'une requête LINQ. Il est immédiatement réifié mais on pourra ajouter un ToArray() explicite afin de clarifier le code. L'avantage d'une requête LINQ est qu'il possible de se jouer facilement de situations complexes par exemple lorsque le nombre d'éléments est variable :

```
IEnumerable<string> urls = ...;
var client = new HttpClient();
string[] results = await Task.WhenAll(urls.Select(url =>
client.GetStringAsync(url)));
```

Ici peu importe le nombre d'URL dans la liste urls, toutes les pages seront retournées en une seule liste, en une fois et ce dès que toutes les pages auront été chargées.

Task.WhenAny

Cette méthode reprend les mêmes principes que la précédente sauf que la sortie à lieu dès qu'une des tâches passées en paramètre se termine.

Il peut y avoir un intérêt notamment quand on souhaite obtenir une information qui peut exister dans plusieurs sources. Par exemple un fichier se trouvant sur des serveurs miroirs. Ce qu'on souhaite c'est télécharger le plus rapidement, et dans ce cas on peut lancer une tâche WhenAny sur plusieurs serveurs à la fois. On récupèrera le fichier qui arrive le premier. Dans cet exemple il faut faire la part entre la logique qui est parfaite et la réalité (plusieurs téléchargements simultanés prendront de la bande passante et se ralentiront mutuellement). Mais on comprend l'idée.

```
var client = new HttpClient();
string results = await await Task.WhenAny(
    client.GetStringAsync("http://example.com"),
    client.GetStringAsync("http://microsoft.com"));
```

Cet exemple est très proche du précédent utilisé pour WhenAll, tellement qu'il est identique sauf pour l'appel de WhenAny au lieu de WhenAll.

Dans ce cas le résultat n'est plus une liste mais une tâche. Celle qui terminera la première. Ici on récupèrera le code HTML de la page HTML qui sera chargée le plus vite.

Le "double await" peut être troublant à première lecture, mais il permet de simplifier le code. S'il vous gène il suffit de spécifier les types et de séparer le code, cela revient au même :

```
var client = new HttpClient();
Task<string> firstDownloadToComplete = await Task.WhenAny(
    client.GetStringAsync("http://example.com"),
    client.GetStringAsync("http://microsoft.com"));
string results = await firstDownloadToComplete;
```

Rappelez-vous que WhenAny retourne une Task<string> dans notre cas. il y a donc nécessité de faire un await pour le WhenAny en lui-même qui est asynchrone mais aussi pour le Task<string> qui est retourné sinon on le perd. D'où les deux await dans la première version. Await qu'on retrouvent aussi dans la seconde version mais pas l'un derrière l'autre ce qui est moins troublant et décompose mieux cette "double" attente pourtant nécessaire.

Conclusion

C'est dans ces petites choses que async/await et Task peuvent devenir "tricky". On croit que c'est facile et pan! un coup sur le museau pour venir vous rappeler que ce n'est pas si évident que ça ! 

Dans certains cas on pourrait préférer lire Result pour éviter le await, mais le premier agrège les exceptions ce qui complique leur traitement alors que le second ne le fait pas. C'est une raison de préférer await.

Dans la 10me partie nous aborderons la façon de créer des Task (autrement que par leur constructeur ce qui n'est pas recommandé et vous le savez si vous avez lu les 9 parties jusqu'ici !).

Task, qui es-tu ? partie 10

Lancer des Task de type Delegate peut prendre différents chemins, nombreux sont ceux qui sont obsolètes ou qui n'ont que peu d'intérêt, finalement le choix se réduit à peu de possibilités essentielles à connaitre.

Les constructeurs

Comme nous avons pu le voir au cours de cette série l'utilisation des constructeurs de Task est exceptionnelle et sans véritable intérêt pour la vaste majorité des développements. La raison est que les constructeurs de Task retournent une tâche non planifiée (associée à aucun scheduler) et que cela n'apporte rien dans la pratique.

Que reste-t-il alors pour créer des Task de type Delegate et les exécuter ?

(la différence entre Delegate Task et Promise Task est traité dans les parties précédentes de cette série)

TaskFactory.StartNew

StartNew autorise la création et le lancement d'une tâche à partir d'un delegate sans valeur de retour (un Action) ou avec (Func<Result>). Cette méthode retourne une Task typée correctement basée sur le delegate et sa valeur de retour éventuelle.

Il faut noter qu'aucun des delegate traités par StartNew ne sont de type "async aware" ce qui occasionne des complications quand le développeur veut utiliser StartNew pour démarrer une tâche asynchrone. Alors que Task.Run sait gérer des tâches adaptées à un contexte asynchrone.

De fait StartNew est trop souvent utilisé au détriment de Task.Run qui pourtant représente une solution mieux adaptée à la majorité du code moderne.

StartNew existe en plusieurs versions proposant des paramètres plus ou moins nombreux, environ 16, pour les Action ou les Func<Result>. Si on simplifie ces différents constructeurs il reste fonctionnellement deux variantes (simple jeu de l'esprit, ces variantes n'existent pas dans le Framework) :

```
Task StartNew(Action, CancellationToken, TaskCreationOptions, TaskScheduler);
```

```
Task<TResult> StartNew<TResult>(Func<TResult>, CancellationToken, TaskCreationOptions,
TaskScheduler);
```

Dans les variantes “réelles” les paramètres non présents sont remplacés par des valeurs par défaut. Et ces dernières proviennent de l’instance de TaskFactory. CancellationToken est par exemple initialisé à TaskFactory.CancellationToken s’il n’est pas spécifié dans l’un des constructeurs utilisé. Idem pour les TaskCreationOptions ou le Scheduler.

CancellationToken

Savoir lancer une tâche est une chose, savoir l’arrêter à tout moment en est une autre. Cet aspect ne doit pas être négligé car dans une programmation moderne et fluide l’utilisateur ne doit jamais être bloqué à attendre quelque chose dont il n’a pas ou plus besoin (erreur de manipulation, changement de choix…).

Mais le paramètre CancellationToken est souvent mal compris. Il est vrai que son rôle n’est pas tout à fait intuitif.

Beaucoup de développeurs même expérimentés pensent qu’un CancellationToken leur permettra d’arrêter le delegate exécuté par la tâche à tout moment. Il est vrai que c’est à quoi on pourrait s’attendre. Mais ce n’est pas ce qui arrive… Le CancellationToken qu’on passe à un StartNew n’a de pouvoir d’annulation uniquement avant que la tâche exécute le delegate ! Dit autrement, CancellationToken dans un StartNew ne permet que de stopper le démarrage du delegate, pas celui-ici une fois lancé.

Une fois le delegate lancé CancellationToken ne sert plus à rien. En réalité il peut servir, mais ce n’est pas magique, si on veut que la tâche puisse être arrêtée par CancellationToken le delegate doit lui-même observer la valeur du token par exemple en utilisant un CancellationToken.ThrowifCancellationRequested. A partir de ce moment-là CancellationToken va fonctionner tel qu’on le pense au départ. Mais uniquement à partir de ce moment…

Comme vu plus haut, toutes les variantes de StarNew ne possèdent pas un paramètre CancellationToken qui est alors remplacé par la valeur par défaut de TaskFactory. Ce qui signifie, et c’est ce que montre les deux variantes imaginaires, que dans tous les cas un CancellationToken est passé à tâche. Soit celui qu’on indique soit celui par défaut donc.

De fait un delegate lancé par StartNew sans utiliser de variantes avec CancellationToken peut tout de même manipuler ce dernier et éventuellement annuler l’opération en cours en levant l’exception de Cancellation Requested.

Quelle est la différence entre un delegate qui agit de la sorte au sein d'un StartNew sans précision du token et un autre qui serait démarré par un StartNew précisant le token ?

La différence est subtile mais elle mérite d'être connue !

Dans le cas où le delegate observe le token et qu'il annule la tâche il lèvera OperationCancelledException. Si StartNew est utilisé sans précision du token la tâche sera retournée comme Faulted avec l'exception indiquée. Mais i le delegate lève une OperationCancelledException depuis le même CancellationToken passé à StartNew alors la tâche sera retournée non plus Faulted mais Canceled. Et l'exception est remplacée par une autre : TaskCanceledException.

Pour faire simple disons que si on prévoit d'arrêter une tâche il est nécessaire de passer un CancellationToken à StartNew et de l'observer dans le corps du delegate. Cela garantit que la tâche peut bien être arrêtée même une fois démarrée le tout avec un état de sortie correspondant à la situation (état Canceled et non pas Faulted).

Arrêt dans un code asynchrone

Certes la différence que nous venons de voir n'a pas un impact gigantesque, la tâche est arrêtée. Mais l'état de sortie de la tâche et l'exception levée sont différents. Et cela peut poser quelques soucis dès lors qu'on utilise les patterns habituels pour vérifier si une tâche a été arrêtée ou non… Pour du code asynchrone par exemple on await la tâche et on protège le code pour attraper l'exception OperationCanceledException :

```
try
{
  // "task" démarrée par StartNew, et soit StartNew soit
  // la tâche observent le token de cancellation.
  await task;
}
catch (OperationCanceledException ex)
{
  // ex.CancellationToken contient le token de cancellation,
  // si votre code en a besoin.
}
```

Arrêt dans un code synchrone

Dans le cadre d'un code synchrone await ne sera pas utilisé. A la place on appellera Task.Wait ou Task.Result. Dans ce cas on obtiendra une exception agrégée de type AggregateException dont il faudra inspecter la InnerException pour voir s'il s'agit de OperationCanceledException :

```
try
{
  // même conditions que le code précédent
  task.Wait();
}
catch (AggregateException exception)
```

```
{
  var ex = exception.InnerException as OperationCanceledException;
  if (ex != null)
  {
    // ex.CancellationToken contient aussi le token
    // comme le code précédent
  }
}
```

Token dans le StartNew ?

Au final l'utilisation du token d'annulation dans StartNew ne fait que compliquer les choses sans apporter grand chose puisqu'il n'agit que sur le démarrage. En revanche le token est utile s'il est pris en charge par le delegate lui-même, peut importe s'il a été spécifié dans le StartNew ou non.

Les effets étant légèrement différents, chacun adoptera la solution qui lui semble être la plus claire pour son code mais utiliser un token dans StartNew n'offre rien d'intéressant et dans tous les cas il ne fait pas ce à quoi on s'attend…

TaskCreationOptions

La création d'une tâche est quelque chose de finalement assez simple et direct, alors à quoi peuvent servir les options de créations ?

Tout se joue dans la création donc dans la planification de la tâche. Les options passées ici sont transmises au scheduler. Le mode PreferFairness indique à ce dernier d'utiliser une planification de type FIFO. LongRunning est une indication qui spécifie au planificateur que la tâche en question sera longue dans tous les cas ce qui permet de mieux optimiser le fonctionnement des autres tâches éventuelles. Le scheduler créera un thread spécifique pour la tâche longue en dehors du thread pool.

Le plus intéressant n'est pas dit : ces options ne sont que des indications données au scheduler et il n'y a aucune garantie qu'il les prenne en compte ni même s'il le fait de quelle façon il le fera…

Dautres options ne concernent pas le TaskScheculer dans son fonctionnement. Par exemple HideScheduler ajouté dans .NET 4.5. La tâche sera planifiée en utilisant le scheduler spécifié mais durant l'exécution de la tâche la librairie indiquera qu'il n'y a pas de scheduler courant… On suppose que cela permet de contourner une erreur assez sournoise sur le scheduler par défaut. On verra plus bas ce qu'il en est en abordant le TaskScheduler.

L'option RunContinuationsAsynchronously ajouté dans .NET 4.6 force toutes les continuations de la tâche à s'exécuter de façon asynchrone. Il s'agit de cas d'utilisation un peu limites et il difficile comme ça de trouver un exemple de l'utilité pratique de cette option, mais je suis certain que ceux qui rencontreront le besoin seront content de s'en servir !

Les options de parenté sont intéressantes car elles modifient la façon dont la tâche est reliée à la tâche en cours d'exécution. Les tâches enfants attachées changent le comportement de leur classe parent. Ce mode de fonctionnement est particulièrement bien adapté à la parallélisation dynamique des tâches. Toutefois en dehors de ce scénario assez limité ces options n'ont que très peu d'intérêt. AttachedToParent attache la tâche comme un enfant de la tâche en cours d'exécution. En réalité ce n'est pas forcément quelque chose de souhaitable, on ne doit pas pouvoir accrocher d'autres tâches à l'une des vôtres. C'est pourquoi l'option DenyChildAttach a été ajoutée dans .NET 4.5. Elle interdit d'autres tâches de devenir enfant de celle qui utilise cette option.

On notera que pour l'instant TaskFactory.StartNew utilise une valeur par défaut inappropriée pour TaskCreationOptions (qui est TaskCreationOptions.None) alors que Task.Run, dont j'ai souvent conseillé l'utilisation dans cette série, utilise une valeur désormais plus conforme aux bonnes pratiques à savoir TaslCreationOptions.DenyChildAttach. Une raison de plus de préférer Task.Run à StartNew ou autres méthodes d'exécution des Task.

TaskScheduler

On en a entendu parler souvent dans cette série.. Le TaskScheduler est un planificateur qui ordonne l'exécution des tâches lorsqu'il y a continuation. Une TaskFactory peut définir son propre planificateur qui est alors utilisé par défaut.

Attention, le scheduler par défaut de la méthode statique Task.Factory n'est pas TaskScheduler.Default mais TaskScheduler.Current. Depuis des années cela a causé beaucoup de confusion parce que la majorité des développeurs s'attendait avec raison à ce que la valeur soit TaskScheduler.Default. Il vaut mieux le savoir et en tenir compte.

Pour comprendre toute la subtilité de l'impact de ce choix étrange de la part des concepteurs de cette partie du Framework prenons un petit exemple. Le code suivant créée une task factory pour planifier un tâche à exécuter sur le thread d'UI, mais au sein de ce travail planifié une tâche de fond est aussi démarrée :

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    var ui = new
TaskFactory(TaskScheduler.FromCurrentSynchronizationContext());
    ui.StartNew(() =>
    {
        Debug.WriteLine("UI on thread " +
Environment.CurrentManagedThreadId);
        Task.Factory.StartNew(() =>
        {
            Debug.WriteLine("Background work on thread " +
Environment.CurrentManagedThreadId);
        });
    });
```

```
}
```

La sortie console va être :

```
UI on thread 8

Background work on thread 8
```

Le numéro de thread n'a pas d'importance, ce qui compte ici c'est que la tâche de fond s'exécute sur le même thread que celui de l'UI ! Voilà qui peut causer certains ennuis... notamment une lenteur de l'UI, son blocage éventuel, alors même que le développeur croit tout faire pour avoir une UI fluide !

Le problème vient du fait que le ui.StartNew tourne sur le thread de l'UI (le clic d'un bouton). De fait TaskScheduler.Current est le thread d'UI ce qui est d'ailleurs parfaitement normal et exact.

Mais en raison de la confusion Default/Current, le second StartNew qui est à l'intérieur du premier ne va pas utiliser la valeur Default mais Current du scheduler... Et par ce petit tour de passe-passe toute la belle construction de ce code réactif tombe à l'eau puisque la tâche de fond sera créée sur le thread de l'UI...

Qu'en penser ?

Task.Fasctory.StartNew ne doit finalement pas être utilisé en dehors de cas assez rares de parallélisation dynamique. A la place il est bien plus intelligent d'utiliser Task.Run qui est "la" méthode pour exécuter une tâche. Si vous avez créer votre propre TaskScheduler (une instance de l'un de ceux proposé dans ConcurrentExclusiveSchedulerPair) il y a alors un intérêt à créer votre propre instance de TaskFactory et d'utiliser StartNew dans ce contexte. Dans les autres cas, préférez systématiquement Task.Run. A moins d'être un expert super pointu vous risquez de faire plus de bêtises qu'autre chose en utilisant StartNew (d'autant qu'il ne support pas directement l'asynchronisme et qu'il faut utiliser un Unwrap de TaskExtensions qui créée un proxy de la tâche pour son utilisation dans un contexte asynchrone).

Bref, c'est bien de savoir ce que fait StartNew, c'est encore mieux d'avoir saisi qu'il vaut mieux l'éviter pour écrire du code qui marche...

Task.Run

Task.Run est finalement l'unique,le seul moyen propre et efficace dans un code moderne d'exécuter une Task qui sera empilée sur le thread pool. Cette méthode ne permet pas d'utiliser un scheduler personnalisé et propose une API plutôt simple et compréhensible n'offrant pas les risques de celle de Task.Factory.StartNew. Et puis Task.Run est async-aware ce qui est très important dans le cadre d'une

programmation moderne donc adaptée aux OS récents et aux machines qui les supportent.

Les différentes variantes de Task.Run sont les suivantes :

```
Task Run(Action);
Task Run(Action, CancellationToken);

Task Run(Func<Task>);
Task Run(Func<Task>, CancellationToken);

Task<TResult> Run<TResult>(Func<TResult>);
Task<TResult> Run<TResult>(Func<TResult>, CancellationToken);

Task<TResult> Run<TResult>(Func<Task<TResult>>);
Task<TResult> Run<TResult>(Func<Task<TResult>>, CancellationToken);
```

Simple et clair comme je le disais. On peut exécuter des delegate avec ou sans valeur de retour, un jeton d'annulation. Le tout en étant async-aware.

On notera que Task.Run ne créée pas forcément une Delegate Task (voir billets précédents de la série) mais qu'il peut dans un contexte asynchrone créer une Promise Task (lorsque le delegate passé est asynchrone lui-même). Mais on peut conserver à l'esprit que Task.Run permet d'exécuter des delegate sur le thread pool.

Bien entendu le jeton d'annulation souffre des mêmes limitations que dans le cadre de StartNew c'est à dire qu'il ne joue que sur le lancement de la tâche. On utilisera le même contournement que pour StartNew afin de d'assurer que la tâche annulée est retournée Canceled et non pas Faulted.

Task, qui es-tu ? partie 11

La partie 10 a permis de visiter les moyens de créer des tâches de type Delegate, il reste à voir comment faire de même avec des tâches de type Promise qui sont en réalité des sortes d'évènements sans code à exécuter.

Delegate vs Promise

La nuance entre ces deux types de tâche a été traitée dans cette série. Le lecteur qui prendrait cette dernière en plein milieu aura donc tout intérêt à partir depuis le premier article pour être sûr de profiter pleinement de cette édifiante lecture !

Task.Delay

Task.Delay est en quelque sorte la contrepartie asynchrone de Thread.Sleep.

Les variantes sont peu nombreuses et explicites :

```
Task Delay(int);
Task Delay(TimeSpan);
```

```
Task Delay(int, CancellationToken);
Task Delay(TimeSpan, CancellationToken);
```

L'argument entier est exprimé en millisecondes. Dans un code bien écrit on préfèrera la version avec TimeSpan qui clarifie les intentions.

Dans la pratique Task.Delay créée un timer et exécute la tâche une fois celui-ci arrivé en fin de décompte. Sauf si entre temps le jeton d'annulation a été utilisé pour annuler la tâche. Ici encore le jeton ne concerne que le démarrage de la tâche. Une fois celle-ci lancée le jeton n'a aucun effet…

L'utilité de Task.Delay est très limitée. Cela peut être intéressant pour gérer le classique problème des timeouts mais dans un contexte asynchrone. Si une opération possédant un timeout échoue on tenter de la reprogrammer pour plus tard en utilisant un Task.Delay ce qui est conforme au modèle de programmation asynchrone (au lieu d'un affreux Thread.Sleep).

Toutefois les logiques de type retry sont plutôt le rôle de librairie comme Transient Fault Handling ou Polly et ce sont ces librairies qui en interne utilisent Task.Delay, rarement le développeur directement dans son code.

Les librairies que je viens d'évoquer sont d'une extrême importance bien qu'elles soient peu connues. Elles offrent des moyens puissants et élégants pour gérer les erreurs transitoires de type timeout par exemple avec des retry, retry forever, wait etc… Pour avoir souvent écrit du code de ce type je ne peux que vous conseiller d'utiliser ces librairies car ce qui semble être un problème simple est en réalité un véritable casse-tête dont la gestion alourdit inutilement le code applicatif. Polly est un projet Github alors que TFH est un projet de l'excellente équipe de Patterns & Practices de Microsoft bien documenté comme d'habitude. Je vous conseille d'ailleurs la lecture de leur introduction à la librairie qui pose de façon claire tout le contexte des erreurs dites transitoires. Un must à connaitre (et à utiliser !).

Task.Yield

Le but de cette série n'est pas d'être exhaustif mais pratique au sens qu'elle donne des informations pour utiliser au mieux Task.

Et nous l'avons vu finalement avec Task.Run on a à peu près tout ce qu'il faut pour lancer des tâches … Tout le reste est d'utilisation occasionnelle dans des contextes hyper pointus ou bien de fausses bonnes idées dans un code moderne et réactif.

Dépoussiérer Task de toutes ces méthodes et "on dit" qui ne servent à rien est néanmoins très utile.

Dans ce cadre il faut parler de Task.Yield.

Cette méthode ne retourne pas une Task mais un YieldAwaitable, une sorte de Promise Task qui agit avec le compilateur pour forcer un point asynchrone à l'intérieur d'une méthode.

En dehors de code exemple pour tester la feature je n'ai jamais vu aucune utilisation de Task.Yield en production. C'est un truc exotique…

Certains pensent en avoir compris l'effet et se disent que c'est un moyen intéressant pour placer des points "d'air' dans une boucle par exemple afin que l'UI puisse se rafraichir (ou d'autres cas du même genre). Dans la boucle on voit ainsi apparaitre un await Task.Yield() qui fait très savant… Mais qui ne sert à rien du tout. Dans le cas de l'UI elle s'exécute sur un thread prioritaire, bien plus que les messages WM_PAINT de Windows… Donc oui il y a bien une sorte de "prise d'air" créée par le Yield mais elle ne sert à rien.

La bonne méthode pour ce genre de situation mérite d'être abordée puisque cela nous permet de rester dans le sujet des Task :

Au lieu d'un code de ce type

```
// MAUVAIS CODE !
async Task LongRunningCpuBoundWorkAsync()
{
  // Méthode appelée sur le thread de l'UI
  //  effectuant un travail CPU intensif.
  for (int i = 0; i != 1000000; ++i)
  {
    ... // CPU-bound work.
    await Task.Yield();
  }
}
```

Il faut bien entendu séparer le code long ou CPU intensif du code traité par le thread de l'UI. Ce qui devient dans une version propre qui elle fonctionne en plus, et en asynchrone :

```
void LongRunningCpuBoundWork()
{
  for (int i = 0; i != 1000000; ++i)
  {
    ... // CPU-bound work.
  }
}

// Appelé comme suit
await Task.Run(() => LongRunningCpuBoundWork());
```

En bref : n'utilisez pas Task.Yield.

Task.FromResult

Là encore on touche l'un des nombreux exotismes de Taks qui en rend l'approche douloureuse alors que tout cela ne sert pas à grand chose…

En gros FromResult permet de retourner une Task qui serait déjà terminée avant même d'avoir commencé. Cela n'existe pas. Donc à quoi cela peut-il servir ? En réalité il y un petit créneau d'utilisation : l'écriture de stubs pour le Unit Testing par exemple. Mais ce n'est pas du code de production et on peut parfaitement utiliser Task sans rien comprendre ni jamais utiliser FromResult.

Le cas d'utilisation typique et quasi unique de FromResult est celui dans lequel on a une interface qui prévoit une méthode de type async retournant une tâche mais qu'on dispose d'une implémentation synchrone qui ne fait que retourner la valeur. Pour "adapter" le code synchrone à la signature de l'interface, on utilise FromResult sur le résultat déjà calculé ce qui va créer une Task<TResult> qui conviendra parfaitement à l'interface, sa signature et les méthodes qui font un await dessus…

Même dans ce cas bien particulier il faut faire attention à des petits détails comme le fait que le code synchrone ainsi enrubanné par FromResult ne soit pas bloquant… Utilisé un code bloquant dans une méthode asynchrone réserve plein de mauvaises surprises. On peut donc utiliser FromResult pour retourner un résultat déjà existant par exemple, mais s'il doit prendre un peu de temps à obtenir alors autant respecter la signature de l'interface et implémenter un véritable code asynchrone…

Je parlais d'un cas d'utilisation "quasi unique" cela laisse entendre qu'il y aurait peut être d'autres utilisations de FromResult. Mais lesquelles ?

On en trouve une dans le cadre d'un système de cache de valeurs qui fonctionne en mode asynchrone. Dans un tel contexte on retourne des Task<T>, donc si la valeur est déjà dans le cache il faut la transformer en Task<T> bien qu'il n'y ait aucun code à exécuter, c'est une tâche terminée avant d'avoir commencée. Et si la valeur n'est pas dans le cache on appelle la méthode qui la calcule et qui elle est une vraie tâche retournant aussi un Task<T> avec du vrai code à exécuter. Ce qui pourrait donner quelque chose comme :

```
public Task<string> GetValueAsync(int key)
{
  string result;
  if (cache.TryGetValue(key, out result))
    return Task.FromResult(result);
  return DoGetValueAsync(key);
}

private async Task<string> DoGetValueAsync(int key)
{
  string result = await ...;
  cache.TrySetValue(key, result);
  return result;
```

```
}
```

On notera la présence dans .NET 4.6 d'une propriété Task.CompletedTask dont le rôle est justement de permettre le retour d'une valeur "immédiate" avec une tâche dont le statut est bien RanToCompletion, simulant parfaitement une Task<T> mais sans code à exécuter. Dans le code ci-dessus on utilisa donc plutôt un Task.CompletedTask au lieu d'un FromResult. La documentation de .NET 4.6 ne montre aucun exemple de la syntaxe et Google a beau être mon ami, il ne m'en dit pas plus, tout comme Bing. Il faudra attendre un peu pour en savoir plus…

Dans le même esprit .NET 4.6 offre des moyens de retourner des tâches en mode Faulted ou Canceled sans aucune exécution de code (FromCanceled et FromException). Cela peut être intéressant dans certaines situations où il est important de conserver un écriture asynchrone.

Conclusion

Comprendre Task est essentiel. Cette série, du moins je l'espère, vous aura montré que sont utilisation n'est pas aussi complexe que le laisse supposer les nombreuses méthodes de cette classe. S'il faut savoir à quoi sert Task.Yield, FromResult, StartNew etc, on s'aperçoit bien vite qu'au final seul Task.Run (et Run<T>) est véritablement utile au quotidien dans du code de production.

Laissez tomber les complexes face aux kékés qui à la machine à café viendront jouer les savants, maintenant vous savez ce qu'est Task et comment s'en servir… Tout est intéressant à savoir, mais ici peu est à savoir pour bien s'en servir…

J'ai bien conscience de n'avoir pas épuisé le sujet, mais ma prétention n'était pas de faire un cours sur l'asynchonisme, juste de vous parler de la classe Task. Mais j'y reviendrais forcément…

Task, qui es-tu ? partie 12 Les patterns de l'Asynchrone

Après l'étude de Task en 11 parties qui précèdent faisons un point sur les différentes approches de l'asynchronisme pour conclure. Task y joue un rôle important mais aussi async/await tout comme la bonne compréhension de l'asynchronisme lui-même…

L'asynchronisme

La programmation asynchrone est souvent mal comprise car dès le départ le terme n'est pas forcément parlant (on comprend mais on ne "visualise" pas ce que c'est) et il arrive dans un flot incroyable de librairies ou modifications du langage C# qui semblent toutes tourner autour du même problème. Le parallélisme par exemple est

une de ces notions qui vient parasiter en quelque sorte la compréhension de l'asynchronisme.

L'objectif

Posons alors le but de l'asynchronisme : Cela permet de rendre les applications "responsive" (réactives), c'est à dire non bloquantes et donnant au moins l'illusion à l'utilisateur que rien n'est bloqué ou figé.

Le moyen

Sur quoi se base l'asynchronisme pour atteindre ce but ? … Sur l'utilisation habile du temps d'attente des opérations longues qui ne sont pas réalisées par le code de l'application.

En effet, de nombreuses opérations dépendent au moins de l'OS lui-même. Lire un fichier, l'écrire sont des choses qui prennent un "certain temps" mais qui bloquent le programme jusqu'à ce qu'elles soient terminées. On pense aussi à toutes les connexions avec "l'extérieur" qui se font via des réseaux locaux ou pire via internet dont la disponibilité, la bande passante, la réactivité sont aléatoires.

En programmation classique accéder à de telles ressources bloque le code en cours car il attend la fin de l'opération pour continuer. Si j'ai besoin de lire un fichier de configuration, inutile que je passe à la ligne suivante de mon programme qui a besoin des valeurs lues tant que tout le fichier n'est pas lu.

Cela semble tellement évident ce besoin de séquentiel qu'on voit mal comment on pourrait faire autrement. Il n'est pas possible de voyager dans le temps ni d'utiliser un résultat avant qu'il ne soit disponible…

C'est vrai. Mais on peut faire autre chose en attendant… Lire un autre fichier, transmettre des informations via internet sur l'état du programme, envoyer un mail automatique à un administrateur ou milles et une choses qui n'ont pas besoin du résultat de la première lecture évoquée plus haut et qui elles-mêmes peuvent induire des attentes de ressources externes. Pourquoi attendre, pourquoi bloquer le programme alors que toutes ces attentes sont "extérieures" et que pendant celles-ci aucun code de l'application n'est utilisé ? En libérant au moins le thread principal qui gère l'UI dans les applications modernes on redonne de la réactivé à l'application plutôt que de la voir rester figée, et rien que cela change tout dans la perception que l'utilisateur a de l'environnement, de l'application et de l'OS.

Comment ?

Il y a plusieurs façons d'arriver au but. Certaines opérations peuvent tout simplement être réalisées dans des threads distincts, c'est ce à quoi sert Task le plus souvent. D'autres ne sont qu'attente par essence, il faut donc mettre en place un moyen de faire autre chose pendant cette attente (sans qu'aucun code ne soit exécuté par cette

attente), c'est le cas avec Task aussi quand on créée des Promise Tasks au lieu de Delegate Tasks (avec code exécuté).

On peut aussi rendre une application plus réactive en utilisant le parallélisme qui lui s'attache à utiliser au mieux les multiples cœurs des machines modernes. Pour cela on utilise la classe Parallel ou PLINQ par exemple. Le plus souvent il s'agit de traiter des données et en découpant le travail sur plusieurs processeurs on accélère grandement l'application la rendant plus agile, plus fluide et plus réactive aussi. Mais le parallélisme ne s'intéresse qu'à cela, traiter des données sur plusieurs cœurs. C'est son seul but. Si parmi les avantages on retrouve la réactivité globale de l'application ce n'est que par effet de bord, tout bêtement parce que l'application travaille plus vite donc semble plus performante. C'est là qu'il faut comprendre la nuance entre parallélisme et asynchronisme et on lira à ce sujet mon article

- De la bonne utilisation de async/await en C#

Aucune de ces deux approches n'est "la meilleure", au contraire, elles se complètent et doivent être utilisées en même temps pour converger vers le but final d'une *application fluide et réactive*.

Si la performance brute d'une application peut être améliorée par le parallélisme, l'asynchronisme n'a pas d'autre but que la réactivité. Et d'ailleurs le premier implique l'exécution simultanée de code sur plusieurs cœurs alors que le second n'implique en aucun cas de faire du multitâche ! Là aussi il faut comprendre cette nuance de taille (que j'ai déjà abordé plusieurs fois il est vrai, donc ça devrait commencer à rentrer !).

Bref, il existe de nombreuses façons d'atteindre le but, l'asynchronisme qui nous intéresse dans cet article fait partie de la panoplie en citoyen de première classe. Car la réactivité est avant tout affaire d'asynchronisme plus que de multitâche. Et c'est vrai, quoi qu'on fasse, multitâche et asynchronisme finissent toujours pas se rencontrer quelque part, entretenant une confusion que seule la pratique de ces deux aspects permet de lever…

Les trois patterns de l'asynchronisme

Il y en existe bien plus mais en tout cas le framework .NET nous en offre trois :

- **APM** (Asynchronous Programming Model – Modèle de Programmation Asynchrone) qui est aussi appelé pattern IAsyncResult du nom de l'interface qui joue un rôle central dans cette approche. Ce modèle est basé sur un découpage des méthodes longues en deux opérations, l'une préfixée par "Begin" pour lancer la tâche, l'autre par "End" pour récupérer le résultat de celle-ci (par exemple BeginWrite et EndWrite). Il s'agit de la première tentative de Microsoft de rationnaliser l'asynchronisme sous .NET. Ce modèle était particulièrement pénible dès que plusieurs opérations devaient s'enchaîner ou

dépendaient les unes des autres. Silverlight avec les RIA Services a donné l'occasion de s'apercevoir à quel point cette approche avait ces limites ! Bien que toujours présente dans le framework pour gérer la compatibilité des anciennes applications APM n'est plus utilisé et ne doit plus l'être.

- **EAP** (Event-bases Asynchronous Pattern – Pattern Asynchrone basé sur les Evènements) utilise une autre approche, celles de méthodes qui ont le suffixe "Async" et met en oeuvre des stratégies basées sur un ou plusieurs évènements, des types de delegate et des types dérivés de EventArgs. Cette approche a été introduite dans .NET 2.0 et là encore elle a montré ses limites et n'est plus utilisée.
- **TAP** (Task-based Asynchronous Pattern – Pattern basé sur les Tâches) utilise une seule méthode pour gérer le lancement d'une opération et la récupération de son éventuel résultat le tout de façon asynchrone. TAP est une nouveauté de .NET 4.0 et c'est aujourd'hui l'approche recommandée. Les ajouts tels que async/await font partie des améliorations de TAP pour rendre le pattern encore plus simple à mettre œuvre.

Pour bien comprendre TAP rien de mieux que de comprendre l'évolution APM/EAP/TAP.

Ici aussi je renvoie le lecteur à l'un de mes articles :

- TAP, APM, EAP : et vous, vous en êtes où ?

Les autres approches

D'autres approches ont été tentées, notamment par le biais de librairies tierces, des frameworks MVVM, etc…

En 2011, ce qui ne nous rajeunit pas mais qui prouve à quel point Dot.Blog vous donne toujours une information qui a de l'avance (pub gratos pour moi-même !) j'avais présenté différents procédés dans l'article suivant :

- Silverlight : Sérialiser les tâches Asynchrones

Si Silverlight n'est plus d'actualité, la définition de l'Asynchronisme, des co-routines, la présentation des méthodes proposées par des frameworks comme Jounce sont autant de vérités intemporelles sur la problématique de l'asynchronisme et la lecture de cet article vous permettra certainement de mieux aborder encore cette dernière en puisant aux racines de sa mise en œuvre sous .NET.

Dans la même veine et d'une même époque, j'avais aussi écrit l'article suivant :

- Appels synchrones de services : Est-ce possible ou faut-il penser autrement ?

Ici aussi il s'agissait de répondre à ce besoin d'asynchronisme par différentes approches. Le plus intéressant avec le recul est la façon dont je montre comment arrivée à "penser autrement" pour utiliser l'asynchronisme avec le pattern MVVM. C'est un sujet que j'aborderais bientôt dans le contexte ultra moderne de Universal App Platform, la relecture de cet ancien article permettra certainement de donner du relief à la problématique…

Conclusion

J'avais utilisé dans la première partie la tête du Terminator mi-humaine mi-robot pour illustrer les deux faces d'une tâche, dualité un peu inquiétante. Arrivés à la fin de cette série nous pouvons donner un peu le sourire à notre ami Schwarzy, Task ne vous fait plus peur désormais !

Cette douzième partie était surtout l'occasion de replacer Task dans une longue progression vers une modèle de programmation ultra simplifié et efficace pour mieux le comprendre. C'était aussi un moyen de pointer quelques articles qui depuis des années jalonnent Dot.Blog en distillant à chaque fois un peu plus de vérité sur l'asynchronisme. Les relire renforce certainement la compréhension de la problématique posée, les différentes approches pour arriver au modèle TAP éclairant mieux sa raison d'être. Et la chance que nous avons d'utiliser un framework et un langage qui savent évoluer !

Programmation par Aspect en C# avec RealProxy

Une bonne programmation est découplée mais de nombreux besoins comme les logs par exemple peuvent traverser les couches et n'avoir aucun lien hiérarchique avec elles. La programmation par aspect est une réponse. Et C# offre des solutions élégantes…

AOP – Aspect Oriented Programming

Je citerai juste pour rappel Wikipédia :

> *"l'AOP est un paradigme de programmation qui permet de traiter séparément les préoccupations transversales (cross-cutting concerns) qui relèvent souvent de la technique (aspect), des préoccupations métier qui constituent le cœur d'une application." (ref: wikipédia)*

On le comprend bien, l'organisation d'un logiciel bien fait se veut fortement découplée et souvent organisée selon le métier et des couches spécialisées (type DAL, BOL…). Cela fait déjà plusieurs axes de découpage plusieurs dimensions d'un espace virtuel à prendre en compte et c'est en cela que créer des architectures solides est un vrai métier. Comme souvent lorsqu'un logiciel est bien conçu et qu'on en regarde le code on peut … ne rien se dire. Ca semble propre, bien séparé, finalement c'est "naturel", "facile"… Mais tout le monde sait que *donner l'impression de la facilité et de l'aisance est ce qui réclame souvent le plus de réflexion et de savoir-faire* !

J'aime dans un tel cas cité mon ami Jean-Sébastien qui disait malicieusement "*L'orgue c'est facile, il suffit d'appuyer sur la bonne touche au bon moment*". Quelle farceur ce JS ! Tout est dans dans la "bonne" touche et le "bon" moment. Mais sinon c'est facile. Certains morceaux de Bach peuvent se jouer avec deux doigts et vous faire voyager très loin. La bonne touche au bon moment…

L'architecture d'un logiciel c'est la même chose, la bonne décision, le bon découplage, au bon endroit. Et quand cela coule de source (!) et que c'est bien réalisé, l'œil du développeur standard ne verra rien de spécial. Il aura même l'impression fallacieuse que c'est un code qu'il aurait pu écrire lui-même… En revanche celui qui sait comprend la difficulté de la démarche.

De fait bien architecturer un code en respectant la logique métier, les séparations techniques, les grands "faut pas!" comme KISS, DRY et YAGNI est déjà une affaire complexe (voir la About page de E-naxos pour la définition de ces sigles !). Mais même lorsque tout est bien géré de la sorte il reste des besoins qui dépassent ces

découpages et découplages. C'est cela que la définition de Wikipédia nous indique en parlant de "*préoccupations transversales*".

Les besoins qui sortent du cadre

Il n'est pas rare qu'en plein milieu d'un développement on ait à faire face à certains besoins qui ne peuvent pas se régler facilement par la création d'une couche supplémentaire, d'un service, d'une simple injection de dépendances. Supposons par exemple :

- Que les données envoyées à la base SQL doivent être pré-validées de façon globale (test de cohérence ou autre)
- Qu'on doive ajouter de quoi auditer le code avec plus de finesse dans certaines opérations
- Qu'il faille s'assurer de l'authentification de l'utilisateur avant chaque requête aux données (en lecture et en écriture par exemple)
- Qu'il faille mesurer avec finesse le temps de certains traitements pour les monitorer et vérifier qu'ils restent dans des bornes fixées
- Ou tout simplement qu'il faille ajouter un système de log très invasif pour pister toutes les opérations en débogue… Etc…

Chacune de ces nouvelles obligations peuvent entrainer un énorme travail avec à la clé l'une des pires horreurs, la répétition qui viole le saint DRY et rendra la maintenance particulièrement "rock'n'roll" donc couteuse…

Dans de tels cas on aimerait disposer d'un pré-processeur ou d'un autre moyen technique pour que le compilateur lui-même soit capable d'ajouter partout le code nécessaire.

Si C# ne propose pas de solutions de ce type, nous allons voir qu'il est possible de faire des miracles avec le Framework .NET et la classe RealProxy généralement fort peu connue…

La force de l'AOP

La programmation par Aspect offre de nombreux avantages mais le plus important est certainement de permettre la centralisation en un seul point de certaines

préoccupations transversales. Et ces dernières sont plus nombreuses qu'on ne le pense !

- Gérer une authentification avant chaque opération (de type filtrage pour certains utilisateurs par exemple)
- Ajouter un logging de debogue très fin (donc très répétitif en termes de code)
- Instrumentaliser le code pour surveiller ses performances
- Ajouter des évènements, des notifications ou des messages MVVM sur le changement de certaines valeurs
- Changer le comportement de certaines méthodes, etc.

L'AOP a aussi ses dangers : du code est exécuté en dehors des méthodes lorsque ces dernières sont appelées. Il y a quelque chose d'invisible, de magique qui se passe et qui peut rendre la compréhension de certains comportement voire bogue bien plus difficile. Un bogue dans la partie AOP se répètera partout et aura un impact énorme et sera délicat à pister. L'AOP a donc un cout et ce dernier doit être évaluer en fonction du gain espéré. Si on n'utilise que quelques fois l'AOP dans un logiciel mieux vaut encore répéter certains bouts de code, cela sera plus maintenable.

Comme toutes les bonnes idées architecturales ou méthodologiques *s'il peut s'agir de véritables paradigmes il ne s'agit jamais de dogmes* et chaque architecte doit être en mesure, par son savoir-faire et sa compétence, de faire les meilleurs choix dans chaque cas particulier. Et chaque application est un cas particulier.

Il existe de nombreuses façons d'implémenter l'AOP comme par exemple l'utilisation d'un pré-processeur comme je l'évoquais plus haut ou un post-processeur qui ajoute du code binaire (ou IL) à l'exécutable. On peut aussi utiliser certains compilateurs capables d'ajouter eux-mêmes le code à la compilation ou utiliser un intercepteur de code au runtime qui intercepte les instructions visées et ajoute à ce moment le comportement désiré.

Sous .NET on utilise plus généralement deux techniques : le post-processing et l'interception de code.

On retrouve la première de ces techniques dans PostSharp (qui existe en licence free limitée et en version payante couvrant plus de besoins) et la seconde dans les conteneur d'injection de dépendance comme CastleDynamicProxy et Unity.

Je ne reparlerai pas ici de ce qu'est l'injection de dépendance de très nombreux articles de Dot.Blog abordent déjà la question. Quant au post-processing tout le monde voit de quoi il s'agit je pense.

Ce qui mérite d'être noté c'est qu'en général les outils proposés reposent sur l'application d'un design pattern nommé Décorateur ou Proxy pour effectuer l'interception du code.

Les Design Pattern Décorateur et Proxy

Il ne faut trop pinailler mais ces deux patterns ne sont pas tout à fait identiques. J'aime m'en tenir à la bible en la matière, le livre de Gamma, Helm, Johnson et Vlissides, autrement appelés "le gang des quatre".

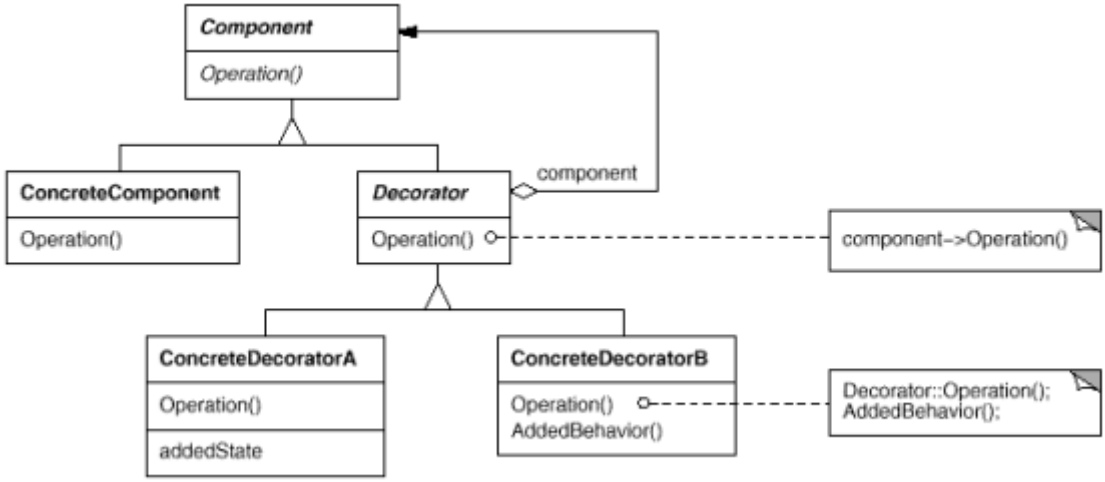
Décorateur et proxy sont deux patterns du groupe dit "*structurel*".
Les patterns de cette catégorie se préoccupent de la façon dans les classes et les objets sont composés pour former des structures plus grandes. On retrouve, outre les deux déjà citées, l'Adaptateur, le Pont, le Composite, la Façade et le Flyweight ("poids-mouche").

La grande différence entre Décorateur et Proxy se tient dans le fait que le Décorateur se focalise sur l'ajout dynamique de fonctions à un objet alors que le Proxy s'intéresse plus au contrôle de l'objet lui-même. Le Proxy est généralement un procédé compile-time (le proxy est compilé comme le reste du code et c'est lui qui se débrouille pour instancier l'objet dont il est le proxy) alors que le Décorateur est plutôt un procédé runtime, il est assigné à un objet existant durant l'exécution de l'application. On verra plus loin, la nuance entre décorateur et proxy devient encore plus équivoque si on utilise un proxy dynamique (donc runtime).

Mais il s'agit ici de généralités qui peuvent ou non se retrouver dans toutes les implémentations et utilisations possibles de ces deux patterns.

Selon le "gang of four" l'intention du Décorateur est d'attacher des responsabilités additionnelles à un objet de façon dynamique. Il est présenté comme une alternative flexible au sous-classement pour étendre une classe. La définition du Proxy est assez éloignée de prime abord puisqu'il s'agit de fournir une substitution, un remplaçant pour un autre objet pour le contrôler et y accéder. Nous verrons dans les faits que les proxy dynamiques et les décorateurs peuvent dans certains cas se confondre, voire se fondre en un troisième concept qui n'a pas vraiment de nom.

Le décorateur à la structure suivante :



Les participants sont :

Component : définit l'interface des objets qui peuvent avoir des responsabilités ajoutées dynamiquement

- ConcreteComponent : Définit un objet auquel des responsabilités sont ajoutées dynamiquement

- Decorator : Il maintient une référence sur un objet de type Component et définit une interface qui se conforme à celle de Component

- ConcreteDecorator : Les différents décorateurs réels qui ajoutent des responsabilités au composant.

Pourquoi utiliser le Décorateur ?

Pour rester concrets essayons de voir plutôt des cas dans lesquels nous ne l'utiliserions pas et quelles conséquences cela pourrait avoir :

- On peut par exemple ajouter la nouvelle fonctionnalité directement dans la ou les classes concernées. Mais cela donne à ces dernières de nouvelles responsabilités ce qui en change le sens et va à l'encontre du principe "une classe = une responsabilité".

- On peut aussi créer une nouvelle classe qui exécute la nouvelle fonctionnalité et l'appeler depuis l'ancienne classe. C'est un peu tordu mais surtout que ce passe-t-il si on souhaite utiliser la classe sans la nouvelle fonctionnalité ?

- On peut aussi hériter de la classe à enrichir et ajouter la nouvelle fonctionnalité dans la sous-classe. Mais ici on va multiplier les classes ce qui n'est pas non plus souhaitable… Imaginons que nous ayons une classe qui réalise les opérations CRUD sur une table. Si on veut ajouter du code d'audit on peut en effet la sous-classer et instrumentaliser les méthodes. D'une part il faut que la classe ne soit pas SEALED et que ces méthodes soient virtuelles ce qui pose une contrainte forte à cette méthode. Mais admettons. Plus tard on veut ajouter un mécanisme de validation des données. On sous-classe la sous-classe… Et puis un nouveau besoin d'authentification s'exprime et il faudra sous-classer la sous-classe de la sous-classe… Aie aie aie … Je ne parle même pas du choix dans l'utilisation de toutes ces classes, des morceaux de code anciens qui auront des opérations CRUD instrumentalisées mais non authentifiées et toute sorte de mélanges de ce genre. Une horreur à fuir !

- Enfin on peut décorer la classe avec le nouvel aspect. C'est-à-dire créer une nouvelle classe qui utilise l'aspect et qui appelle l'ancienne classe pour tout le reste. De cette façon si on veut ajouter un nouvel aspect on décore la classe une fois. Si on veut ajouter deux nouveaux aspects on décore la classe deux fois, etc. Pour mieux comprendre imaginons un smartphone. Il a besoin d'un emballage de présentation en magasin. On peut aussi vouloir l'offrir dans ce cas il sera emballé dans du papier cadeau avec un petit nœud. On peut vouloir le faire livrer, il sera alors emballé dans un carton avec du papier bulle, le fabriquant l'emballe dans des cartons, eux-mêmes dans des palettes, des containers pour bateau, etc. Chaque emballage s'ajoute au précédent mais peu très bien "fonctionner" seul. On peut expédier un smartphone sans emballage cadeau par exemple ou utiliser un container pour expédier une voiture. Et tous ces emballages ("décorateurs") sont indépendants les uns des autres mais aussi et surtout le sont vis-à-vis du smartphone qui peut même fonctionner sans emballage du tout… Si vous comprenez cette métaphore vous venez de comprendre le principe du Décorateur !

Implémenter le décorateur

Repartons de notre idée d'opérations CRUD utilisée pour illustrer le propos en début d'article. Je m'inspire ici du code publié dans un numéro de 2014 de MSDN

Magazine qui m'avait séduit et dont je m'étais juré de vous parler, mieux vaut tard que jamais !

On peut définir une interface générique pour toutes les entités qui supportent ces opérations, cela donnera :

```
public interface IRepository<T>
{
  void Add(T entity);
  void Delete(T entity);
  void Update(T entity);
  IEnumerable<T> GetAll();
  T GetById(int id);
}
```

Implémentons maintenant une classe qui support les opérations CRUD de IRepository<T> :

```
public class Repository<T> : IRepository<T>
{
  public void Add(T entity)
  {
    Console.WriteLine("Adding {0}", entity);
  }
  public void Delete(T entity)
  {
    Console.WriteLine("Deleting {0}", entity);
  }
  public void Update(T entity)
  {
    Console.WriteLine("Updating {0}", entity);
  }
  public IEnumerable<T> GetAll()
  {
    Console.WriteLine("Getting entities");
    return null;
  }
  public T GetById(int id)
  {
    Console.WriteLine("Getting entity {0}", id);
    return default(T);
  }
}
```

Bien entendu il s'agit d'une classe qui simule les opérations, rien ne sert de compliquer le code exemple par des accès réels à une base de données. Le repository concret pourrait utiliser le nom de la classe T pour déduire le nom de la table concernée par exemple, ou un service ou une interface supportée par les entités qui retourne ce nom de table etc. De même on laisse de côté ici la logique des transactions et autres mécanisme de connexion.

On peut dès lors utiliser notre repository concret sur la classe Customer que nous définissons de la sorte :

```
public class Customer
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Address { get; set; }
}
```

On remarque que cette classe est *totalement ignorante* des opérations CRUD.

Mais grâce au mécanisme utilisé on peut persister ou manipuler les objets de type Customer de la façon suivante (j'ai utilisé LinqPad pour tester le code, un peu de pub au passage pour cet outil génial !) :

```
Console.WriteLine("***Début de Programme***");
var customerRepository = new Repository<Customer>();
var customer = new Customer
{
  Id = 1,
  Name = "Customer 1",
  Address = "Address 1"
};
customerRepository.Add(customer);
customerRepository.Update(customer);
customerRepository.Delete(customer);
Console.WriteLine("\r\nFin du programme***");
```

Un run donnera la sortie suivante :

```
***Début de Programme***

Adding UserQuery+Customer

Updating UserQuery+Customer

Deleting UserQuery+Customer



Fin du programme***
```

Décorer pour simplifier

C'est ici que les choses deviennent intéressantes.

Vous avez créé la classe Customer. On vous a demandé qu'elle supporte des opérations CRUD mais au lieu de toucher à la classe elle-même vous avez choisi de créer IRepository<T> et sa classe concrète. C'était une bonne idée, la classe Customer n'a pas à se mélanger les pinceaux dans des problèmes de persistances sur une base SQL. Ce n'est pas sa *responsabilité*. Customer garde tout son sens et peut être utilisé dans mille contextes même ceux n'impliquant aucune base de données… Quel choix d'architecture judicieux ! On ajoutera que dans la réalité l'implémentation de IRepository<T> reposera certainement sur des classes écrites dans le DAL ou sur une couche ORM de type Entity Framework.

Mais voici que votre chef vient vous dire "*au fait machin, j'ai oublié de te dire qu'il nous faut un log de toutes les opérations pour le débogue*".

Rien de plus simple ! Au lieu de bricoler encore et encore cette pauvre classe Customer qui serait désormais méconnaissable et qui aurait perdu toute notion de sa responsabilité originelle il suffit créer un **Décorateur** !

Créons une classe LoggerRepository<T> qui implémentera IRepository<T> et qui pourra décorer directement une classe Repository<T> (la classe Customer est toujours totalement absente de ces manipulations vous noterez) :

```
public class LoggerRepository<T> : IRepository<T>
{
  private readonly IRepository<T> _decorated;
  public LoggerRepository(IRepository<T> decorated)
  {
    _decorated = decorated;
  }
  private void Log(string msg, object arg = null)
  {
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(msg, arg);
    Console.ResetColor();
  }
  public void Add(T entity)
  {
    Log("In decorator - Before Adding {0}", entity);
    _decorated.Add(entity);
    Log("In decorator - After Adding {0}", entity);
  }
  public void Delete(T entity)
  {
    Log("In decorator - Before Deleting {0}", entity);
    _decorated.Delete(entity);
    Log("In decorator - After Deleting {0}", entity);
  }
  public void Update(T entity)
  {
    Log("In decorator - Before Updating {0}", entity);
    _decorated.Update(entity);
    Log("In decorator - After Updating {0}", entity);
  }
  public IEnumerable<T> GetAll()
```

```
  {
    Log("In decorator - Before Getting Entities");
    var result = _decorated.GetAll();
    Log("In decorator - After Getting Entities");
    return result;
  }
  public T GetById(int id)
  {
    Log("In decorator - Before Getting Entity {0}", id);
    var result = _decorated.GetById(id);
    Log("In decorator - After Getting Entity {0}", id);
    return result;
  }
}
```

Rien de compliqué, LoggerRepository<T> ne fait qu'implémenter l'interface déjà définie plus haut. Seulement elle possède un constructeur auquel on passe un objet supportant IRepository<T>, objet dont les méthodes sont désormais précédées et suivies d'un log...

Comment utiliser cette nouvelle classe ? En modifiant juste une ligne de notre programme :

```
Console.WriteLine("***Début de programme***");
// var customerRepository = new Repository<Customer>();
var customerRepository = new LoggerRepository<Customer>(new
Repository<Customer>());
var customer = new Customer
{
  Id = 1,
  Name = "Customer 1",
  Address = "Address 1"
};
customerRepository.Add(customer);
customerRepository.Update(customer);
customerRepository.Delete(customer);
Console.WriteLine("\r\nFin du programme***");
```

Vous remarquez que lors de la création de l'objet repository nous utilisons désormais un LoggerRepository<Customer> auquel nous passons en référence une instance de Repository<Customer> ...

Du coup, sans rien toucher la sortie du programme devient :

```
***Début de programme***

<font color="#c0504d">In decorator - Before Adding UserQuery+Customer</font>
```

```
Adding UserQuery+Customer

<font color="#c0504d">In decorator - After Adding UserQuery+Customer</font>

<font color="#c0504d">In decorator - Before Updating UserQuery+Customer</font>

Updating UserQuery+Customer

<font color="#c0504d">In decorator - After Updating UserQuery+Customer

In decorator - Before Deleting UserQuery+Customer</font>

Deleting UserQuery+Customer

<font color="#c0504d">In decorator - After Deleting UserQuery+Customer</font>


Fin du programme***
```

C'est génial mais…

Je vous vois venir…

Je vous connais… Toujours un peu tatillon sur les bords. "*Oui, tout ça c'est très beau, encore une belle démo d'un beau principe de chercheur en blouse blanche, mais moi j'ai plein de classes et je ne vais pas réimplementer toutes les méthodes et ajouter tous les aspects et tout ce boulot de fou !*"

Et je vous entends bien. N'ayez crainte. Cela serait beaucoup de travail et compliquerait la maintenance ce qu'on cherche à éviter justement…

En fait sous .NET et grâce à cette géniale idée qu'est la réflexion qui permet à un programme de se connaitre lui-même il serait tout à fait possible d'automatiser le Décorateur et d'appliquer les logs de notre exemple à toutes les méthodes de n'importe quel objet.

C'est encore trop de travail ?

J'ai ce qu'il vous faut !

RealProxy

Le framework .NET est une merveille. Cette API objectivée a été conçue avec un tel soin et par des gens si compétents, des méthodologistes et des ingénieurs qui méritent mille fois ce titre, que même ce genre de besoin est couvert ! Tout à

l'intérieur du Framework, mais accessible à l'extérieur. A l'intérieur car les concepteurs de .NET au fait des bonnes pratiques ont utilisé le mécanisme pour leur propre code… A l'extérieur car ils ont été assez compétents pour développer quelque chose qui était exposable, pas un bricolage pour soi-même qu'on cache dans le binaire d'une sombre DLL…

Cette classe s'appelle RealProxy, présente depuis la version 1.1 au moins de .NET, c'est dire si elle est à la base même de tout l'édifice et si ce dernier a été bien conçu dès le départ.

Le principe de RealProxy est de fournir les mécanismes de base pour réaliser ce que nous disions plus haut, décorer une classe en utilisant la réflexion. RealProxy est une classe abstraite il faut donc en créer des émanations concrètes, ce qui est logique puisque .NET ne peut pas deviner toutes les décorations possibles. .NET c'est fantastique mais pas magique.

Une fois qu'on a hérité de RealProxy il suffit de surcharger sa méthode Invoke pour ajouter de nouvelles fonctionnalités à la classe décorée (aux méthodes appelées dans cette classe). On trouve cette petite perle bien cachée dans son coquillage au fin fond de System.Runtime.Remoting.Proxies.

Oui, le namespace contient "Remoting". Vous rappelez-vous de .NET Remoting ? c'était avant WCF. Et cela permettait déjà de travailler sur des instances distantes… Et forcément il fallait un proxy pour décorer les classes de l'utilisateur afin de déporter les appels aux méthodes. Objet proxy en local, objet réel sur le serveur, et RealProxy pour jouer le rôle de décorateur…

Implémentation

Reprenons l'idée de notre décorateur qui ajoute des logs à chaque méthode du Repository.

Avec RealProxy nous écrirons :

```
class DynamicProxy<T> : RealProxy
{
  private readonly T _decorated;
  public DynamicProxy(T decorated)
    : base(typeof(T))
  {
    _decorated = decorated;
  }
  private void Log(string msg, object arg = null)
  {
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(msg, arg);
    Console.ResetColor();
```

```
  }
  public override IMessage Invoke(IMessage msg)
  {
    var methodCall = msg as IMethodCallMessage;
    var methodInfo = methodCall.MethodBase as MethodInfo;
    Log("In Dynamic Proxy - Before executing '{0}'",
      methodCall.MethodName);
    try
    {
      var result = methodInfo.Invoke(_decorated, methodCall.InArgs);
      Log("In Dynamic Proxy - After executing '{0}' ",
        methodCall.MethodName);
      return new ReturnMessage(result, null, 0,
        methodCall.LogicalCallContext, methodCall);
    }
    catch (Exception e)
    {
     Log(string.Format(
       "In Dynamic Proxy- Exception {0} executing '{1}'", e),
       methodCall.MethodName);
     return new ReturnMessage(e, methodCall);
    }
  }
}
```

Dans le constructeur de la classe vous devez appeler le constructeur de la classe de base et passer le type de la classe décorée. Vous surcharger ensuite Invoke qui reçoit un IMessage en paramètre. Ce paramètre contient un dictionnaire de tous les paramètres passés à la méthode appelée. On le transtype en IMethodCallMessage afin d'extraire le paramètre MethodBase (de type MethodInfo).

Ne reste plus qu'à ajouter les aspects désirés et à appeler la méthode originale grâce à MethodInfo.Invoke. On ajoute un second aspect après l'appel pour terminer (dans le cas présent on souhaite avoir un log avant et après l'appel de chaque méthode, ce n'est qu'un exemple).

Avec la logique en place on aimerait appeler directement notre DynamicProxy<T> mais ce n'est pas un IRepository<T>, de fait on ne peut pas écrire :

```
IRepository<Customer> customerRepository =
  new DynamicProxy<IRepository<Customer>>(
  new Repository<Customer>());
```

Pour utiliser le proxy dynamique on doit utiliser sa méthode GetTransparentProxy qui retourne une instance de IRepository<Customer> dans notre cas. Chaque méthode de cette instance particulière qui sera appelée passera par la méthode Invoke du proxy. Pour simplifier tout cela il suffit de créer une factory qui cache cette astuce :

```
public class RepositoryFactory
{
  public static IRepository<T> Create<T>()
  {
    var repository = new Repository<T>();
    var dynamicProxy = new DynamicProxy<IRepository<T>>(repository);
    return dynamicProxy.GetTransparentProxy() as IRepository<T>;
  }
}
```

On notera que IMessage utilisé dans le code du proxy provient du namespace System.Runtime.Remoting.Messaging. De même que ReturnMessage.

Armez de notre factory, le code de notre application se modifie de façon très simple :

```
Console.WriteLine("***Avec proxy dynamique***");
  // IRepository<Customer> customerRepository =
  //   new Repository<Customer>();
  // IRepository<Customer> customerRepository =
  //   new LoggerRepository<Customer>(new Repository<Customer>());
  IRepository<Customer> customerRepository =
    RepositoryFactory.Create<Customer>();
  var customer = new Customer
  {
    Id = 1,
    Name = "Customer 1",
    Address = "Address 1"
  };
  customerRepository.Add(customer);
  customerRepository.Update(customer);
  customerRepository.Delete(customer);
  Console.WriteLine("***Fin***");
```

On remarque en commentaire le premier essai (création d'un Repository<T> simple), puis du second essai (avec LoggerRepository<T>) puis le nouveau code de création du repository utilisant DynamicRepository qui se base sur le code de RealProxy de .NET.

Le reste n'a pas changé d'une virgule (même si mes copies d'écran ne sont pas tout à fait identiques au fil de mes bricolages pour l'article !), la classe Customer est restée ce qu'elle est depuis le début, sa responsabilité est toujours la même, son code est clair, sa taille est très modeste (comme doit l'être toute classe). Mais Customer donne l'impression de supporter des opérations CRUD qui elles-mêmes donnent l'impression de supporter un système de log. Mais tout cela est indépendant, ne brouille pas la logique des couches en place ni ne change les classes de sens. Mieux, tout peut être utilisé seul et dans d'autres contextes. Demain on veut utiliser Customer pour faire des listes mémoire sans embarquer les namespaces propres à

SQL Server, c'est possible. On souhaite avoir un repository adapté à Oracle, c'est possible. On ne veut plus des logs ? C'est possible. Bref, on a fabriqué un édifice où tout se tient mais démontable et utilisable d'une autre façon comme les emballages de ma métaphore un peu plus haut.

Au fait, la sortie du programme modifié devient :

```
***Avec proxy dynamique***

<font color="#c0504d">In Dynamic Proxy - Before executing 'Add'</font>

Adding UserQuery+Customer

<font color="#c0504d">In Dynamic Proxy - After executing 'Add'

In Dynamic Proxy - Before executing 'Update'</font>

Updating UserQuery+Customer

<font color="#c0504d">In Dynamic Proxy - After executing 'Update'

In Dynamic Proxy - Before executing 'Delete'</font>

Deleting UserQuery+Customer

<font color="#c0504d">In Dynamic Proxy - After executing 'Delete'</font>

***Fin***
```

On a bien créé un proxy dynamique qui autorise l'ajout d'aspects au code sans avoir besoin de le répéter. Si on désire ajouter un nouvel aspect on créera seulement une nouvelle classe qui héritera de RealProxy et qu'on utilisera pour décorer le premier proxy ! On peut à l'infini décorer les décorateurs sans répéter de code ni défigurer les classes décorées.

Une nouvelle demande…

Votre chef revient… il jette un œil désabusé à votre code (ça fait longtemps qu'il ne code plus et n'y comprend rien de toute façon, mais ça fait classe de regarder avec l'air d'un connaisseur) puis vous dit "*au fait coco, je t'avais dit que seuls les admins ont accès au repository. Ah bon je te l'ai pas dit ? Bon c'est pas grave, la recette avec le client c'est dans 2h au fait.*"

Ceux qui n'auront pas suivi cet article auront toutes les raisons de paniquer. Mais pas vous !

En deux minutes voici le code d'un nouveau proxy dynamique qui refusera obstinément tout accès au repository aux users qui ne sont pas des admins :

```
class AuthenticationProxy<T> : System.Runtime.Remoting.Proxies.RealProxy
{
  private readonly T _decorated;
  public AuthenticationProxy(T decorated)
    : base(typeof(T))
  {
    _decorated = decorated;
  }
  private void Log(string msg, object arg = null)
  {
    Console.ForegroundColor = ConsoleColor.Green;
    Console.WriteLine(msg, arg);
    Console.ResetColor();
  }
  public override System.Runtime.Remoting.Messaging.IMessage
Invoke(System.Runtime.Remoting.Messaging.IMessage msg)
  {
    var methodCall = msg as
System.Runtime.Remoting.Messaging.IMethodCallMessage;
    var methodInfo = methodCall.MethodBase as MethodInfo;
    if (Thread.CurrentPrincipal.IsInRole("ADMIN"))
    {
      try
      {
        Log("User authenticated - You can execute '{0}' ",
          methodCall.MethodName);
        var result = methodInfo.Invoke(_decorated, methodCall.InArgs);
        return new System.Runtime.Remoting.Messaging.ReturnMessage(result,
null, 0,
          methodCall.LogicalCallContext, methodCall);
      }
      catch (Exception e)
      {
        Log(string.Format(
          "User authenticated - Exception {0} executing '{1}'", e),
          methodCall.MethodName);
        return new System.Runtime.Remoting.Messaging.ReturnMessage(e,
methodCall);
      }
    }
    Log("User not authenticated - You can't execute '{0}' ",
      methodCall.MethodName);
    return new System.Runtime.Remoting.Messaging.ReturnMessage(null, null,
0,
      methodCall.LogicalCallContext, methodCall);
  }
}
```

Dernière chose : modifier la Factory pour que les deux proxies soient appelés (l'un décorant l'autre n'oubliez pas ce jeu de poupées russes !) :

```
public class RepositoryFactory
{
```

```
  public static IRepository<T> Create<T>()
  {
    var repository = new Repository<T>();
    var decoratedRepository =
      (IRepository<T>)new DynamicProxy<IRepository<T>>(
      repository).GetTransparentProxy();
    // Create a dynamic proxy for the class already decorated
    decoratedRepository =
      (IRepository<T>)new AuthenticationProxy<IRepository<T>>(
      decoratedRepository).GetTransparentProxy();
    return decoratedRepository;
  }
}
```

Sans changer une ligne à l'application voici sa sortie :

```
***Avec proxy dynamique***

User not authenticated - You can't execute 'Add'

User not authenticated - You can't execute 'Update'

User not authenticated - You can't execute 'Delete'

***Fin***
```

Notre code est bien sécurisé, pas de user identifié, pas d'accès aux méthodes du repository…

Mais modifions maintenant le code de création du repository dans notre application pour fournir un user ayant un rôle d'administrateur puis un user sans rôle d'admin :

```
Console.WriteLine(

    "***Début avec authentification***");

  Console.WriteLine("\r\nROLE ADMIN");

  Thread.CurrentPrincipal =

    new System.Security.Principal.GenericPrincipal(new
System.Security.Principal.GenericIdentity("Administrator"),

    new[] { "ADMIN" });

  IRepository<Customer> customerRepository =

    RepositoryFactory.Create<Customer>();
```

```
var customer = new Customer
{
  Id = 1,
  Name = "Customer 1",
  Address = "Address 1"
};
customerRepository.Add(customer);
customerRepository.Update(customer);
customerRepository.Delete(customer);
Console.WriteLine("\r\nROLE SIMPLE USER");
Thread.CurrentPrincipal =
  new System.Security.Principal.GenericPrincipal(new System.Security.Principal.GenericIdentity
  new string[] { });
customerRepository.Add(customer);
customerRepository.Update(customer);
customerRepository.Delete(customer);
Console.WriteLine(
  "***Fin***");
  Console.ReadLine();
```

Et voyons tout de suite une sortie de ce code :

```
***Début avec authentification***


ROLE ADMIN
<font color="#0000ff">User authenticated - You can execute 'Add'</font>
<font color="#c0504d">In Dynamic Proxy - Before executing 'Add'</font>
```

```
Adding UserQuery+Customer

<font color="#c0504d">In Dynamic Proxy - After executing 'Add'</font>

<font color="#0000ff">User authenticated - You can execute 'Update'</font>

<font color="#c0504d">In Dynamic Proxy - Before executing 'Update'</font>

Updating UserQuery+Customer

<font color="#c0504d">In Dynamic Proxy - After executing 'Update'</font>

<font color="#0000ff">User authenticated - You can execute 'Delete'</font>

<font color="#c0504d">In Dynamic Proxy - Before executing 'Delete'</font>

Deleting UserQuery+Customer

<font color="#c0504d">In Dynamic Proxy - After executing 'Delete'</font>


ROLE SIMPLE USER

<font color="#0000ff">User not authenticated - You can't execute 'Add'

User not authenticated - You can't execute 'Update'

User not authenticated - You can't execute 'Delete'</font>

***Fin***
```

N'est-ce pas un peu magique ? Si forcément. L'architecture et la méthodologie sont des sciences qui ne se voient pas, on ne peut voir que du code qui met ou non en œuvre les principes qu'elles préconisent… Ce n'est que pur concept, impalpable, accessible uniquement à l'intelligence de celui qui regarde le code…

On notera que la Factory retourne toujours un IRepository<T> c'est ce qui fait que le programme fonctionne normalement même sans modification comme nous l'avons vu juste avant de l'adapter aux deux types d'utilisateur. Cela respecte le Principe de Substitution de Liskov qui pose que si S est un sous-type de T alors les objets de type T peuvent être remplacés par des objets de type S.

Dans notre cas en utilisant l'interface IRespository<Customer> on peut utiliser n'importe quelle classe qui l'implémente sans aucune modification dans le programme …

Filtrer le proxy

Il est vrai que jusqu'ici nous n'avons fait que des choses systématiques s'appliquant à toutes les méthodes d'une classe. C'est en réalité une situation assez rare. Par exemple pour notre système de log nous pouvons ne pas souhaiter monitorer les méthodes Get mais uniquement les méthodes agissant sur les données. Un des moyens simples d'y arriver est d'effectuer un filtrage sur le nom des méthodes. On remarque d'ailleurs que les bonnes pratiques de nommage rendent les choses faciles (consistance principalement, pas d'abréviation, etc). Si nous avons un GetById(int id) dans une classe un ReturnById(int id) dans une autre il sera difficile d'écrire un code générique pour le filtre…

Un tel filtrage pourrait s'écrire de la sorte (détail de la méthode invoke du proxy) :

```
public override IMessage Invoke(IMessage msg)
{
  var methodCall = msg as IMethodCallMessage;
  var methodInfo = methodCall.MethodBase as MethodInfo;
  if (!methodInfo.Name.StartsWith("Get"))
    Log("In Dynamic Proxy - Before executing '{0}'",
      methodCall.MethodName);
  try
  {
    var result = methodInfo.Invoke(_decorated, methodCall.InArgs);
    if (!methodInfo.Name.StartsWith("Get"))
      Log("In Dynamic Proxy - After executing '{0}' ",
        methodCall.MethodName);
      return new ReturnMessage(result, null, 0,
       methodCall.LogicalCallContext, methodCall);
  }
  catch (Exception e)
  {
    if (!methodInfo.Name.StartsWith("Get"))
      Log(string.Format(
        "In Dynamic Proxy- Exception {0} executing '{1}'", e),
        methodCall.MethodName);
      return new ReturnMessage(e, methodCall);
  }
}
```

Le nom des méthodes est ici testé sur le préfixe "Get". Si ce préfixe est présent l'aspect n'est pas ajouté sinon il l'est.

Il y a quelques défauts à cette implémentation, d'abord DRY… le même test est répété trois fois ce qui est beaucoup en si peu de lignes ! Mais plus gênant conceptuellement, le filtre est défini dans le Proxy ce qui rendra toute modification pénible. Les choses peuvent être grandement améliorées en adoptant un prédicat IsValidMethod comme le montre ce code :

```
private static bool IsValidMethod(MethodInfo methodInfo)
{
  return !methodInfo.Name.StartsWith("Get");
```

```
}
```

Les changements sont maintenant centralisés en un seul point. DRY est respecté. Mais il faut toujours modifier le Proxy pour modifier le filtre.

Pour régler ce dernier problème il faut complexifier un peu le code du Proxy pour lui ajouter une propriété de type Predicate<MethodInfo> et l'utiliser ensuite comme filtre. Cela donne le code suivant :

```
class DynamicProxy<T> : RealProxy
{
  private readonly T _decorated;
  private Predicate<MethodInfo> _filter;
  public DynamicProxy(T decorated)
    : base(typeof(T))
  {
    _decorated = decorated;
    _filter = m => true;
  }
  public Predicate<MethodInfo> Filter
  {
    get { return _filter; }
    set
    {
      if (value == null)
        _filter = m => true;
      else
        _filter = value;
    }
  }
  private void Log(string msg, object arg = null)
  {
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(msg, arg);
    Console.ResetColor();
  }
  public override IMessage Invoke(IMessage msg)
  {
    var methodCall = msg as IMethodCallMessage;
    var methodInfo = methodCall.MethodBase as MethodInfo;
    if (_filter(methodInfo))
      Log("In Dynamic Proxy - Before executing '{0}'",
        methodCall.MethodName);
    try
    {
      var result = methodInfo.Invoke(_decorated, methodCall.InArgs);
      if (_filter(methodInfo))
        Log("In Dynamic Proxy - After executing '{0}' ",
          methodCall.MethodName);
        return new ReturnMessage(result, null, 0,
          methodCall.LogicalCallContext, methodCall);
    }
    catch (Exception e)
    {
      if (_filter(methodInfo))
        Log(string.Format(
```

```
            "In Dynamic Proxy- Exception {0} executing '{1}'", e),
            methodCall.MethodName);
        return new ReturnMessage(e, methodCall);
      }
    }
}
```

La propriété est initialisée avec un Filter = m => true; ce qui signifie que toutes les méthodes passent le test, il n'y a aucun filtrage.

Pour revenir sur le même filtrage des "Get" que précédemment il faut désormais passer le filtre lors de la création du Proxy ce que nous pouvons faire dans la Factory pour que tout reste centralisé et transparent pour l'application :

```
public class RepositoryFactory
{
  public static IRepository<T> Create<T>()
  {
    var repository = new Repository<T>();
    var dynamicProxy = new DynamicProxy<IRepository<T>>(repository)
    {
      Filter = m => !m.Name.StartsWith("Get")
      };
      return dynamicProxy.GetTransparentProxy() as IRepository<T>;
    }
  }
}
```

Un cran plus loin

On peut souhaiter aller encore plus loin et il faut le faire… Plus le code sera générique plus il sera facilement réutilisable.

Ici notre décorateur est spécialisé dans l'ajout de logs autour des méthodes. Même avec l'astuce du filtre sous forme de prédicat cela reste toujours un ajout de log autour des méthodes.

On peut vouloir généraliser le proxy en transformant les logs en évènements. De cette façon il sera très facile de modifier l'aspect, qu'on ajoute des logs, ou qu'on fasse n'importe quoi d'autre.

Certes le code devient un peu plus long, mais le proxy devient immuable, il peut servir sans modification à toute sorte d'aspects :

```
class DynamicProxy<T> : RealProxy
{
  private readonly T _decorated;
  private Predicate<MethodInfo> _filter;
```

```
public event EventHandler<IMethodCallMessage> BeforeExecute;
public event EventHandler<IMethodCallMessage> AfterExecute;
public event EventHandler<IMethodCallMessage> ErrorExecuting;
public DynamicProxy(T decorated)
  : base(typeof(T))
{
  _decorated = decorated;
  Filter = m => true;
}
public Predicate<MethodInfo> Filter
{
  get { return _filter; }
  set
  {
    if (value == null)
      _filter = m => true;
    else
      _filter = value;
  }
}
private void OnBeforeExecute(IMethodCallMessage methodCall)
{
  if (BeforeExecute != null)
  {
    var methodInfo = methodCall.MethodBase as MethodInfo;
    if (_filter(methodInfo))
      BeforeExecute(this, methodCall);
  }
}
private void OnAfterExecute(IMethodCallMessage methodCall)
{
  if (AfterExecute != null)
  {
    var methodInfo = methodCall.MethodBase as MethodInfo;
    if (_filter(methodInfo))
      AfterExecute(this, methodCall);
  }
}
private void OnErrorExecuting(IMethodCallMessage methodCall)
{
  if (ErrorExecuting != null)
  {
    var methodInfo = methodCall.MethodBase as MethodInfo;
    if (_filter(methodInfo))
      ErrorExecuting(this, methodCall);
  }
}
public override IMessage Invoke(IMessage msg)
{
  var methodCall = msg as IMethodCallMessage;
  var methodInfo = methodCall.MethodBase as MethodInfo;
  OnBeforeExecute(methodCall);
  try
  {
    var result = methodInfo.Invoke(_decorated, methodCall.InArgs);
    OnAfterExecute(methodCall);
    return new ReturnMessage(
      result, null, 0, methodCall.LogicalCallContext, methodCall);
  }
  catch (Exception e)
  {
```

```
      OnErrorExecuting(methodCall);
      return new ReturnMessage(e, methodCall);
    }
  }
}
```

Trois évènements ont été créés : BeforeExecute, AfterExecute et ErrorExecuting. Ces méthodes sont appelées par les méthodes "Onxxxx" correspondantes qui vérifie si un gestionnaire d'évènement a été défini ou non et dans l'affirmative si la méthode passe le filtrage. Seulement dans ces cas précis l'évènement est appelé et l'aspect peut être ajouté.

La Factory devient alors :

```
public class RepositoryFactory
{
  private static void Log(string msg, object arg = null)
  {
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine(msg, arg);
    Console.ResetColor();
  }
  public static IRepository<T> Create<T>()
  {
    var repository = new Repository<T>();
    var dynamicProxy = new DynamicProxy<IRepository<T>>(repository);
    dynamicProxy.BeforeExecute += (s, e) => Log(
      "Before executing '{0}'", e.MethodName);
    dynamicProxy.AfterExecute += (s, e) => Log(
      "After executing '{0}'", e.MethodName);
    dynamicProxy.ErrorExecuting += (s, e) => Log(
      "Error executing '{0}'", e.MethodName);
    dynamicProxy.Filter = m => !m.Name.StartsWith("Get");
    return dynamicProxy.GetTransparentProxy() as IRepository<T>;
  }
}
```

On dispose maintenant d'une classe Proxy qui peut s'adapter à toute sorte d'aspects de façon fluide et transparente et qui sait même filtrer les méthodes via un prédicat totalement libre. Beaucoup de choses peuvent désormais se régler sans aucune répétition et sans casser les responsabilités des classes de vos applications !

Pas une panacée, mais un code puissant

L'intérêt du code ici présenté est qu'il ne dépend de rien d'autre que de lui même et qu'il est facilement personnalisable (en dehors d'être formateur lorsqu'on essaye de le comprendre et de le mettre en œuvre, *aspect*pédagogique plus attrayant pour les colonnes de Dot.Blog que de vous dire d'utiliser un produit tout fait…).

Dans certains cas cela peut s'avérer suffisant et même nécessaire. Il est agréable d'avoir la totale maitrise de son code source et de ne dépendre d'aucun code externe dont les évolutions restent toujours une question sans réponse absolue.

Toutefois ne nous y méprenons pas, un outil comme PostSharp est autrement plus subtile par exemple. Il agit au niveau du code IL et n'utilise pas la réflexion. Si les performances comptent alors c'est une meilleure solution !

On peut encore faire évoluer le code étudié ici par exemple en ajoutant le support des attributs pour choisir l'aspect à appliquer. On gagne forcément en souplesse d'utilisation au prix d'une petite sophistication du proxy, la méthode Invoke pouvant devenir quelque chose comme :

```
public override IMessage Invoke(IMessage msg)
{
  var methodCall = msg as IMethodCallMessage;
  var methodInfo = methodCall.MethodBase as MethodInfo;
  if (!methodInfo.CustomAttributes
    .Any(a => a.AttributeType == typeof (LogAttribute)))
  {
    var result = methodInfo.Invoke(_decorated, methodCall.InArgs);
    return new ReturnMessage(result, null, 0,
      methodCall.LogicalCallContext, methodCall);
  }
    ...
```

Conclusion

L'AOP est une approche intelligente qui minimise le code écrit et qui permet à chaque classe de conserver sa responsabilité propre sans altération. Sa façon d'agir transversalement vient compléter sans abimer les concepts habituels de séparation des couches.

On peut s'en servir de mille façons, tests, mocks, filtrage, log, etc.

Chaque projet pose ses problèmes particuliers, l'AOP permet de donner des réponses simples à des problèmes d'architecture parfois délicats.

Pensez-y et …

C# : Comment simuler des implémentations d'Interface par défaut ?

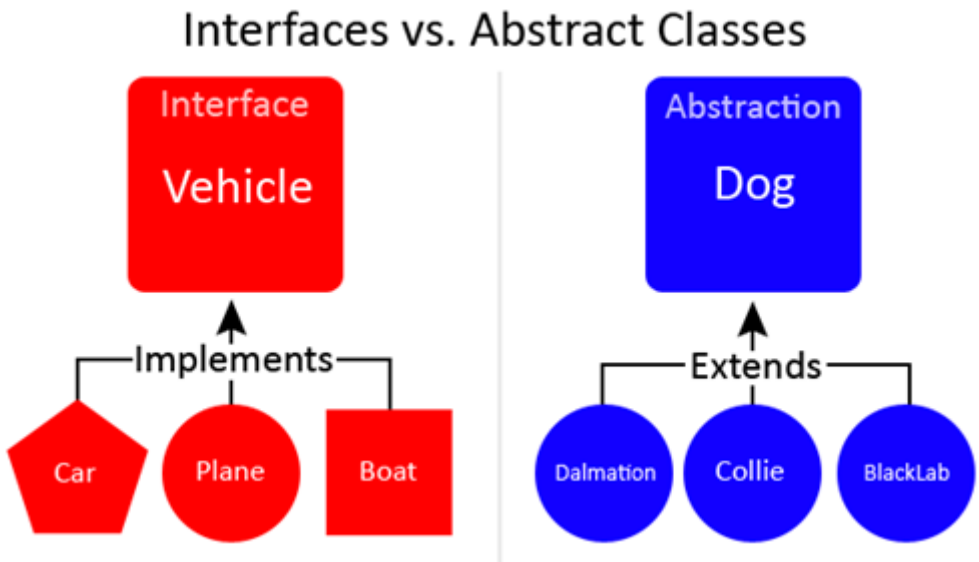
Les interfaces sont précieuses mais au contraire d'une classe abstraite elles ne peuvent offrir d'implémentations par défaut. Il existe toutefois une astuce…

Interface ou classe abstraite ?

C'est un vieux débat mais il est toujours d'actualité tant ces concepts ne semblent pas toujours clairs à de nombreux développeurs. Je ne m'appesantirai pas non plus sur le sujet, juste un rappel :

Une classe abstraite (abstract) est une classe qui force à la dériver pour l'utiliser. Il faut en effet en hériter dans une nouvelle classe "normale" pour pouvoir ensuite créer des instances. Une classe abstraite propose des méthodes, des propriétés et généralement, au moins pour certaines, des implémentations dites "par défaut". Si une méthode contient du code et qu'elle n'est pas overridée, la nouvelle classe bénéficiera de ce code déjà écrit.

Une interface ne fait que fixer un contrat, ce n'est qu'une description de ce dernier. Pas moyen d'offrir un code par défaut pour les éléments du contrat. Si dix classes supportent une interface, dix fois il faudra écrire le code pour supporter le contrat.



Si on parle de choix entre ces deux approches c'est parce qu'elles semblent partager quelque chose qui les rend, ou tend à les rendre, similaires. Cette "chose", c'est la notion de contrat. L'interface décrit un contrat sans l'implémenter, un classe abstraite décrit elle aussi un contrat mais en proposant généralement une implémentation. Elle gère aussi d'autres aspects essentiels comme fixer certains mécanismes (ordre d'exécution des méthodes pour accomplir une tâche particulière par exemple). La

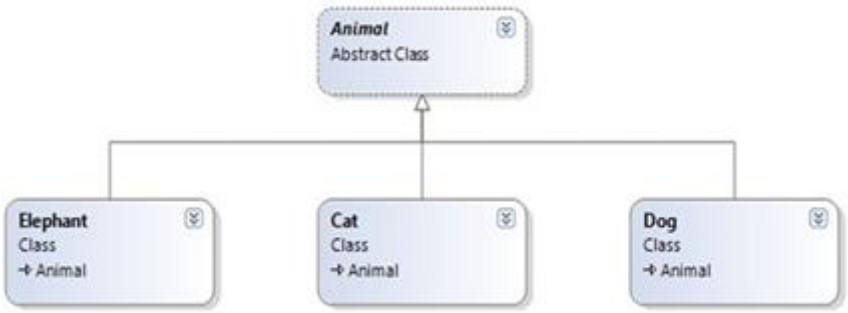
classe abstraite est bien plus riche qu'une interface dans le sens où elle fixe deux types de contrats à la fois : un contrat public (comme l'interface) et un contrat privé (garantie de certains traitements).

Alors pourquoi choisir ? La classe abstraite ne serait-elle pas "mieux" dans tous les cas ?

Il n'en est rien car s'il existe des similitudes entre les deux approches, il existe aussi une divergence de taille : la nature même du contrat est totalement différente.

L'héritage imposé par la classe abstraite créé un lien de type "est un". Si A est la classe abstraite, et B une classe dérivée, par la nature même de l'héritage on pourra affirmer que "B est un A". Prenons la classe Animal et la classe Chien qui en hériterait on peut en effet affirme que "un Chien est un Animal", ou plus précis "une instance de classe Chien peut être considérée comme une instance de la classe Animal".

Un interface ne crée pas la même proximité avec les classes qui l'implémentent. Une classe A qui supporte l'interface I, n'a aucune relation particulière avec I en dehors du fait qu'elle supporte I. Sa nature même le fameux "est un" n'est pas impacté par cette relation très distendue entre l'interface et ses implémentations.



Le choix entre interface ou classe abstraite ne doit donc pas se faire sur le plan de ce qui serait le "mieux" (ce qui ne veut de toute façon rien dire) sous-entendu qu'une classe abstraite par sa plus grande richesse serait un avantage en vertu du principe du "qui peut le plus peut le moins". Le choix doit s'effectue sur les implications de la nature même du contrat imposé par les deux approches. Avec une classe abstraite on impose un héritage et une relation "est un", avec une interface on s'engage juste à respecter un contrat parmi d'éventuels autres tout en se gardant la possibilité de "être un" quelque chose qui a de la signification (fonctionnelle, métier…).

Bref, si les interfaces sont de plus en plus utilisées c'est bien parce qu'elles n'imposent que peu de choses. Elles permettent ainsi d'éviter de polluer la logique d'héritage d'un code par un héritage technique qui interdira de définir des arborescences chargées de sens pour l'application. Les interfaces offrent aussi un moyen simple d'écrire un code à faible couplage, ce qui est visée par toutes les approches modernes de développement (type MVVM, MvvmCross ou autres).

Dès lors choisir de créer des interfaces est dans 99% des cas une meilleure stratégie que d'imposer un héritage technique en créant des classes abstraites.

Certes… Mais quid des implémentations par défaut ?

Implémentations par défaut

Les interfaces n'ont ainsi que des avantages, sauf un : elles ne permettent pas de proposer des implémentations par défaut.

On voit ainsi parfois des développeurs créer une classe qui supporte une interface en offrant une implémentation de base. Ne reste plus qu'à hériter de cette classe pour bénéficier des implémentations… Mais ce stratagème ne trompe personne, on retrouve ici tous les défauts de l'héritage et de l'utilisation d'une classe abstraite !

Comme il n'y a pas de magie et puisqu'il faut forcément une classe pour implémenter un code, fournir du code par défaut pour les membres d'une interface oblige à créer une classe pour le faire.

Cette vision des choses nous laisse malgré tout quelques latitudes qu'il ne faudrait pas négliger…

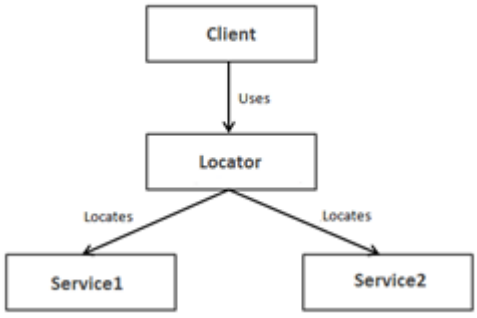
La notion de service

En effet s'il faut une classe pour implémenter le code du contrat d'une interface personne ne dit que c'est "votre classe" qui doit le faire !

C'est tout l'intérêt de l'approche par service. On définit une interface puis on créée une classe d'implémentation qui la supporte et on ajoute une instance de celle-ci dans un système d'injection de dépendances ou dans un conteneur d'IoC (inversion de contrôle) par exemple Unity.

Les classes qui ont besoin des méthodes de l'interface utilisent celle-ci tout simplement, les instances se chargeront de demander ou de recevoir l'instance qui rend le service.

Cette façon de pratiquer permet de créer un code à faible couplage ce qui en garantit une bien meilleuremaintenabilité et la possibilité de changer à volonté les implémentations du service sans impacter sur les classes qui en font usage.



Largement utilisée par les framework cross-plateformes pour sa souplesse, la notion de "service" est une façon intelligente et efficace de

proposer une implémentation pour une interface, implémentation économe puisque codée une fois et utilisable potentiellement à l'infini par autant de classes qu'on désire sans créer aucun lien de dépendance pas même celui de supporter l'interface en question.

Si cette approche est séduisante à plus d'un titre – je ne lui connais pas d'inconvénients même mineurs – elle n'est pas forcément applicable à toutes les situations et puis, vis à vis de la question posée, ce n'est qu'un ersatz, une ficelle un peu grosse car il ne s'agit pas tout à fait de ce qu'on entendait par une implémentation d'interface par défaut. Alors ?

Les méthodes d'extension à la rescousse

Les méthodes d'extensions sont l'une des facettes de la richesse de C#. Ici ce sont plutôt des librairies comme LINQ qui en font grand usage pour s'intégrer plus facilement dans le code. On les retrouve aussi à l'œuvre dans P-LINQ et dans bien d'autres situations.

Le côté pratique des méthodes d'extension c'est de faire croire qu'une instance supporte des méthodes qui en réalité n'existent pas dans sa classe et qui seront traitées par un code qui se trouve "ailleurs".

Un code "ailleurs" qui implémentent des méthodes ? Cela ressemble beaucoup à la notion de service je vous l'accorde mais aussi à celle d'une implémentation d'interface par défaut …

Avec un peu de créativité on peut détourner cette proximité fonctionnelle pour enfin s'approcher au plus près de ce qu'on attend.

Voici un exemple simplifié de l'utilisation de cette technique ce qui rendra le propos plus éloquent :

```
interface ITruc
{
}

static class ITruc_Extensions
{
    public static void Machin(this ITruc self) { Console.WriteLine("Truc
!"); }
}

class ImplementingClass : ITruc
{

}


class MainClass
{
```

```
    public static void Main (string[] args)
    {
        var aTruc = new ImplementingClass ();
        aTruc.Machin();
    }
}
```

Comme on le voit ci-dessus la méthode Machin est en réalité fournie par une classe qui étend l'interface ITruc (et non pas ImplementingClass). Le fait que la classe qui supporte ITruc puisse être vue comme un ITruc permet d'appeler les méthodes d'extension de cette interface. L'instance aTruc peut ainsi appeler Machin uniquement parce que sa classe supporte ITruc… Le tout sans écrire le code correspondant.

C'est une feinte qui peut paraitre un peu grosse puisqu'en réalité ITruc ne définit rien du tout, ce n'est qu'une sorte de marqueur qui permet aux classes qui indiquent la supporter de bénéficier des méthodes d'extension conçue pour elle.

Mais fonctionnellement on a bien l'impression que la classe supporte ITruc et quelle possède un code par défaut pour les méthodes de ITruc (qui n'en a en réalité aucune, le contrat vide et se trouve déporté vers la classe implémentant les méthodes d'extension).

L'avantage est que si la classe Implementing Class veut fournir sa propre implémentation cela reste tout à fait possible puisque ITruc est vide… Fonctionnellement on a bien le support d'une interface offrant des implémentations par défaut qui peuvent être surchargée ! Rien n'interdit donc de définir une méthode Machin dans la classe ImplementingClass car dans ce cas c'est la méthode locale qui sera appelée car elle a la préséance sur la méthode d'extension. Futé non ?

Mais que se passe-t-il si on souhaite obliger l'implémentation de certaines méthodes du contrat ? Avec l'astuce présentée il suffit tout simplement de placer la définition dans l'interface au lieu de la laisser vide…

Imaginons qu'en plus de `Machin`() nous souhaitions ajouter à l'interface Bidule() mais que son implémentation soit forcée (donc sans implémentation par défaut), il suffit d'ajouter "public void Bidule();" dans ITruc. Et ImplementingClass se verra obligée d'implémenter `Bidule`() alors que l'implémentation de Machin() sera optionnelle. Krafty, isn't it ?

Conclusion

Bien entendu on ne peut pas modifier C# pour lui faire faire des choses qu'il ne propose pas. Il n'existe pas de moyen de fournir ce qui serait de "vraies" implémentations par défaut pour les interfaces.

Toutefois en faisant preuve d'un peu d'originalité et de créativité on peut utiliser la très grande richesse du langage pour simuler au plus près ce concept.

C# Scripting ajoutez de la vie dans vos applications !

C# ce merveilleux langage… On aimerait souvent pouvoir l'utiliser comme outil de scripting dans les applications. Grâce à CS-Script cela est tout à fait possible. Gratuitement…

Pourquoi "Scripter"

Comme je l'indiquais en introduction, l'une des principales utilisations du scripting est de pouvoir étendre le paramétrage et ajouter des extensions à des applications pour les rendre plus souples et plus versatiles. Ce mode d'extension réclame souvent un niveau que l'utilisateur final ne possède pas et il reste plutôt utilisé par les développeurs eux-mêmes pour personnaliser un logiciel aux besoins spécifiques d'un client.

Néanmoins lorsqu'on s'adresse à une population d'utilisateurs "experts", la mise à disposition d'un scripting de qualité a toujours été l'apanage des "grands" logiciels. Les macros de Word ou Excel, les extensions en python de Blender, les scripts Photoshop, etc, quel que soit le langage choisi tout logiciel d'une certaine envergure possède un mode scripting.

Scripter peut aussi être utile pour créer des extensions à la Console ou simplement tester facilement des bouts de code sans sortir tout l'arsenal officiel dont Visual Studio ou Xamarin Studio…

CS-Script

CS-Script est parti d'un article dans Code Project en septembre 2009. Une sorte de POC, d'exercice de style.

Mais l'intérêt était tel que ce petit projet a vite évolué et est devenu un véritable outil utilisé un peu partout dans le monde pour étendre des applications ou pour servir d'environnement de test de code rapide.

Aujourd'hui c'est un projet Open-Source sous licence MIT l'une des plus permissives qui existe puisque tout est permis sauf d'oublier d'ajouter un lien vers la licence MIT. Bref cela est un peu comme s'il n'y avait pas de licence mais en réalité elle existe et protège les auteurs en leur donnant le droit de faire ce qu'ils veulent du code source tout en ne conférant aucun droit de réclamation à l'utilisateur. Cela n'est pas forcément légal dans le cadre juridique Européen et Français surtout s'il y a vente d'un produit. Mais ces aspects juridiques ne sont pas l'essentiel de mon propos d'aujourd'hui.

Techniquement ?

CS-Script se présente comme l'implémentation d'un système de scripting basé sur la définition ECMA de C# basé sur un CLR (Common Language Runtime).

Actuellement CS-Script cible les implémentations Microsoft du CLR (.NET 2.0; 3.0; 3.5; 4.0; 4.5) ainsi que Mono.

Dans la pratique ?

Dans la pratique CS-Script se présente sous la forme soit de binaires soit de code source.

Beaucoup de produits gratuits ou commerciaux l'utilisent, comme par exemple MediaPortal, portail de l'Open Source, FlashDevelop, qui est un EDI de développement Open Source, etc.

On trouve même, et c'est là que cela devient intéressant au quotidien, une extension pour NotePad++ qui permet d'obtenir depuis ce dernier un environnement totalement fonctionnel pour écrire et exécuter du C# !

CS-Script + Notepad++

Nodepad++ est une super éditeur de code externe, c'est à dire autonome (en opposition aux éditeurs intégrés à Visual Studio par exemple). C'est plutôt un bloc-notes Windows qui aurait été adapté à la saisie de code. Aussi facile et rapide à invoquer et utiliser que le bloc-notes mais presque aussi complet qu'un éditeur d'EDI professionnel.

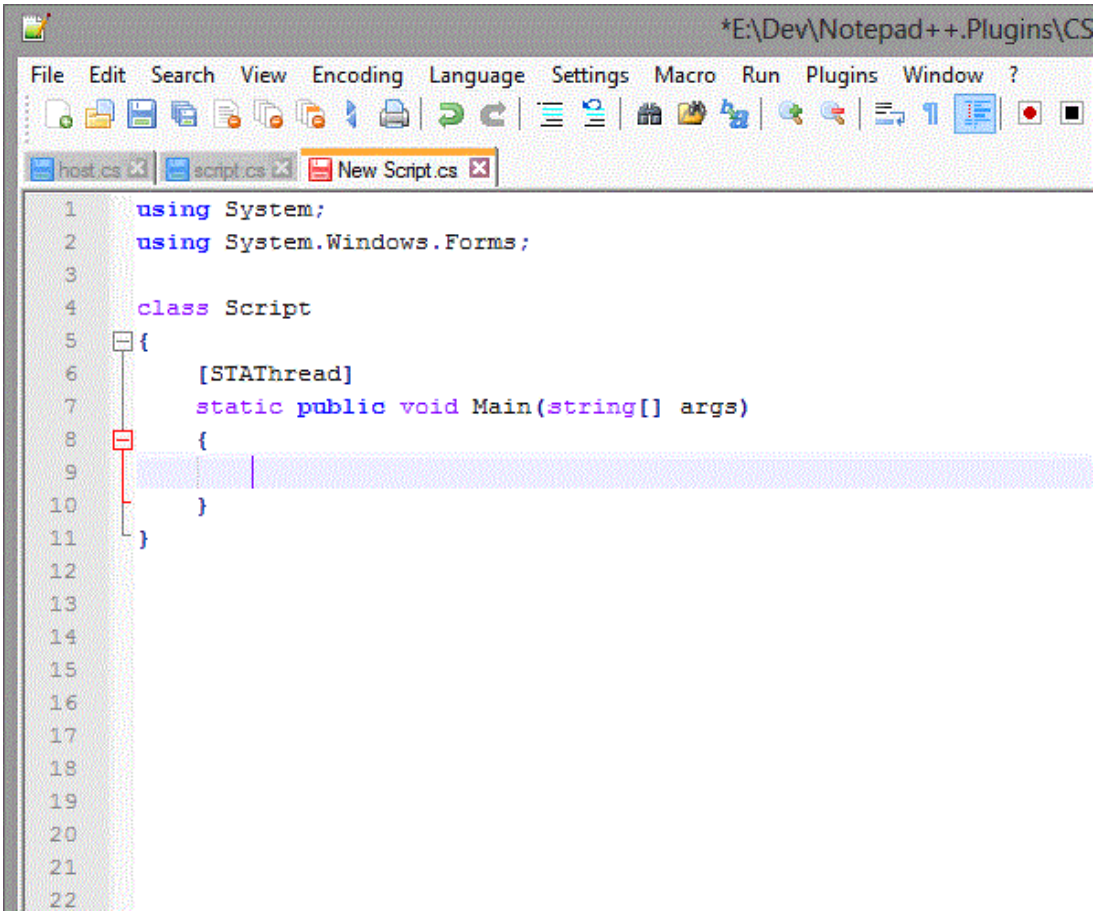
J'ai toujours utilisé Notepad2 pour remplacer le bloc-notes de Windows car il sait faire du syntax highlighting ce qui est pratique quand on ouvre un fichier html, xaml, csharp, etc. Mais depuis Notepad++ est bien plus complet et surtout il s'ouvre au mariage avec CS-Script pour devenir un mini EDI C# exécutant immédiatement le code…

Une fois Notepad++ installé (il existe des modes d'installation par simple copie fournis en Zip, donc pas de truc embrouillé qui en colle de partout sur le PC…) il suffit d'installer l'extension CS-Script pour Notepad++. Le site de cette extension se trouvant ici : cs-script.Npp. Notepad++ gère un petit système de plugin à la Nuget ce qui facilite l'installation des extensions (nombreuses dont certaines vraiment bien).

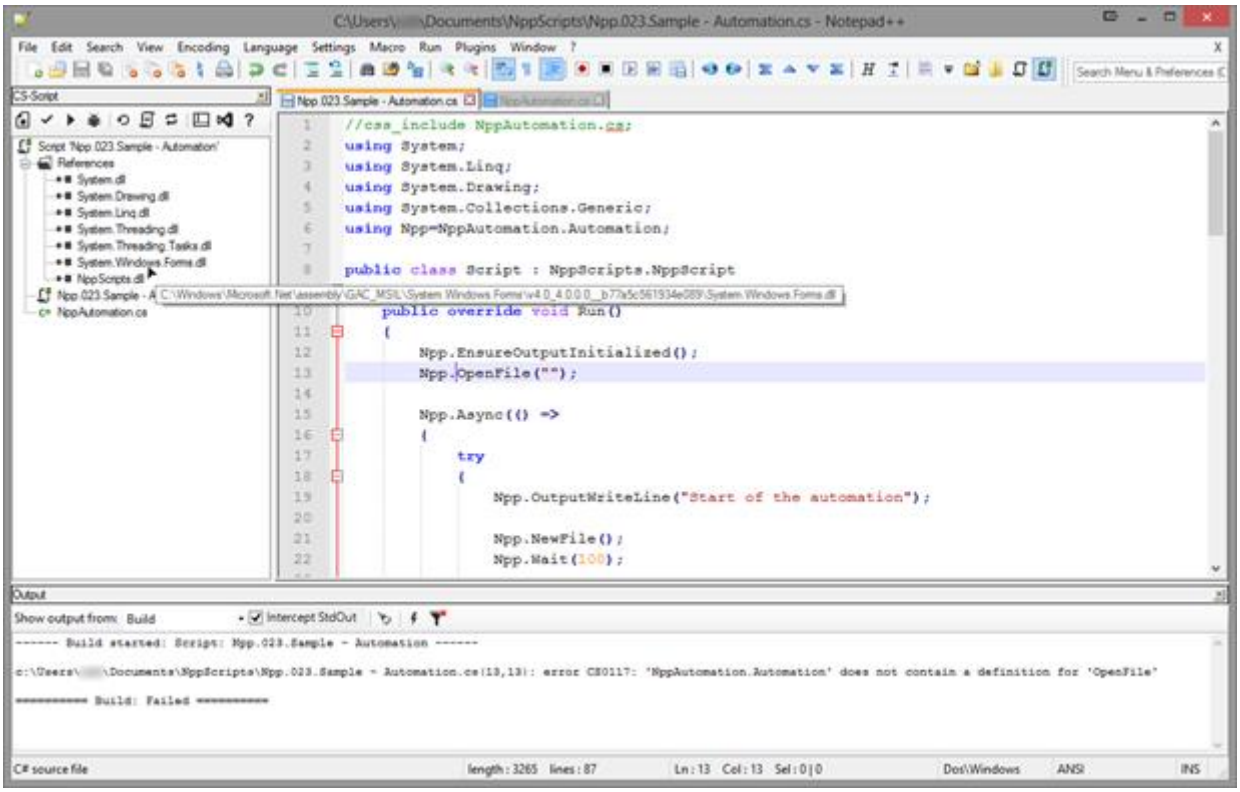
L'avantage est bien entendu de disposer d'une petite application aussi légère qu'un bloc-notes, rapide à exécuter, gérant la syntaxe de C# et la mise en forme du texte parfaitement mais intégrant en plus un module d'exécution du code tapé ! Une sorte

de mini Visual Studio de poche ultra pratique pour tester une idée, un bout de code, codé sur une machine qui n'a pas VS, etc.

L'extension actuelle ajoute même Intellisense et c'est là que les choses deviennent vraiment pratiques car le framework est tellement vaste qu'on a du mal à tout retenir de chaque namespace, de chaque classe… Le petit GIF ci-dessous montre tout cela en mouvement (c'est plus sympathique !).



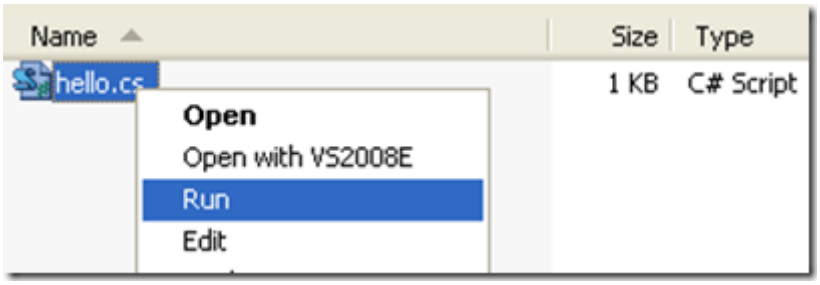
Le système est tellement bien conçu et complet qu'on obtient à l'arrivée plus qu'une sorte de console pour tester trois lignes de C# mais bel et bien un environnement de développement gratuit, prenant peu de place, et fournissant l'essentiel pour travailler ou tester des idées sophistiquées. D'ailleurs quand on regarde le look final de Notepad++ intégrant le mode CS-Script on dirait vraiment un EDI classique :



Mais je ne vais pas tout vous faire voir, je vous laisse le plaisir de découvrir tout ça par vous-mêmes !

Exécution en mode command-prompt

CS-Script peut aussi exécuter du code C# avec un clic droit dans l'explorateur de fichiers Windows, cela est très pratique dès lors qu'on se créée des librairies de code de test par exemple :



Script hosting

Beaucoup de lecteurs seront certainement plus intéressés par cette utilisation de CS-Script : ajouter du scripting dans une application !

En réalité c'est d'une facilité déconcertante. Une fois la librairie placée dans les références du projet on peut exécuter tout code scripté et recueillir les résultats éventuels en quelques instructions. Voici quelques exemples :

Exécution d'un script contenant la définition d'une classe

```
dynamic script = CSScript.Evaluator
                         .LoadCode(@"using System;
                                     public class Script
                                     {
                                         public int Sum(int a, int b)
                                         {
                                             return a+b;
                                         }
                                     }");
int result = script.Sum(1, 2);
```

Exécution d'un script ne contenant qu'une définition de méthode

```
dynamic script = CSScript.Evaluator
                         .LoadMethod(@"void SayHello(string greeting)
                                       {
                                           Console.WriteLine(greeting);
                                       }");
script.SayHello("Hello World!");
```

ou bien

```
ar Product = CSScript.Evaluator
                     .CreateDelegate(@"int Product(int a, int b)
                                       {
                                           return a * b;
                                       }");
int result = (int)Product(1, 2);
```

ou encore

```
int result = (int)CSScript.Evaluator.Evaluate("1 + 2");
CSScript.Evaluator.Run("using System;");
CSScript.Evaluator.Run("Console.WriteLine(\"Hello World\\\")");
```

Vous retrouverez ces exemples et bien d'autres informations sur le site de CS-Script.

Les scripts peuvent être stockés sur disque, dans le cloud, être générés à la volée, tapés par l'utilisateur, sauvegardés dans une base de données, bref tout est possible et ce sont bien des horizons totalement nouveaux qui s'ouvrent !

Notepad++/CS-Script vs LinqPad

On ne peut s'interdire une telle comparaison. Les deux approches sont similaires : un exécutable et ses dll utilisable sans besoin d'une installation complexe, exécution et débogue de code C# sous .NET, etc.

On notera tout de même les différences suivantes :

- Notepad++ et CS-Script sont totalement Open Source et gratuits, LinqPad est propriétaire, les sources ne sont pas disponibles.
- Npp/CS offre Intellisense là où il faut acheter une licence Pro ou Premium de LinqPad pour bénéficier de cette aide indispensable.
- LinqPad reste très orienté LINQ et est parfaitement adapté aux tests de requêtes de ce type. Il sait même utiliser les dll Entity Framework de vos projet pour autoriser des tests directs de ces dernières.
- LinqPad n'offre aucun support de type scripting et code le code est fermé on ne peut pas s'en inspirer là où CS-Script peut s'intégrer à vos applications.

Peut-on départager les deux produits ? J'en doute. Chacun a ses forces et faiblesses. Pour faire des tests rapides de C# les deux se valent même si à l'utilisation Npp+CS-Script grâce à Intellisense s'avèrent plus pratiques. LinqPad le fait mais il faut alors la version payante…

Pour tester du LINQ, LinqPad est supérieur.

Mais si on ne parle que de scripting, SC-Script prend l'avantage puisqu'il est utilisable en dehors de Notepad++ là où le module intégré à LinqPad n'est qu'un appel aux API du compilateur Microsoft.

Conclusion

On a souvent besoin de tester un bout de code. LinqPad est très utile pour cela, mais Notepad++ avec l'extension CS-Script le fait aussi bien avec Intellisense en prime et en offrant les possibilités d'un véritable éditeur de texte permettant de travailler aussi sur du XML, du JSON, etc.

Mais Notepad++ n'était pas mon propos ici mais SC-Script. Et là ne parlons plus de rien d'autre car il n'y a pas d'équivalence gratuite aussi bien achevée que SC-Scripting pour ajouter un mode script aux applications. Notepad++ et CS-Script sont deux produits différents et ils peuvent parfaitement s'utiliser l'un sans l'autre, il est utile de le préciser.

Mais d'une pierre deux coups, d'un côté vous vous offrez pour rien un mini EDI pour tester du C#, de l'autre un système de scripting super efficace pour étendre vos applications !

Ces produits sont tous dans ma boîte à outils et je m'en sers très régulièrement. Si cela peut vous rendre service aussi alors ce billet aura atteint son objectif !

C# Quick Tip

Vite fait, un peu de C# et un rappel sur la façon d'écrire du code de type "inline" ou de recréer les sous-fonctions du Pascal qui n'existaient pas avant la V7. Ce billet a été écrit avant cet ajout et reste intéressant en tant qu'exercice syntaxique utilisant les Lambda.

Quick c'est quick, c'est même fast

Pas de blabla, il suffit de regarder le code ci-dessous avec un exemple de la sortie produite pour comprendre tout l'intérêt de ce style d'écriture. Définir une fonction (ou méthode) dans une autre est certainement la chose la plus intéressante de mon point de vue. Ancien Pascalien, j'ai toujours trouvé dommage qu'il faille déclarer une méthode private de plus juste pour découper un algorithme là où des sous-fonctions sont milles fois plus efficaces. Polluer une classe de méthodes utilisées une seule fois dans une seule méthode s'éloigne de ma conception d'une classe bien écrite.

Au fil de ses évolutions C# s'est tellement sophistiqué qu'on peut presque tout faire, même des sous-fonctions à la Pascal. Ce qui est même le cas depuis la V7 non prise en compte ici.

Bref voici le code copié tout droit de LinqPad dont on ne dit jamais assez de bien :

```
//calcul
var add = new Func<int,int,int>((i,ii)=>{return i+ii;});
var a1 = add(add(add(1,2),3),4);
Console.WriteLine(add(5,8)+" "+a1);

//chaînes
var normalize = new Func<string,string>((s)=>{return s.Trim().ToUpper();});
var c="    test   ";
Console.WriteLine(">"+normalize(c)+"<");

// dicos
var dico = new Func<Dictionary<int,string>>(()=>
{
    return new Dictionary<int,string>
    {
        {0,"zéro"},
        {1,"un"},
        {2,"deux"},
        {3,"trois"}
    };
})();

Console.WriteLine(dico[2]);

// multiplier + tostring
Func<int,int,string> AddToStr = (i,ii) => (i+ii).ToString();

var x = "5+3= "+AddToStr(5,3);
Console.WriteLine(x);
```

```
// sans param et chaînage
Func<string> TenBlankChars = ()=> new string(' ',10);
var k = "1"+TenBlankChars()+"2";
Console.WriteLine("k: '{0}' Longueur: {1}",k,k.Length);
Func<string> OneCol = () => TenBlankChars()+"|";
Func<string> TenCol = () => string.Concat(Enumerable.Repeat(OneCol(),10));

Console.WriteLine(TenCol());
```

Et voici la sortie :

```
13 10

>TEST<

deux

5+3= 8

k: '1          2' Longueur: 12

          |          |          |          |          |          |          |          |
```

Conclusion

Quick, c'est quick.

Pas de révélation choc ici, mais juste un petit rappel de la puissance de C# car je ne vois que très rarement du code utiliser ce type d'écriture pourtant très efficace et évitant souvent la création de méthodes privées utilisées une seule fois et qui polluent les classes.

Si vous le saviez, et bien servez-vous en ! Et si vous aviez zappé cet aspect de la syntaxe de C#, adoptez-le sans tarder !

Mais encore mieux désormais, si le but est d'écrire des méthodes locales, utilisez la nouvelle feature de C# 7 ! Néanmoins jongler avec les Lambda peut s'avérer très utile dans bien d'autres cas raison pour laquelle j'ai maintenu ce billet dans cette troisième édition du présent ouvrage.

Débogue

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 :

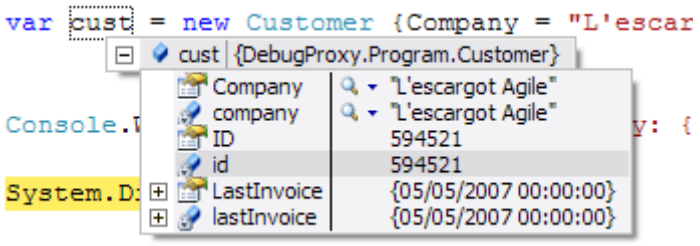
```
public class Customer
        {
            private string company;
            public string Company
            {
                get { return company; }
                set { company = value; }
            }

            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 !

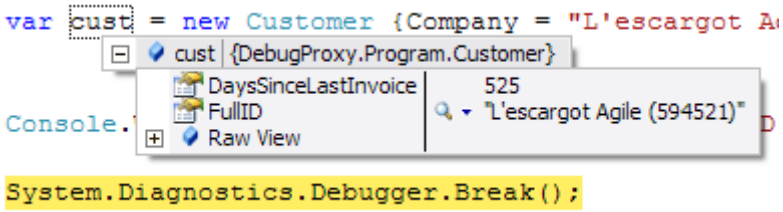
Vous avez parfaitement raison, nous n'allons pas créer d'instance de cette classe, nous n'allons pas même modifier le code de l'application en cours de debug (ce qui serait une grave erreur... modifier ce qu'on test fait perdre tout intérêt au test). Non, nous allons simplement indiquer au framework .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 !

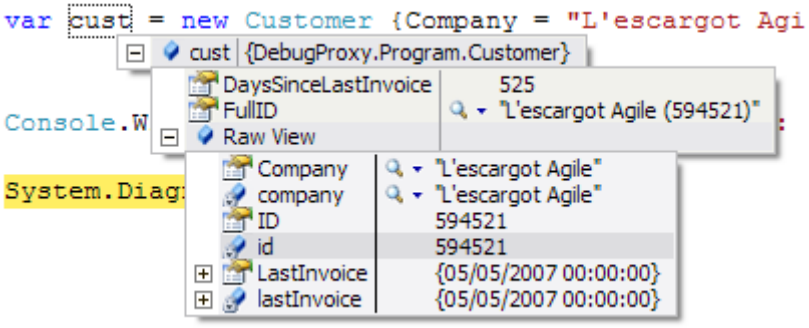
Le proxy en marche

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 !

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.

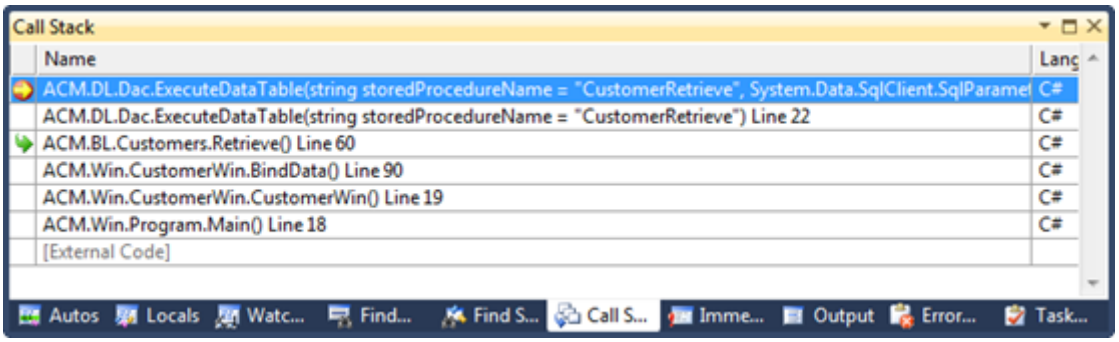
Vous vous êtes aussi rendu compte certainement que pour faciliter la lecture la ligne "courante" est surlignée en jaune et que ses appelants le sont en vert.

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 :

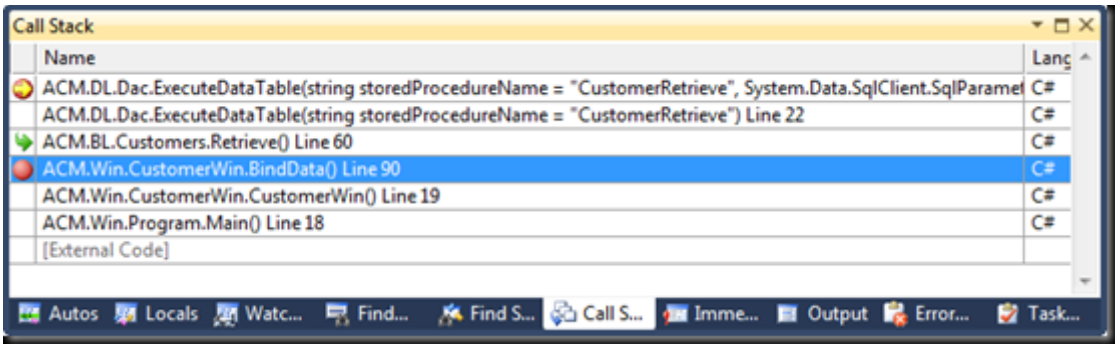


On peut voir ici une cascade d'appels depuis le Main() du programme jusqu'à la méthode ExecuteDataTable, en passant par toutes les couches de l'architecture du logiciel traversées par l'appel.

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...

Astuce de debug avec les exceptions imbriquées

Les exceptions peuvent en contenir d'autres, comment gérer cette hiérarchie simplement ?

Une exception peut en cacher une autre !

A l'instar des trains, une exception peut en cacher une autre. Ce qui de plus en informatique est récursif puisque cette autre exception peut elle-même en cacher une autre… etc.

Une exception est facile à traiter, mais dès lors qu'on cherche à savoir si elle contient une autre exception (et ainsi de suite) les choses se compliquent et souvent cette hiérarchie est ignorée par le développeur. Pourtant – au moins pour une trace de débogue par exemple – connaitre facilement toutes les exceptions qui s'enchainent est crucial, car souvent le message intéressant se trouve caché dans cette chaîne.

Aplatir la hiérarchie

Joli rêve d'anarchiste ! Ici il ne s'agira que de rendre facilement manipulable la structure chainée des exceptions en la linéarisation sous forme d'une liste.

L'idée est plutôt simple et on peut régler le problème par une méthode d'extension comme celle-ci :

```
1.  using System;
2.  using System.Collections.Generic;
3.
4.  namespace Utils
5.  {
6.      public static class DebugExtensions
7.      {
8.          public static IEnumerable<Exception> FlattenHierarchy(this Exception
    ex)
9.          {
10.             if (ex == null) throw new ArgumentNullException("ex");
11.
12.             var innerException = ex;
13.             do
14.             {
15.                 yield return innerException;
16.                 innerException = innerException.InnerException;
17.             }
18.             while (innerException != null);
19.         }
20.     }
21. }
```

Plus de chainage à gérer dans le code mais une simple liste d'exceptions qu'on peut interroger et traiter facilement avec Linq.

Exemple fictif où l'exception est "aplatie", chaque exception (1er niveau ou interne) étant envoyée à un gestionnaire de message d'erreur global :

```
// ...
catch (Exception ex)
      {
            ex.FlattenHierarchy().ToList()
             .ForEach(x => AddErrorToGlobalList("EmailError", x.Message));
      }
```

On peut aussi filtrer aisément les exceptions ce qui rend le traitement plus exhaustif que de se limiter à tester le type ou le message de premier niveau :

```
if (exception.FlattenHierarchy().Where(e => e.Message == "Disk crash!").Any()) ...
```

Conclusion

Rien de bien sorcier, mais encore faut-il penser à ajouter cette méthode d'extension dans toutes les applications car ne gérer que le premier niveau n'est pas tout à fait suffisant lorsqu'on souhaite cibler une condition particulière sans mettre des try/catch trop génériques (ce qui n'est pas un bon réflexe, à trop masquer toutes les erreurs ont loupe même en phase de test des bogues qui surgiront plus tard et plus gravement en production !).

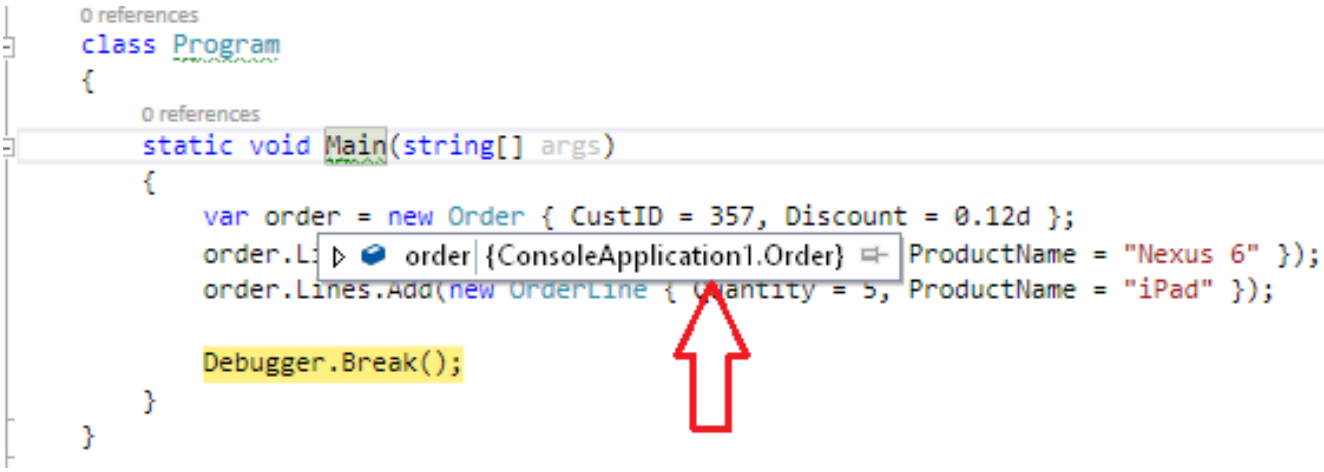
Bon débogue…

Simplifiez-vous le debug

Un tout petit ajout au code d'une classe, sans le perturber, permet d'obtenir un confort incroyable lorsqu'on débogue…

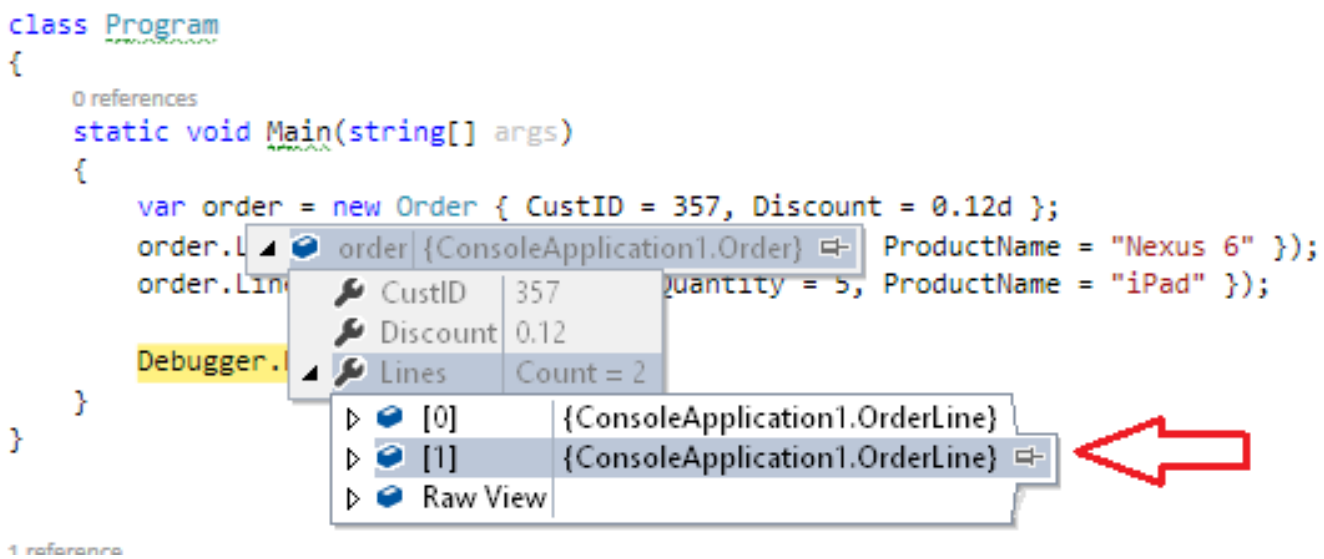
Prenons une classe simple

Essayons le debug sur la variable "order" :



C'est pas génial…

Essayons sur la propriété "Lines" de cet Order :



Ce n'est pas beaucoup mieux…

Il faut entrer dans une arborescence de relations pour espérer voir l'information qui intéresse en débogue.

Bricoler ToString() ?

C'est assez moche comme solution… D'autant que ToString() appartient à la classe et qu'il peut avoir utilisé pour simplifier un affichage dans une ListView ou autre. C'est mal de faire ainsi, mais ça existe. Aller tripatouiller le ToString() juste pour les besoins du débogue c'est peut-être casser le code utilisé ailleurs…

Un Simple Attribut

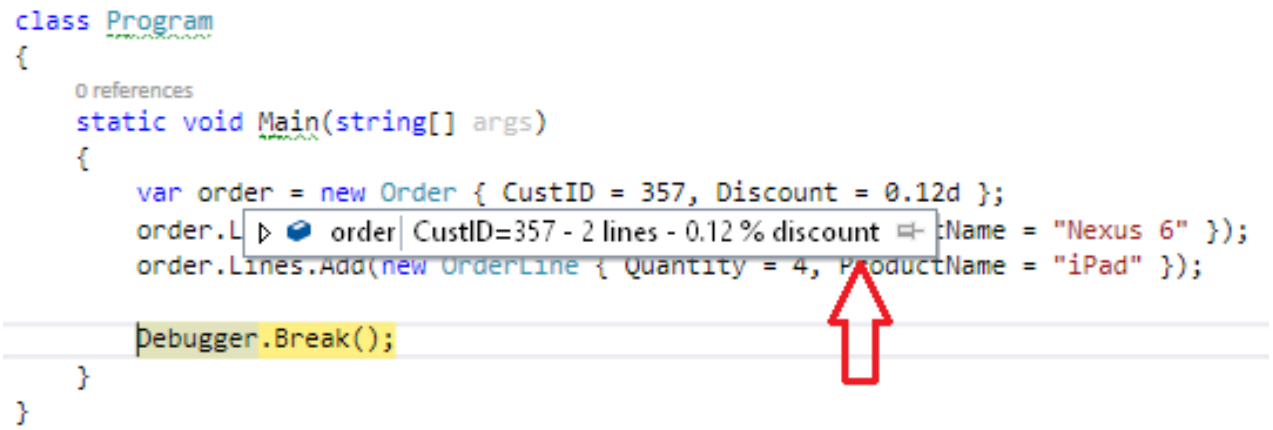
Avec l'attribut *DebuggerDisplay* qu'on place sur la classe inutile de bricoler le code de celle-ci, on l'instrumentalise de l'extérieur. Totalement dédié au debug on fait ce qu'on veut, on affiche certaines données à un moment donné parce qu'elles ont un sens pour la séance de débogue en cours, on en affiche d'autres si besoin est dix minutes plus tard. La classe décorée n'a pas changé et c'est essentiel.

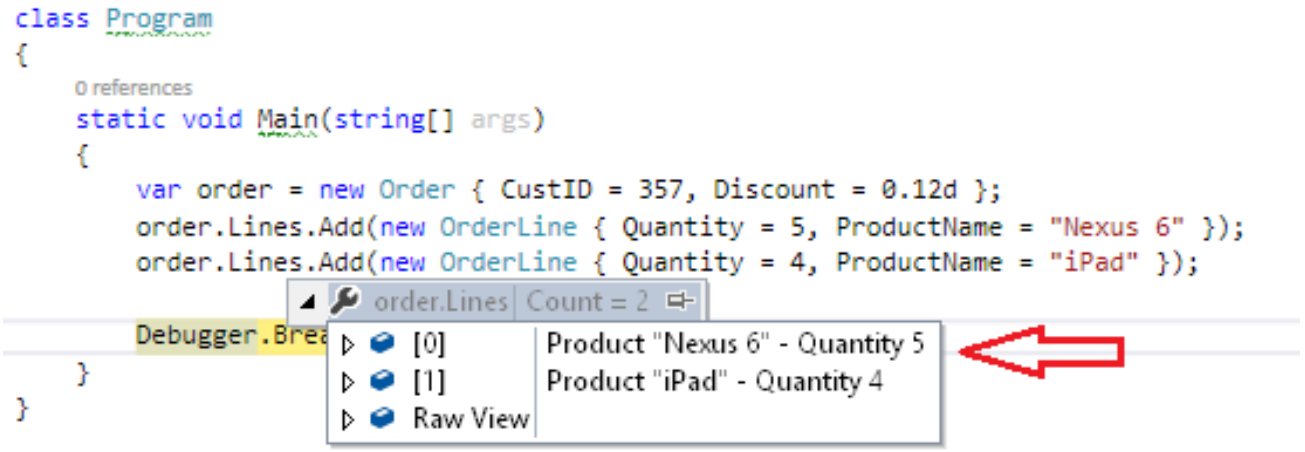
Voici les classes utilisées par l'exemple maintenant décorées :

```
[DebuggerDisplay("CustID={CustID} - {Lines.Count} lines - {Discount} % discount")]
1 reference
public class Order
{
    1 reference
    public int CustID { get; set; }
    2 references
    public IList<OrderLine> Lines { get; } = new List<OrderLine>();
    1 reference
    public double Discount { get; set; }
}

[DebuggerDisplay("Product {ProductName} - Quantity {Quantity}")]
4 references
public class OrderLine
{
    2 references
    public int Quantity { get; set; }
    2 references
    public string ProductName { get; set; }
}
```

Et voici les mêmes affichages de débogue présentés plus haut mais avec l'attribut :





Tout de suite c'est plus clair et sans toucher le code de la classe. L'attribut peut rester, en version release il est ignoré.

Conclusion

Il me semble que j'avais déjà du en parler il ya longtemps mais comme je remarque que cet attribut n'est toujours pas beaucoup utilisé, autant en remettre une petite couche !

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 !

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.

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 ?

Du mauvais usage de DateTime.Now dans les applications

Qui pourrait s'interroger sur le bienfondé de l'utilisation de `DateTime.Now` pour obtenir la date et l'heure courantes ? Il y a pourtant matière à réfléchir !

Une solution simple à un besoin simple

Utiliser `DateTime.Now` pour connaitre l'heure ou la date courantes dans une application est une merveilleuse solution simple à un besoin qui l'est tout autant. La solution est simple car elle utilise le framework .NET correctement, c'est à dire comme des API rationalisant et objectivant des fonctions système (heure, fichiers, mémoire…).

Le problème lui-même est souvent d'une simplicité tout aussi déconcertante :

- Effectuer une opération à une heure ou un jour particulier (type prélèvement bancaire par exemple)
- Gérer un système de réservation et d'expiration d'entités
- Lancer une lettre d'information par e-mail ou un traitement particulier uniquement la nuit
- Gérer des dates anniversaires (début, fin de contrats, date de relance…)
- Etc, etc,

Dans de tel cas on écrira le plus souvent un code qui peut se généraliser à :

```
if (DateTime.Now>LaDateTest) { … action }
```

Un besoin moins simple qu'il n'y parait…

En réalité tout cela est idyllique, donc irréel…

Une véritable application sera soumise à deux cas d'utilisation principaux que DateTime.Now ne permet pas d'adresser :

- Si on souhaite globalement (ou non) appliquer un "offset" à la date ou l'heure courante (ie: "avancer" ou "retarder" sur l'heure légale locale)
- Quand on doit tester l'application !

Dans le premier cas il faut revoir toutes les séquences de code utilisant directement (c'est assez facile) ou indirectement (beaucoup plus dur) la propriété DateTime.Now. La charge de travail sera lourde et le résultat incertain, entendez plein de petits bogues difficiles à déceler. D'autant plus qu'il n'y a aucun moyen de tester tout cela proprement justement…

Dans le second cas rien n'est possible tout simplement.

Imaginons juste un instant une fonctionnalité qui dans l'application considérée effectue une clôture comptable à 18h00 précises avec déclenchement de plusieurs actions (somme des mouvements de la journée de chaque compte, à nouveau pour la journée suivante, alerte sur comptes débiteurs, etc).

Vous êtes en train de coder cette application. Et vous devez vous assurer que tout marche bien. C'est un peu le minimum syndical du développeur…

Allez-vous rester assis jusqu'à 18h00 pour voir si tout se met en route comme prévu alors que vous venez à peine d'arriver au bureau et que le café fume encore dans son gobelet ? Et si cela ne marche pas et qu'il faut effectuer une correction, allez-vous rester planté là 24h jusqu'a temps qu'il soit de nouveau 18h00 pile ?

Je n'y crois pas… Donc la fonction sera testée en production n'est-ce pas ?

Ne rougissez pas, et ne niez pas non plus, je sais que ça se passe comme ça ! 

Du bon usage de la date et de l'heure dans une application bien conçue et bien testée

Constater une faiblesse c'est déjà progresser, mais encore faut-il avancer un pas de plus et proposer une solution…

Et cette dernière est toute simple et tient en une règle : *n'utilisez jamais DateTime.Now* !

C'est radical, absolu, donc clair et facile à respecter !

C'est bien joli d'inventer des règles de ce genre, mais en réalité on fait comment ?

… On remplace la fichue propriété de la classe DateTime par une autre propriété d'une autre classe que nous allons créer pour cet usage.

Là aussi c'est simple, radical et donc facile à mettre en œuvre.

Imaginons une classe Horloge, statique (qui peut être vue comme un service), qui expose une propriétéMaintenant, de type DateTime. Propriété statique aussi.

C'est cette propriété que nous allons utiliser systématiquement dans notre application à la place deDateTime.Now…

En quoi cette propriété et cette nouvelle classe peuvent elles résoudre ce que la classe du framework ne peut pas faire ?

C'est très simple là encore. La propriété Maintenant retournera soit DateTime.Now, par défaut, sans avoir rien à faire de spécial, soit le résultat d'une fonction de type DateTime que l'application pourra initialiser à tout moment.

Pour tester si tel morceau de l'application se déclenche à 18h00 pile il sera facile de retourner directement cette heure là. Quelle que soit la véritable heure, pour l'application il sera 18h00 pile… On peut aussi complexifier un peu et se débrouiller pour qu'il soit 18h00 moins une minute pour voir le passage entre "l'avant" et "l'après" 18h00 pile s'effectuer (souvent un bogue peut se cacher dans des transitions de ce type, là où un test sur une valeur précise ne détecte pas le problème).

Bref, la nouvelle classe et sa nouvelle propriété vont permettre d'écrire un code testable à toute heure en fixant arbitrairement l'heure et la date de façon globale (donc cohérente pour toute l'application, ce qui est conforme à une situation réelle en production).

Bien entendu cette classe pourra être complétée à volonté selon le projet, l'utilisation par celui-ci de l'heure UTC ou non, etc. C'est le principe qui nous intéresse ici. Le détail de l'implémentation vous appartenant.

La classe Horloge

Dans l'esprit qui nous habite ici nous ne voulons que modifier l'accès à DateTime.Now, les autres fonctions et propriétés de la classe DateTime restant ce qu'elles sont. Encore une fois si d'autres besoins du même type doivent être couverts par une solution identique, le développeur ajoutera ce qu'il faut à la classe Horloge.

```
public static class Horloge
{
    private static Func<DateTime> fonction;
    static Horloge()
    {
        fonction = () => DateTime.Now;
    }
    public static DateTime Maintenant
    {
        get
        {
            return fonction();
        }
    }
    public static Func<DateTime> FonctionMaintenant
    {
        set
```

```
        {
            fonction = value ?? (() => DateTime.Now);
        }
    }

    public static void Reset()
    {
        FonctionMaintenant = null;
    }
}
```

Il suffit désormais de remplacer systématiquement dans l'application tous les appels à `DateTime.Now` par `Horloge.Maintenant` (chacun choisira d'angliciser ou non ces noms).

Jusque là nous aurons une stricte équivalence fonctionnelle sans rien de moins et sans rien de plus que l'appel à la classe du framework.

Mais si nous décidons de tester notre application pour voir ce qu'il se passe quand il est 18h00, alors il sera facile en début d'exécution ou n'importe où dans le code où cela prend un sens, d'ajouter quelque chose comme :

```
// nota : attention à la closure !
var simulation =
 new DateTime(DateTime.Now.Year, DateTime.Now.Month, DateTime.Now.Day, 18,
0, 0);
Horloge.FonctionMaintenant = () => simulation;
```

Comme je l'indique en commentaire il faut faire attention aux closures dans ce type de code.

Naturellement la fonction retournée peut être beaucoup plus complexe que de retourner une variable. C'est une expression Lambda qui peut contenir tout ce qu'on désire (décalage temporel glissant, heure ou date aléatoires disposant de leur propre timer, incrémentation à chaque appel pour simuler le temps qui passe dans une série de données et la fabrication de statistiques ou de graphiques, etc…).

Conclusion

Penser à ce genre de choses en début d'écriture d'une application apporte un confort qu'on ne regrette jamais. C'est un peu comme utiliser des Interfaces pour découpler des parties de code ou créer une classe mère pour ses ViewModels. Au début cela parait parfois un peut lourd pour pas grand chose, mais le jour où apparait un traitement qui diffère de celui de base ou qu'il faut appliquer systématiquement partout on se félicite d'avoir fait un tel choix !

Faut-il donc systématiquement remplacer DateTime.Now par quelque chose du type de Horloge.Maintenantdémontré ici ? Je pense que oui. Je n'ai trouvé aucun

argument contre qui tienne longtemps, en revanche j'ai trouvé quelques arguments favorables qui incitent à suivre cette guideline.

J'ai trouvé intéressant de vous soumettre ce petit problème histoire que vous aussi vous dormiez moins bien ce soir en pensant à tout le code que vous avez écrit sans prendre la précaution d'isoler Now

Surcharger l'incrémentation dans une classe

Après avoir parlé des dangers de l'incrémentation dans un contexte multitâche parlons de cet opérateur positivement en montrant comment le surcharger dans une classe et l'intérêt qu'on peut y trouver…

Une possibilité trop peu utilisée : la surcharge des opérateurs

Surcharger ou non les opérateurs est une question *clivante*…

Il y a ceux qui sont totalement pour, et ceux qui sont contre. Rarement il n'existe de position intermédiaire.

Les arguments contre la surcharge sont connus : un opérateur qui n'a pas la même signification partout est un piège à bogues et les attire comme un aimant attire la limaille de fer… Ce n'est pas faux.

Les arguments pour la surcharge sont tout aussi simples : c'est une possibilité du langage aucune raison de ne pas s'en servir… Ce n'est pas faux non plus.

La position de Dot.Blog ? Si certains pensent que la vérité est ailleurs, elle est pour moi bien plus souvent au milieu. Je dirais que personnellement je déconseille d'utiliser la surcharge sauf quand celle-ci apporte de la lisibilité et de la clarté au code et que même si on ne sait pas que cette surcharge a été faite l'effet obtenu sera logique et compréhensible…

Il ne s'agit donc pas de "trop" ou de "trop peu", mais uniquement de logique, de bon sens et de mesure.

Exemple : Incrémenter les faces d'un polyèdre

Regardons rapidement le GIF animé ci-dessous (à voir sur Dot.Blog !).

Nous disposons d'une classe Polyhedron (polyèdre en anglais). Un Polyèdre est défini comme un volume constitué de plusieurs faces. Les Faces, et leur nombre précis, sont ainsi l'information essentielle et centrale des instances de la classe peu importe ensuite ce qu'elle fait (dessiner le polyèdre par exemple).

Dans une telle classe il peut être intéressant non pas de surcharger mais simplement de définir un opérateur d'incrémentation (et de décrémentation, ce qui n'est pas fait dans l'exemple). Ainsi au lieu d'écrire `MonPolyèdre.Faces = MonPolyèdre.Faces+1` on écrira de façon plus concise `MonPolyèdre++;`

Le sens de l'incrémentation est parfaitement respecté, cela reste logique et compréhensible. Si on ne sait pas que l'opérateur est défini on comprendra assez facilement qu'un "`++`" sur un polyèdre augmente le nombre de faces.

On peut d'ailleurs utiliser cette technique pour rendre l'incrémentation thread-safe en utilisant `Interlocked.Increment()` dans la définition ou redéfinition de l'opérateur. Dans ce cas on introduit une différence fonctionnelle importante et celle-ci doit être dument documentée…

Conclusion

Comme toutes les bonnes choses il ne faut pas abuser de cette technique car elle introduit tout de même un sens qui n'est pas celui de l'opérateur à l'origine. Incrémenter un nombre de faces par Faces++ est parfaitement logique et ne réclame aucune information en dehors de la connaissance du langage, mais MonPolyèdre++ est déjà plus étrange même s'il y a une logique claire sous-jacente. Incrémenter un polyèdre n'a *stricto sensu* pas de signification.

Si on fait bien attention à cette distorsion de sens, qu'elle est documentée, et que l'apport est réellement intéressant dans le contexte alors se priver de cette possibilité de redéfinir les opérateurs n'est pas acceptable, le langage le permet, autant s'en servir.

On retrouve ici la position médiane exprimée plus haut. Ni pour, ni contre, au milieu je suis… comme aurait pu dire Me Yoda…

Surchargez bien, mais attention à ne pas écraser le code sous ce poids !

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 ?

Static ou Singleton ?

Il peut exister une sorte de confusion entre Static et Singleton, pourtant il s'agit de choses bien différentes…

Langage ou architecture ?

La principale différence se cache dans la nature profonde des deux concepts.

Static est une construction du langage.

Un Singleton est un design pattern, c'est un concept d'architecture.

Il existe donc déjà à la base une nuance forte entre un simple élément de langage et un pattern qui est une construction intellectuelle visant à proposer une meilleure architecture du code.

On peut réaliser un singleton en utilisant une classe statique (simple possibilité) mais une classe statique n'est pas suffisante pour créer un singleton.

Instances et états

Une classe statique n'est pas instanciée par essence. Dans ce cas elle ne peut pas prétendre sérialiser ses états par exemple.

Mais est-ce une bonne pratique de gérer des états dans une classe statique ? non.

Une classe statique ne doit idéalement faire que proposer des méthodes ou des constantes. Par exemple on créera une classe statique pour regrouper des méthodes d'extension.

En revanche un Singleton peut avoir besoin de maintenir des états durant son fonctionnement.

C'est pour cela que l'implémentation d'un Singleton ne passe pas forcément par une classe statique…

Héritage

On ne peut pas hériter d'une classe statique. On peut avoir besoin d'hériter d'un Singleton.

Encore une nuance qui force à réfléchir !

Thread Safe ?

Par essence ni les classes statiques ni les singletons ne sont thread-safe. Il faut ajouter du code pour cela, il n'y a donc pas de différence sur ce point.

Quand utiliser l'un ou l'autre ?

Typiquement une classe statique s'utilise lorsqu'il n'y a besoin d'aucun héritage, immédiat ou dans le futur, et dans le cas où il n'y aucun état à gérer ni sérialisation de ces derniers (par exemple pour sauvegarde l'état en vue d'une réhydratation ultérieure).

Un Singleton s'utilise lorsque qu'on désire s'assurer de l'unicité d'une instance (service par exemple) tout en pouvant gérer ses états et leur éventuelles sérialisation / réhydratation (directement ou via le design pattern Memo par exemple), ceci en conservant la possibilité d'hériter du Singleton pour le spécialiser dans l'immédiat ou dans le futur (simple possibilité, un Singleton est souvent "sealed").

Une classe statique n'a pas d'instance alors que le Singleton est un moyen de restreinte des instances (généralement une d'où son nom).

Enfin on notera qu'un singleton, parce qu'il existe une instance, peut être passé en paramètre à une méthode alors qu'une classe statique ne peut pas l'être. Et dans les constructions de type conteneur IoC et Injection de dépendances ce "petit" détail peut s'avérer tout simplement crucial.

Conclusion

Classe statique et Singleton n'ont finalement rien à voir. Il n'y a pas d'équivalence ni relation particulière entre les deux.

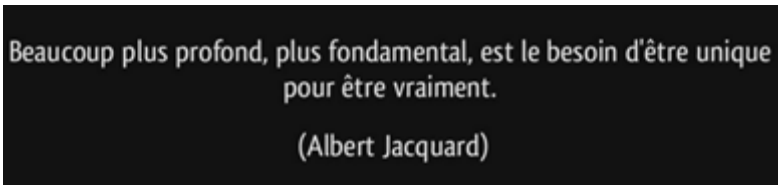
On peut utilise une classe statique pour construire un singleton mais cela n'est pas la meilleure solution. Le lien éventuel s'arrête là.

J'espère que ce petit point vous aidera à mieux choisir quand utiliser l'un ou l'autre de ces concepts pour produire un meilleur code !

Singleton qui es-tu ?

C'est un thème que je n'ai jamais abordé seul en près de 900 billets depuis 2008 et pourtant j'en ai parlé souvent, même très récemment. Ne serait-il pas temps de faire le point sur ce design pattern ?…

Singulier vs Pluriel



Le besoin de se définir comme unique, le "je", ne concerne pas seulement les humains mais aussi ces petits acteurs autonomes qu'on appelle des objets dans notre métier…

Techniquement les objets ont parfois le besoin d'être uniques pour être utiles. Le pluriel, la multitude, sont souvent synonymes d'interchangeabilité. Du point de vue de l'humain qui possède une conscience se voir assimiler à une foule sans visage est la pire des négations, mais pas pour les objets ! Leur multitude est leur avantage comme les fourmis dans une fourmilière. Mais cet avantage a ses limites. Les ruches ont leur reine, les navires leur capitaine, ils ne tirent pas tant leur légitimité du pouvoir qu'ils incarnent que de leur unicité et de la constance dans le temps de leur fonction, piliers singuliers sur lesquels le pluriel peut se reposer pour s'organiser. Les applications ont elles aussi besoin de certains point d'ancrage fiables et constant dans le temps. La notion de "service", les Factories et même les conteneurs IoC ont besoin de tels points fixes. Cette position particulière a été étudiée et porte un nom : le Singleton.

Le Singleton méritait bien une petite intro philosophico-technique !

Le Design Pattern Singleton

La référence absolue en matière de Design Patterns c'est le livre du Gang Of Four. Ca commence à dater mais c'est toujours d'une actualité indiscutable. Il est donc toujours temps de vous le procurer et de le lire et le relire ! En attendant voici la définition théorique du Singleton car rien ne peut se discuter sans l'exposé et la compréhension de celle-ci.

Classification

Le Singleton est classé dans la catégorie des Patterns dit “Creational” groupe qui contient l’Abstract Factory, le Builder, la Factory Method et le Prototype. Leur point commun est d’aider un système à être indépendant vis-à-vis de la création de ses objets, leur composition (au sens des liens entre instances) et leur représentation. Tous ces patterns offrent des abstractions du processus d’instanciation.

Intention

S’assurer qu’une classe ne possède qu’une seule instance et assurer un point d’accès unique et bien défini à celle-ci.

Motivation

j’ai déjà évoqué ici ou dans d’autres articles l’intérêt pour certains objets d’être uniques et “centraux” (accessibles de partout).

Mais on peut ajouter d’autres circonstances comme par exemple tout ce qui est relié à des ressources physiques. Un système d’impression par exemple. Il ne suffira pas de multiplier les instances en mémoire pour posséder plusieurs imprimantes au lieu d’une ! Et même si on permet de lancer plusieurs impressions en même temps, elles seront stockées dans un Spool qui lui sera unique. Au sein d’une application un système d’affichage de messages à destination de l’utilisateur aura tout intérêt à être lui aussi unique. On pourrait y passer des jour à lister toutes les situations qui réclament la présence d’une instance unique et accessible facilement.

Applicabilité

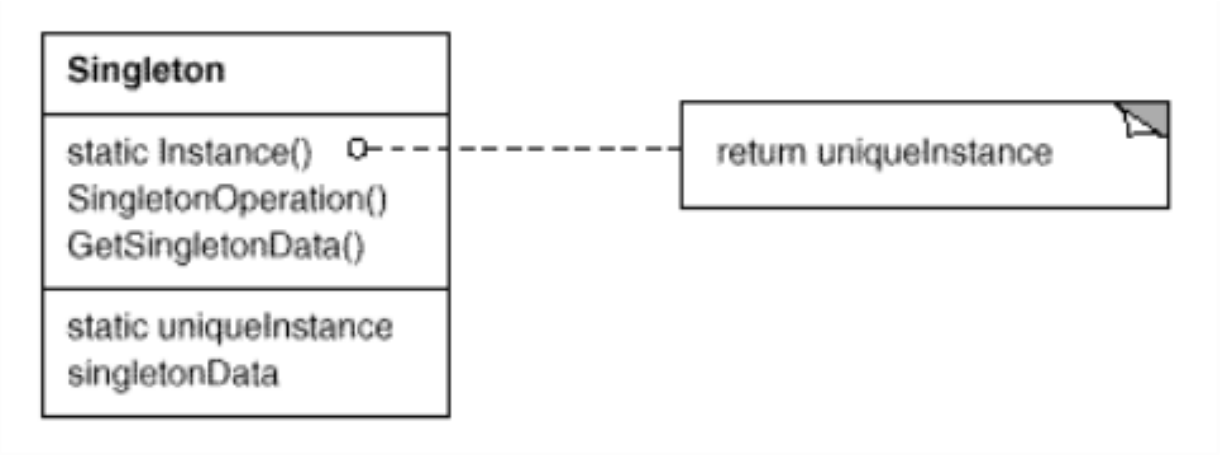
On utilise un Singleton quand :

Il doit y avoir une et une seule instance d’un objet précis et qu’elle doit être accessible par les clients depuis un point d’accès bien identifié.

L’instance unique doit pouvoir être étendue par sous-classement et que les clients doivent pouvoir utiliser la version étendue sans modifier leur code.

Si le premier point est généralement bien compris, le second est moins “médiatisé”. Il explique à lui seul pourquoi un Singleton ne s’implémente pas avec une classe statique non héritable par nature.

Structure



Pour qui sait lire UML (ici très simple) on retrouve les ingrédients des principales implémentations. Ce qui est normal puisque toutes se sont inspirées à un moment où un autre de ce schéma fondateur !

Participants

Le Singleton lui-même. Une seule classe participe à ce design pattern.

Collaborations

Les clients accèdent au Singleton uniquement au travers du point d'accès "Instance" qu'il offre. Une fois l'instance obtenue toutes les opérations offertes par le Singleton sont accessibles.

Conséquences

L'utilisation du Singleton implique les conséquences suivantes :

Contrôle des accès à l'instance unique. Puisque le Singleton encapsule sa seule instance et les moyens d'y accéder il peut contrôler, filtrer, logger les accès des clients.

Réduction de l'espace de nom. En C# le problème ne se pose pas mais dans les langages qui autorisent la déclaration de variables globales l'utilisation du Singleton permet de clarifier les choses et évite de polluer cet espace commun.

Raffinement des opérations et représentations. La classe du Singleton peut être sous-classée et il est facile de configurer l'application pour qu'elle utilise de façon transparente la nouvelle classe.

Autorise un nombre variable d'instances. Le Singleton, quoi que ce nom signifie, est un pattern qui est assez souple pour autoriser un changement d'avis quant au nombre des instances qu'il gère. Il peut donc aussi servir à contrôler un nombre fixe ou variables d'instances.

L'approche est plus souple que l'utilisation de membres ou classes statiques. Ne serait-ce que parce que ces dernières n'autorisent pas l'héritage.

Les implémentations

Il existe plusieurs façons d'implémenter le pattern Singleton même si au bout du compte seule une ou deux sont à retenir.

La version historique

Commençons par la version "historique" celle qui suit les recommandations du livre "Design Patterns : Elements of Reusable Object-Oriented Software" du Gang of Four paru en 1995. En C# cela donne :

```
1. using System;
2.
3. public class Singleton
4. {
5.    private static Singleton instance;
6.
7.    private Singleton() {}
8.
9.    public static Singleton Instance
10.         {
11.            get
12.            {
13.               if (instance == null)
14.               {
15.                  instance = new Singleton();
16.               }
17.               return instance;
18.            }
19.         }
20.      }
```

Cette version possède deux avantages :

Parce que l'objet est instancié dans la propriété "Instance" la classe peut à loisir faire d'autres traitements, notamment instancier une sous-classe.

L'instance n'est pas créée tant qu'il n'y a pas eu un appel à la propriété, procédé appelé "lazy instanciation". L'effet est bénéfique puisque la création l'instance n'a lieu que lorsqu'elle est réellement nécessaire.

Toutefois il existe un point faible à cette implémentation, elle n'est pas thread-safe. Si plusieurs threads entrent dans la propriété Instance en même temps plusieurs instances de l'objet peuvent éventuellement être créées.

Initialisation statique

L'une des raisons invoquées à l'origine pour déconseiller l'utilisation d'une initialisation statique est que C++ est très ambigu sur l'ordre des initialisations statiques. Mais C# a des spécifications bien plus claires et l'ordre est garanti.

Il devient donc possible en C# d'utiliser une initialisation statique de l'instance ce qui donne :

```
1. public sealed class Singleton
2. {
3.     private static readonly Singleton instance = new Singleton();
4.
5.     private Singleton(){}
6.
7.     public static Singleton Instance => instance;
8. }
```

L'avantage de l'initialisation statique est qu'elle conserve la principe du lazy instanciation puisque l'instance ne sera créée que la première fois qu'un membre de la classe sera référencé. Le CLR (Common Language Runtime) gère cet aspect.

Le fait que la classe soit marquée "sealed" est un choix qui se discute forcément et n'est pas obligatoire. L'avantage est de s'assurer qu'aucune sous-classe ne pourra être créée et donc qu'aucune autre instance ne pourra être obtenue par ce biais. Si la classe contrôle une device physique c'est une bonne idée qui offrira une garantie de plus.

Mais dans d'autres circonstances le fait de celer la classe ne fait que rendre impossible le sous-classement qui plus haut était présenté comme l'un des avantages du pattern Singleton… C'est donc à chacun de trancher en fonction du contexte.

On remarquera aussi que la variable privée est marquée "readonly" cela permet de s'assurer qu'en dehors du constructeur éventuel (absent ici puisqu'inutile) le champ ne pourra pas être modifié. C'est une bonne protection. De même le constructeur est privé de telle façon qu'aucun autre code ne puisse créer une instance du Singleton.

Le point d'entrée est statique (la propriété Instance) et permet un accès logique et centralisé. Ce qui fait partie de la définition d'un Singleton.

Ce code diffère principalement du précédent par le fait qu'il se repose sur le CLR pour instancier l'objet.

Le seul problème de cette solution est que le singleton est instancié via son constructeur par défaut. Or un Singleton se caractérise par le contrôle qu'il exerce sur la création de l'instance. Il peut par exemple appeler un constructeur autre que celui par défaut, lui passer d'éventuels paramètres, etc. L'initialisation statique n'apparait donc pas optimale de ce point de vue même si elle permet un code très court et fiable même en environnement multi-threadé.

Singleton thread-safe

L'initialisation statique est donc une bonne solution qui fonctionnera dans de nombreux cas. Toutefois dès lors qu'il faudra utiliser un constructeur autre que celui par défaut ou réaliser d'autres tâches avant l'instanciation dans un environnement multithreadé on devra se tourner vers d'autres implémentations.

Bien entendu la solution passe par l'utilisation de mécanismes de verrouillage propre au langage et à la plateforme considérée. Avec C# sous .NET ou UWP nous savons que nous pouvons nous baser par exemple sur l'opération "lock" qui utilise en interne Monitor (et donc simplifie l'écriture du code).

Or locker tous les accès à la propriétés possède un coût. On aimerait limiter ce dernier.

Ce qui a amené au développement d'un pattern appelé "double-check locking". Il s'agit ici d'acquérir le verrou que si cela est nécessaire, donc une seule fois lorsque le Singleton n'est pas encore instancié.

Le code devient le suivant :

```
1. public sealed class Singleton
2. {
3.    private static volatile Singleton instance;
4.    private static object syncRoot = new Object();
5.
6.    private Singleton() {}
7.
8.    public static Singleton Instance
9.    {
10.           get
11.           {
12.              if (instance == null)
13.              {
14.                 lock (syncRoot)
15.                 {
16.                    if (instance == null)
17.                       instance = new Singleton();
18.                 }
19.              }
20.
```

```
21.                 return instance;
22.             }
23.         }
24.     }
```

Le “double-check” se voit clairement puisque deux fois, hors et dans le lock, la variable instance est testée sur le nul. Double contrôle donc.

Il existe une polémique autour du double-check locking qui a surtout remué la communauté Java. Le procédé ne serait totalement fiable en raison du modèle mémoire de ce langage, ce qui peut être démontré. Et du fait que le résultat de ce pattern ne soit pas constant selon les compilateurs c’est le pattern lui-même qui est tombé en désuétude puisqu’un design pattern se veut le plus générique possible.

Or C# et son CLR corrigent bien des problèmes de C++ et de Java, et notamment en ce qui concerne le problème lié au double-check locking nous avons l’assurance qu’il ne peut se produire. L’utilisation de ce pattern dans ce contexte est donc parfaitement légitime.

Ainsi le code ci-dessus fonctionne parfaitement C#. Mais on remarquera une autre précaution : l’utilisation du mot clé “volatile” dans la déclaration de la variable.

Ce mot clé rarement utilisé et même connu par les développeurs permet de s’assurer que la variable ne sera lue qu’après son assignation (en tout cas ici). Ce qui nous permet d’être certain du bon fonctionnement de l’ensemble.

Volatile est méconnu, car comme tout ce qui tourne autour du multitâche il y a comme un halo de mystère qui rend les choses troubles, floues et inquiétantes. Pourtant volatile a un rôle à jouer dans ce type d’environnement multitâche. Pour être plus précis sur son fonctionnement disons qu’il indique qu’un champ peut être modifié par plusieurs threads en même temps. Les optimisations du compilateur sont désactivées pour la variable ainsi marquée (optimisation qui supposent une utilisation mono thread). De fait un thread lecteur sera assuré de toujours lire la valeur la plus à jour. D’un certain point de vue “volatile” permet de se passer des locks.

Le code ci-dessus pour fonctionner correctement traine ainsi avec lui la sulfureuse réputation du double-check locking en Java et les mystères afférents à l’utilisation de “volatile”. C’est donc un code qui, malgré les assurances que C# fonctionne de façon plus fiable que C++ ou Java, créée une sorte de réticence chez certains.

De fait ces personnes méfiantes préfèreront une version plus explicite du verrouillage, quitte à ce qu’elle soit moins efficace, et qui consiste tout simplement à locker totalement l’accès à la variable au niveau de la propriété, autant en lecture qu’en écriture.

Dans ce cas plutôt que de faire compliqué sans vrai raison, et à condition qu'on puisse assumer le fait que l'objet ne pourra être créé que par son constructeur par défaut (et sans traitement avant cette création) la version avec initialisation statique sera certainement la meilleure solution, fiable et ultra simple comme le prouve le code plus haut dans cet article. A noter que l'on perd aussi la lazy instanciation ce qui au final ne sera gênant que très rarement (car si le code n'est pas appelé, après tout il doit être supprimé, et s'il est appelé, plus tôt ou plus tard ne fait en réalité que peu de différence pour la variable instanciée sauf si elle implique des ressources très gourmandes).

Conclusion

Par le rôle qu'il joue dans de nombreuses applications le Singleton est un pattern essentiel à connaitre, comprendre et maitriser.

Au départ très simple, son implémentation peut se compliquer conceptuellement (le code reste court) dès lors qu'on veut gérer tous les aspects en même temps (lazy instanciation, thread-safe…).

Comme l'initialisation statique permet le code le plus court et qu'elle est thread-safe, c'est mon implémentation préférée.

Codiez-vous correctement le Singleton en parfaite connaissance de cause ? En tout cas maintenant vous savez tout pour le faire convenablement !

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 !

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 subtils 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.

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 après toutes ces années 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 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 une collection de roues et une autre de portières par exemple. 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 avoir zappé cette feature de Delphi dans C# alors que bien d'autres ont été reprises (avec raison ou moins comme le Goto ou plus avec les propriétés) ?

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'abri 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 :-)

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 !)

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>
```

```
53:             <xs:attribute name="CHANGE LOG" use="required">
54:                 <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…

#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 ?

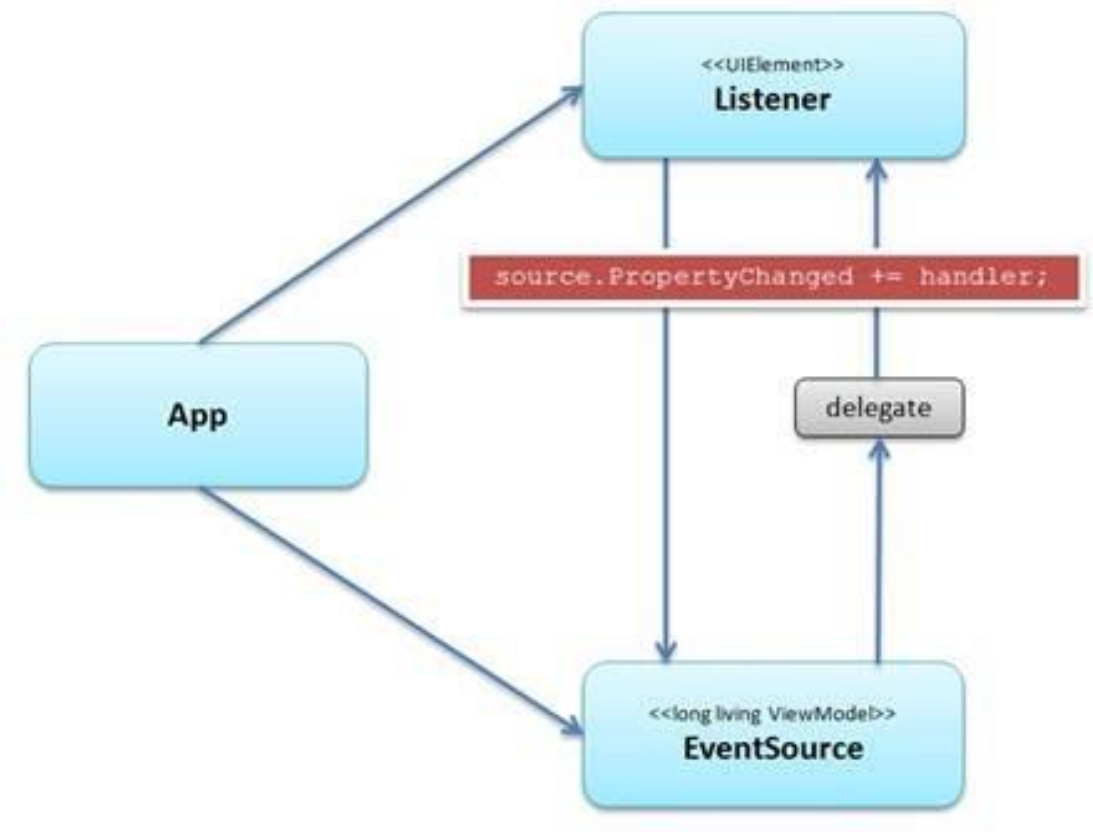
Et bien oui !

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).



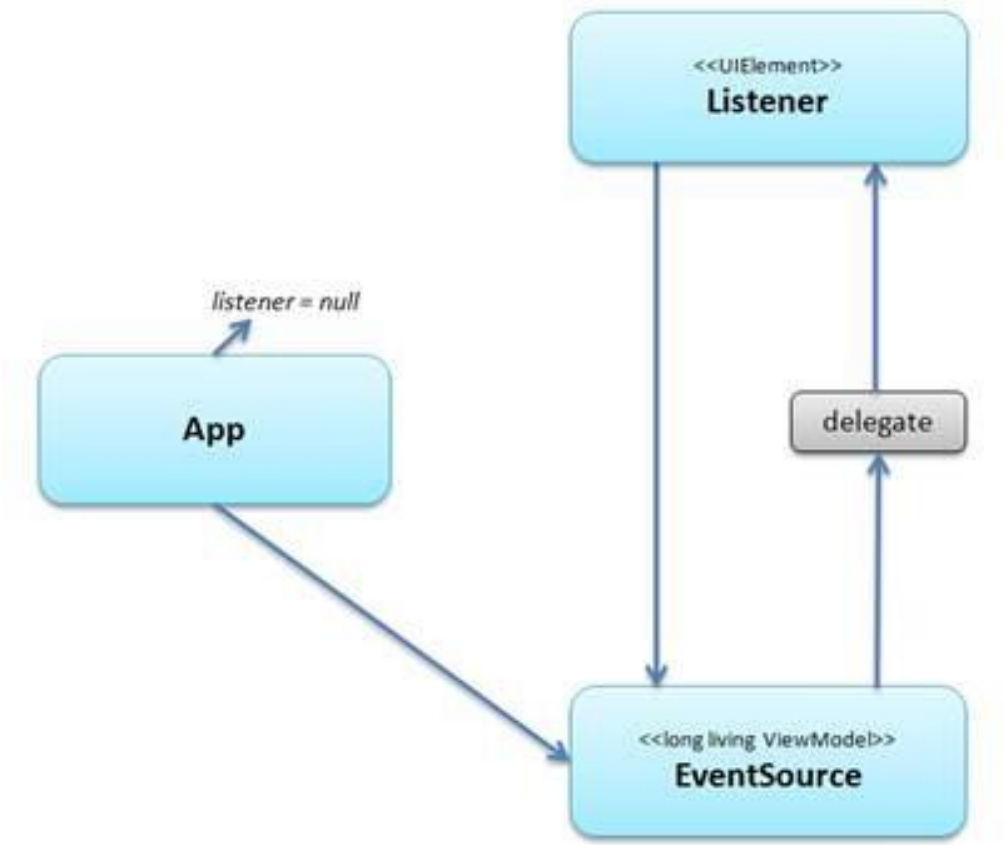
Le schéma ci-dessus nous montre le jeu entre source de l'évènement (EventSource en bas) et écouteur (ou abonné, Listener en haut). L'application représentant la "glue" qui permet à EventSource et Listener d'exister ensembles dans un tout cohérent.

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) !

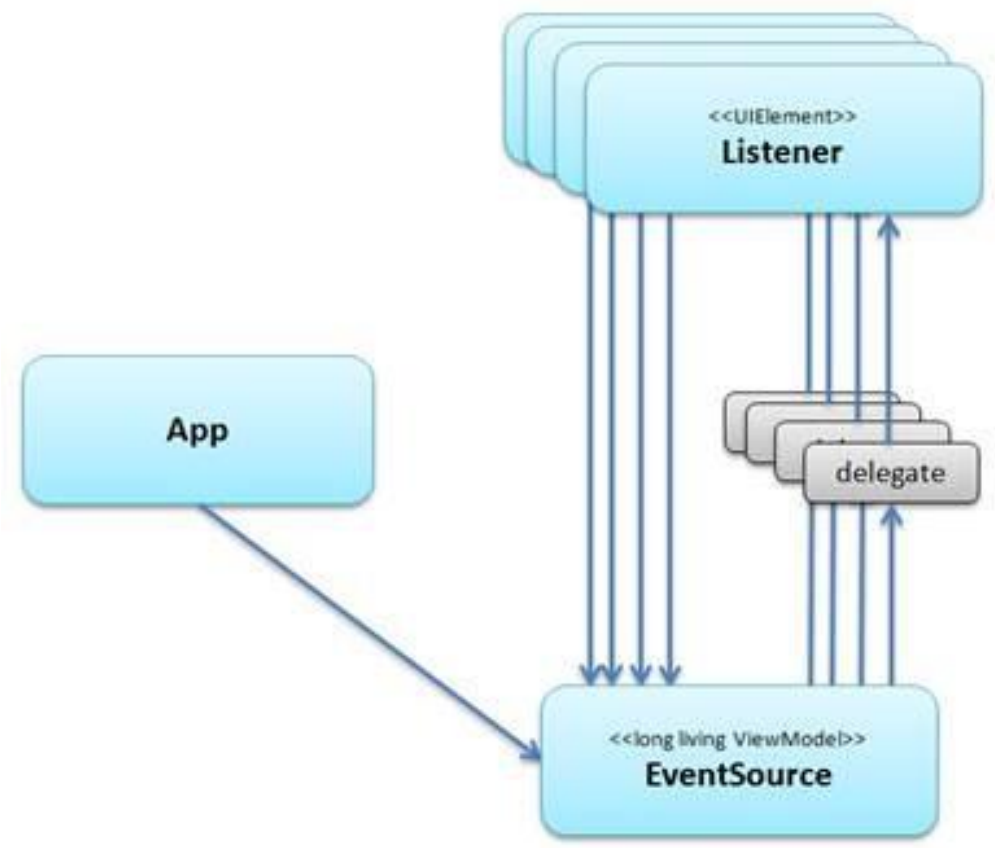


Comme on le voit ci-dessus, l'application ne possède plus de référence vers le Listener, mais il existe toujours bien une telle référence dans EventSource, c'est le delegate qui pointe la méthode du Listener à appeler lorsque l'évènement se produit !

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.

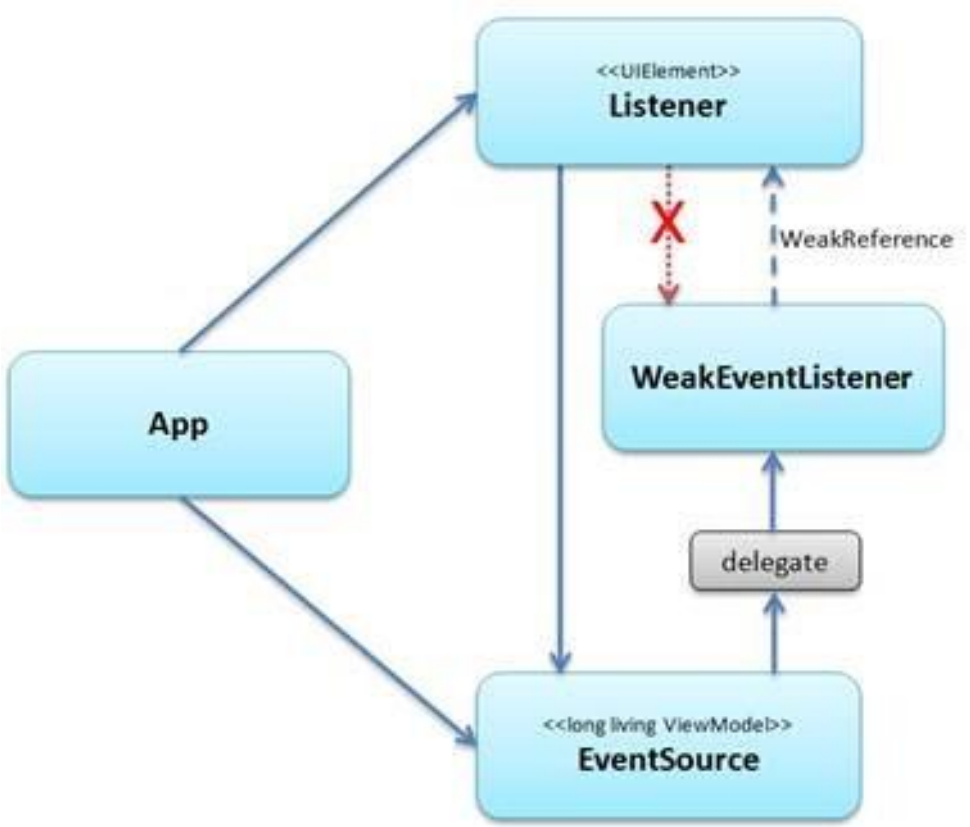
Les Weak References

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.

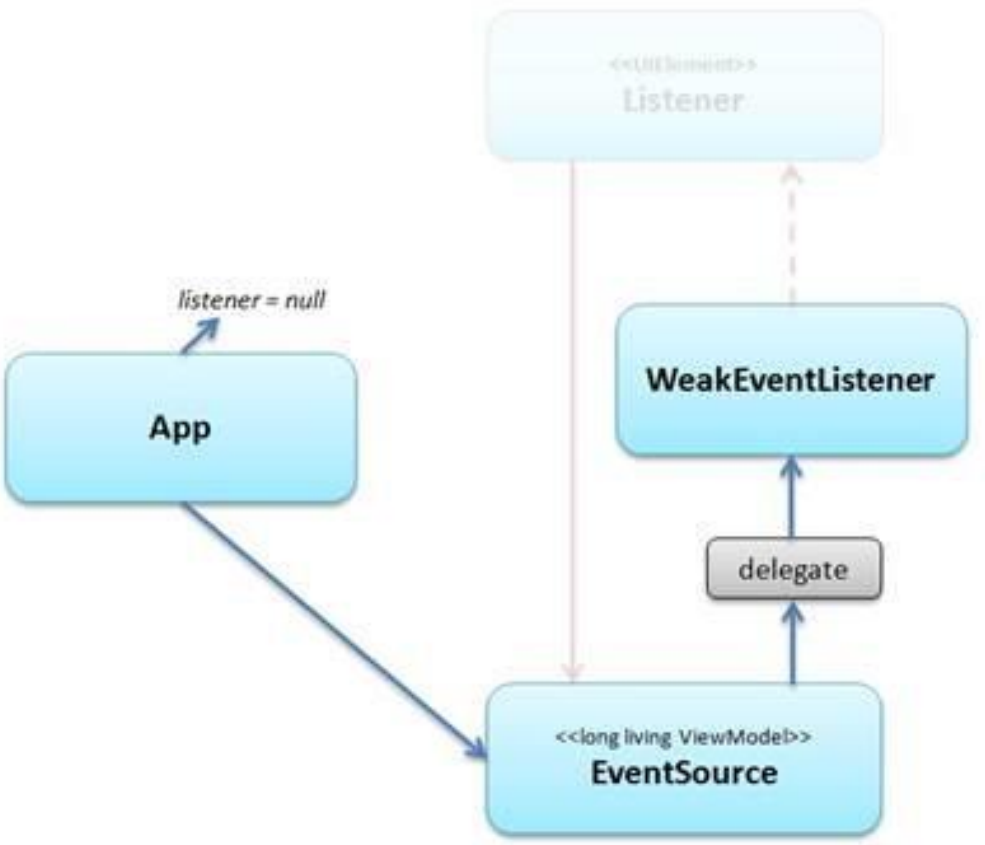
En Pratique

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 ce n'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, UWP…)

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

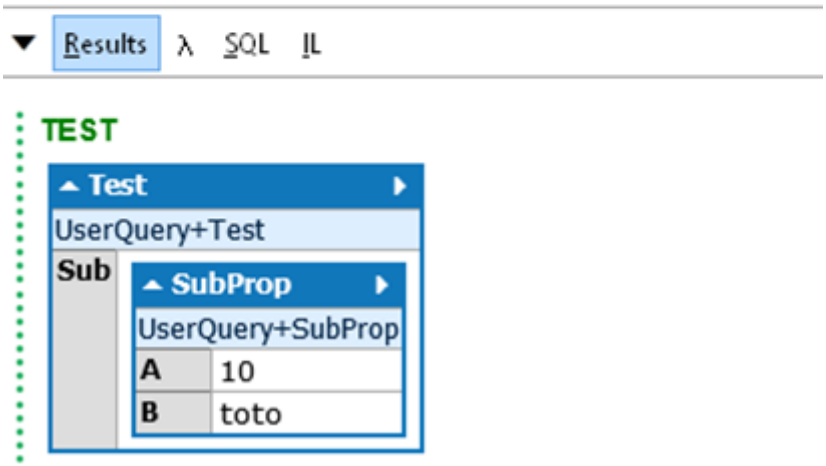
Le plus simple pour tester des petits trucs comme cela c'est d'utiliser LinqPad, un outil indispensable.

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 !).

Les espaces de noms statiques

Une simplification qui fera jaser car elle tend à entretenir la confusion entre les différentes méthodes de différentes classes… Et vous que pensez-vous de ces nouveaux espaces de noms statiques ?

.NET le pays des espaces de noms !

Même si la notion d'espaces de noms n'est pas une invention de .NET ce Framework a exploité à merveille le principe en mettant de l'ordre dans les centaines d'API Windows disponibles. Pour simplifier l'écriture .NET autorise l'utilisation de déclarations "using" en début de code pour lister les espaces de noms utilisés. Ce qui permet ensuite d'utiliser les classes de ces derniers sans avoir besoin de les préfixer. L'écriture s'en trouve allégée grandement. Et la lecture aussi ce qui est finalement le véritablement but.

Les classes statiques comme espaces de noms

La nouveauté de C# 6 est de permettre par une syntaxe similaire d'assimiler une classe statique à un espace de nom, ce qui conceptuellement ne pose aucun problème, le nom d'une classe dans un espace de noms n'est jamais qu'une feuille terminale dans l'arborescence des espaces de noms dont elle est issue.

Avec "using" on s'arrêtait au "namespace" qui précède la déclaration de la classe. Mais il faut avouer que les classes statiques obligeaient ensuite à des répétitions très semblables à ce qu'aurait été le code sans l'astuce des "using".

Ainsi on pouvait écrire jusqu'ici un code de ce type :

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

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("This is test Value");
        }
    }
}
```

Le mieux qu'on pouvait faire pour atteindre la classe statique Console c'était de déclarer la chaîne d'espaces de noms qui la précédait. Dans le cas de Console cela se limitait à "using System;" Pour d'autres classes la chaîne peut devenir très longue.

Malgré cette simplification à chaque utilisation d'une méthode de Console il fallait préfixer l'appel du nom de la classe statique comme le montre l'exemple ci-dessus.

Quand on doit écrire de nombreux appels à des méthodes d'une classe statique le code devient vite répétitif, ce qui endort la vigilance lors de sa lecture.

L'ajout de C# 6 consiste donc à considérer que les classes statiques, plutôt leur nom, font partie de la chaîne qui peut être déclarée par "using".

Le code de l'exemple devient ainsi :

```
using static System.Console;
namespace StaticNamespace
{
  class Program
  {
    static void Main(string[] args)
    {
       Write("This is Test Value");
       ReadKey();
    }
  }
}
```

On remarque la déclaration “using static System.Console;” ce qui permet l’utilisation de “Write” comme s’il s’agissait d’une méthode de la classe en cours (Program).

On peut bien entendu utiliser toute classe statique de cette façon. Par exemple en déclarant un “using static System.Environment” on peut accéder directement à “NewLine”. Le code :

```
Console.Write(Environment.Newline);
```

devient alors

```
Write(NewLine);
```

La lecture devient bien plus simple.

Enfer ou paradis ?

Comme toutes les simplifications syntaxiques il y a un risque de confusion. On voit encore des gens qui n’utilisent pas “var” et qui préfixent de toute la chaîne de namespaces la moindre déclaration pour s’assurer que leur intention est comprise et qu’aucun mélange ne risque de venir prêter à confusion.

Forcément en poussant le “using” jusqu’aux noms des classes statiques, en donnant l’impression qu’une méthode appartient à une classe dans laquelle elle n’est en réalité pas déclarée, il y a comme une confusion possible.

Mais elle est levée d’une part par la connaissance du Framework (mais ce n’est pas un argument suffisant je suis d’accord) et par Visual Studio qui permet en survolant un code de savoir exactement d’où provient une classe (mais cela reste caché à la simple lecture du texte).

Il n’en reste pas moins vrai que certains considèrerons cette façon d’écrire trop peu respectueuse des intentions et donc nuisant à la lisibilité du code (ce qui est un comble mais qui se justifie) et d’autres qui y verront une façon de justement rendre leur code plus lisible…

Conclusion

En partant du même constat on peut arriver à deux positions radicalement opposées… Alors lisible ou pas lisible les espaces de noms statiques ?

Je ne trancherai pas, à vous de dire ce que vous en pensez !

Connaissez-vous l'Expando ?

Le Framework .NET recèle des trésors. Souvent cachés, peu connus, ils peuvent se révéler de précieux auxiliaires pour se sortir de mauvaises passes. Connaissez-vous l'Expando ? Non ce n'est pas un gadget miraculeux vendu sur le web pour agrandir l'objet de votre virilité. Pas de ça ici. Alors c'est quoi ?

L'ExpandoObject

Caché dans System.Dynamic et totalement connecté à ce petit monde bien particulier, l'ExpandoObject est une classe bien facétieuse. Le genre de chose qu'on adore ou qu'on déteste, tout de suite, comme ça, sans réfléchir, instinctivement.

Le type Dynamic est arrivé avec .NET 4, nous en sommes à la 6ème version avec Roselyn et ces intéressantes nouveautés et notre ami l'expando est toujours inconnu…

Heureusement que je suis là pour combler vos lacunes !

C'est quoi ?

C'est un type. Donc en fait des instances. Comme Object. Au départ il n'est d'ailleurs pas plus utile que Object puisque ExpandoObject n'a ni propriété ni méthodes (en tout cas pas comme les autres classes qui "font quelque chose").

Un doublon de Object ? En quelque sorte, mais à la sauve Dynamic…

C'est à dire que les instances de ExpandoObject peuvent se voire ajouter à tout moment, dynamiquement au runtime donc, des propriétés et mêmes des sortes de méthodes !

Un bout de code

Prenons tout de suite un bon de code pour que vous compreniez l'affaire :

```
1. void Main()
2. {
3.     dynamic e = new ExpandoObject();
4.     e.FirstName = "Olivier";
5.     e.LastName = "Dahan";
6.     e.Print = new  Action( ()=> Console.WriteLine(e.FirstName+" "+
                                   e.LastName ));
7.     e.Print();
8. }
```

Etrange, non ?

L'instance "e" est donc un objet dynamic (il faut le définir clairement comme tel, un simple "var" ici ne marchera pas). Objet instancié par la création d'un nouvel ExpandoObject.

Jusque là rien de bien exotique malgré tout.

Mais que voit-on dans les lignes qui suivent ? On affecte une valeur à la propriété FirstName puis à LastName. Mais diable d'où sortent ces propriétés ?

De nulle part.

Vraiment. Nulle part. Elles sont créées dynamiquement.

Et la méthode Print() ? Elle est créée part l'instanciation d'une Action() qui contient le code à exécuter. Ce qui permet à la fin d'appeler e.Print() ce qui, vous l'avez deviné affichera glorieusement sur la console le nom de votre serviteur.

Utiliser la variable "e" à l'intérieur de l'action est un peu "cracra" je l'avoue. On pourrait choisir une écriture plus propre comme celle-ci :

```
1. void Main()
2. {
3.     dynamic e = new ExpandoObject();
4.     e.FirstName = "Olivier";
5.     e.LastName = "Dahan";
6.     e.Print = new  Action<dynamic>( (a)=> Console.WriteLine(
                                        a.FirstName+" "+a.LastName ));
7.     e.Print(e);
8. }
```

Du coup la méthode Print() prend désormais un paramètre de type dynamic. On peut peut lui passer "e" ou tout dynamic qui définirait les mêmes méthodes.

Vous avez compris le concept je suppose.

A quoi cela peut-il servir ?

.NET gère le boxing et l'unboxing depuis toujours. On pourrait donc écrire ce code en remplaçant e par un dictionnaire Dictionary<string,object> ce qui permettrait d'ajouter des "propriétés" (les clés du dictionnaire) et des valeurs (n'importe quoi puisqu'elles seront boxées sous la forme d'un Object).

Mais avouez que ça serait bien moins élégant comme écriture, plus verbeux. Et s'il faut manipuler plusieurs objets de ce type il faudra créer des dictionnaires de dictionnaires… Cela va vite devenir impossible à comprendre et à maintenir.

La maintenabilité et la lisibilité du code doivent primer sur toute autre considération dans un environnement professionnel. De très rares fois et pour des raisons qui restent peu nombreuses, les performances peuvent prendre le pas, dans une partie temps réel critique par exemple. Mais sinon maintenabilité et lisibilité passent avant tout.

Et aussi saugrenu que puisse paraitre l'ExpandoObject et sa loufoquerie congénitale – l'antéchrist caché dans un langage fortement typé qui accepte n'importe quoi reste un pied de nez intéressant aux sacro-saints principes de la programmation Objet – et bien aussi farfelu que cela puisse sembler, l'expando peut aider à rendre le code plus lisible et plus facilement maintenable. Et c'est pour cela que je parle de cet orphelin caché au fond du Framework comme un enfant adultérin qu'on escamoterait à la populace rongé par la honte…

Non, n'aie pas honte mon petit expando, tu n'est certes pas très beau, un peu tordu, mais tu as le droit toi aussi à ton article dans Dot.Blog !

Donc soyons clairs, les dynamics ça passe ou ça casse. J'avais prévenu, on adore ou aime d'instinct. *Haters gonna hate* comme disent les ricains sur les rézossocios et qu'on traduirait par "les haineux vont haïr" ou "les rouspéteurs vont détester" selon la "violence" ressentie dans le contexte…

Autre exemple

Dans la réalité on peut se retrouver devant des structures complexes. Je parlais de dictionnaires de dictionnaires plus haut, c'est un bel exemple.

Imaginons le code suivant :

```
1. Dictionary<String, object> dict = new Dictionary<string, object>();
2. Dictionary<String, object> address = new Dictionary<string,object>();

3. dict["Address"] = address;
4. address["State"] = "TX";
5. Console.WriteLine(((Dictionary<string,object>)dict["Address"])["State
   "]);
```

On a ici un dictionnaire de "choses" puis un dictionnaire d'adresses. Ce dernier étant répertorié dans le premier sous la clé "Address". Puis on créé une valeur dans les adresses en ajoutant la clé "State" et sa valeur "TX".

Si on veut atteindre cette valeur en partant du dictionnaire racine, ce qui est fait en dernière ligne, l'écriture est absolument imbuvable !

Imaginons maintenant un lecteur de Dot.Blog qui vient de passer par ici. Il écrira devant les yeux ébahis et les neurones médusés de ses collègues :

```
1. dynamic expando = new ExpandoObject();
2. expando.Address = new ExpandoObject();
3. expando.Address.State = "TX";
4. Console.WriteLine(expando.Address.State);
```

D'une horreur on obtient un code lisible. Fonctionnant de la même façon, tout aussi sophistiqué (des dictionnaires de dictionnaires ce n'est déjà plus très simple), mais clair. Donc maintenable.

Et mieux qu'un simple dictionnaire ExpandoObject() possède bien entendu des propriétés et des méthodes et même des évènements et des méthodes d'extension. Et parmi cette panoplie (qui le rend bien plus subtil qu'un simple Object) on trouve notre ami INPC dont j'ai parlé encore il y a peu.

Regardez ce code issu d'une réponse sur un forum de Alexandra Rusina qui avait écrit sur MSDN un article sur les Expando :

```
1. class Program
2. {
3.
4.     static void Main(string[] args)
5.     {
6.         dynamic d = new ExpandoObject();
7.
8.         // Initialize the event to null (meaning no handlers)
9.         d.MyEvent = null;
10.
11.             // Add some handlers
12.             d.MyEvent += new EventHandler(OnMyEvent);
13.             d.MyEvent += new EventHandler(OnMyEvent2);
14.
15.             // Fire the event
16.             EventHandler e = d.MyEvent;
17.
18.             if (e != null)
19.             {
20.                 e(d, new EventArgs());
21.             }
22.
23.             // We could also fire it with...
24.             //      d.MyEvent(d, new EventArgs());
25.
```

```
            // ...if we knew for sure that the event is non-null.
        }

        static void OnMyEvent(object sender, EventArgs e)
        {
            Console.WriteLine("OnMyEvent fired by: {0}", sender);
        }

        static void OnMyEvent2(object sender, EventArgs e)
        {
            Console.WriteLine("OnMyEvent2 fired by: {0}", sender);
        }
    }
```

Saurez-vous prévoir la sortie ?

Ce sera :

```
OnMyEvent fired by: System.Dynamic.ExpandoObject
OnMyEvent2 fired by: System.Dynamic.ExpandoObject
```

Conclusion

L'expando (j'aime l'appeler par ce petit nom charmant aux consonances de héro vengeur latino-américain), ce brave petit n'est pas qu'une verrue atroce sur le nez grec de la programmation objet, il n'est pas cet enfant du malin mettant le vers gluant du non-typage dans la pomme fortement typée de C# pour le pervertir. Un peu quand même. Mais pas totalement.

Il peut exister des situations bien réelles où l'utilisation de l'expando rend le code plus lisible et plus maintenable.

Et puis on dispose d'une classe qui au runtime peut se construire petit à petit, propriétés, actions, support de INPC. Presque tout ce qu'il faut pour créer à la volée des ViewModels par exemple. Dans une architecture donnée qui suivrait un paramétrage complexe type gros ERP, cela pourrait même permettre d'écrire le code en 2 ans de moins peut-être… Des candidats pour détrôner SalesForce ou WaveSoft ?

En tout cas l'expando ne laisse pas indifférent. Et c'est bien.

Rester un bon développeur c'est surtout ne pas s'inscrire dans une routine qui endort les neurones. Douter, aimer, haïr, rester passionné donc, est l'aiguillon qui conserve l'esprit en éveil !

Les Weak References

Rares sont les développeurs à utiliser les Weak References qui sont là depuis le début de .NET et sont pourtant très utiles. Bien les connaitre et savoir les utiliser permet de créer du code faiblement couplé même lorsqu'il gère des références à d'autres objets. Mais pas que…

Les Weak References

Dans un environnement managé comme .NET la gestion des références à des instances de classe semble naturelle et sans poser de souci. Un objet est soit référencé, donc « vivant », soit n'est plus référencé, donc « mort » et éligible à sa destruction par le Garbage Collector (plus loin noté GC).

Si cela correspond le plus souvent au besoin, il est des cas où l'on voudrait conserver une référence sur un objet sans pour autant interdire son éventuelle libération. C'est le cas d'un cache d'objets par exemple : il référence des objets et peut les servir si besoin est, mais si la mémoire manque et que certains ont été détruits entre temps, cela n'est pas grave, ils seront recréés. Or, dans un environnement managé comme Java ou .NET, tant qu'une référence existe sur un objet ce dernier ne peut pas – par définition d'un environnement managé – être libéré. Dilemme… C'est là qu'interviennent les références faibles, ou Weak References.

Définition

> ***Une référence faible est une référence à un objet qui bien qu'elle en autorise l'accès n'interdit pas la possible suppression de ce dernier par le Garbage Collector.***

En clair, cela signifie qu'une référence faible, d'où sa « faiblesse », ne créé pas un lien fort avec l'instance référencée. C'est une référence qui, du point de vue du système de libération des instances n'existe pas, elle n'est pas prise en compte dans le graphe des chemins des objets « accessibles » par le GC.

C'est en fait comme cela que tout fonctionne en POO classique dans des environnements non managés comme Win32, rien n'interdit une variable de pointer un objet qui a déjà été libéré, mais tout accès par ce biais se soldera par une violation d'accès.

Sous environnement non managé il n'existe pas de solution simple pour éviter de telles situations, d'où l'engouement des dernières années pour les environnements

managés comme .NET ou Java sans qu'aucun retour en arrière ne semble désormais possible.

En effet, sous .NET une telle situation ne peut tout simplement pas arriver puisqu'on ne libère pas la mémoire explicitement, c'est le CLR qui s'en charge lorsqu'il n'y a plus de référence à l'objet.

Il ne faut d'ailleurs pas confondre mémoire et ressources externes. .NET protège la mémoire, pas les ressources externes. Pour cela les classes doivent implémenter IDisposable. Et si un objet a été «disposé» il n'est pas « libéré » pour autant. On peut donc continuer à l'utiliser mais cela créera le plus souvent une erreur d'exécution puisque les ressources externes auront été libérées entre temps… Prenez une instance de System.Drawing.Font, appelez sa méthode Dispose(). Vous pourrez toujours accéder à l'objet en tant qu'entité, mais si vous tenter d'appeler sa méthode ToHFont() qui retourne le handle de l'objet fonte sous-jacent, une exception sera levée… Il existe une nuance importante entre mémoire et ressources externes, entre libération d'un objet et libération de ses ressources externes. C'est là l'une des difficultés du modèle objet de .NET qui pose souvent des problèmes aux débutants, et parfois même à des développeurs plus confirmés.

Le mécanisme

Le Garbarge Collector du CLR libère la mémoire de tout objet qui ne peut plus être atteint. Un objet ne peut plus être atteint quand toutes les références qui le pointent deviennent non valides, par exemple en les forçant à null. Lorsqu'il détruit les objets qui se trouvent dans cette situation le GC appelle leur méthode Finalize, à condition qu'une telle méthode soit définie et que le GC en ait été informé (le mécanisme réel est plus complexe et sort du cadre de cet article).

Lorsqu'un objet peut être directement ou indirectement atteint il ne peut pas être supprimé par le GC. Une référence vers un objet qui peut être atteint est appelée une référence forte.

Une référence faible permet elle aussi de pointer un objet qui peut être atteint qu'on appelle la cible (target en anglais). Mais cette référence n'interfère pas avec le GC qui, si aucune référence forte n'existe sur l'objet, peut détruire ce dernier en ignorant les éventuelles références faibles (elles ne sont pas totalement ignorées puisque, nous allons le voir, la référence faible sera avertie de la destruction de l'objet).

Les références faibles se définissent par des instances de la classe WeakReference. Elle expose une propriété Target qui permet justement de réacquérir une référence forte sur la cible. A condition qu'elle existe encore… C'est pour cela que cette classe

offre aussi un moyen de le savoir par le biais de sa propriété IsAlive (« est-il encore vivant ? »).

Pour un système de cache, comme évoqué en introduction, cela est très intéressant puisqu'on peut libérer un objet (plus aucune référence valide ne le pointe) et malgré tout le récupérer dans de nombreux cas si le besoin s'en fait sentir. Cela est possible car entre le moment où un objet devient éligible pour sa destruction par le GC et le moment où il est réellement collecté et finalisé il peut se passer un temps non négligeable !

Le GC utilise trois « générations », trois conteneurs logiques. Les objets sont créés dans la génération 0, lorsqu'elle est pleine le GC supprime tous les objets inutiles et déplace ceux encore en utilisation dans la génération 1. Si celle-ci vient à être saturée le même processus se déclenche (nettoyage de la génération 1 et déplacement des objets encore valides dans la génération 2 qui représente tout le reste de la RAM disponible).

De fait, un objet qui a été utilisé un certain temps se voit pousser en génération 2, un endroit qui est rarement visité par le GC. Parfois même, s'il y a beaucoup de mémoire installée sur le

PC ou si l'application n'est pas très gourmande, les objets de la génération 2, voire de la génération 1, ne seront jamais détruits jusqu'à la fermeture de l'application… A ce moment précis le CLR videra d'un seul coup tout l'espace réservé sans même finaliser les objets ni appeler leur destructeur. C'est pourquoi sous .NET on ne programme généralement pas de destructeurs dans les classes : le mécanisme d'appel à cette méthode n'est pas déterministe.

Donc, durant toute la vie de l'application de nombreuses instances restent malgré tout en vie « quelque part » dans la RAM. Si une référence faible pointe l'un de ces objets il pourra donc être « récupéré » en réacquérant une référence forte sur lui par le biais de la propriété Target de l'objet WeakReference. Si l'objet en question réclame beaucoup de traitement pour sa création, il y a un énorme avantage à le récupérer s'il doit resservir au lieu d'avoir à le recréer, le tout sans pour autant engorger la mémoire puisque, si nécessaire, le GC l'aura totalement libéré, ce qu'on saura en interrogeant la propriété IsAlive de l'objet WeakReference qui aura alors la valeur false.

Si vous vous souvenez de ce que je disais plus haut sur la nuance entre libération d'une instance et libération de ses ressources externes, vous comprenez que l'application ne doit utiliser des références faibles que sur des objets qui

n'implémentent pas IDisposable. En effet, pour reprendre l'exemple d'une instance de la classe Font, si avant de mettre la référence à null votre application a appelé sa méthode Dispose(), récupérer plus tard l'instance grâce à une référence faible sera très dangereux : les ressources externes sont déjà libérées et toute utilisation de l'instance se soldera par une exception. Il est donc important de se limiter à des classes non disposables.

L'intérêt

Les références faibles ne servent pas qu'à mettre en œuvre des systèmes de cache, elles servent aussi lorsqu'on doit pointer des objets qui peuvent et doivent éventuellement être détruits. Rappelons-nous : si nous utilisons une simple référence sur un tel objet, il ne sera jamais détruit puisque justement nous le référençons… Les références faibles permettent d'échapper à ce mécanisme par défaut qui, parfois, devient une gêne plus qu'un avantage.

Un exemple d'une telle situation : Supposons une liste de personnes. Cette liste pointe donc des instances de la classe Personne. Imaginons maintenant que l'application autorise la création de « groupes de travail », c'est-à-dire des listes de personnes. Si les listes définissant les groupes de travail pointent directement les instances de Personne et si une personne est supprimée de cette liste, les groupes de travail continuerons de « voir » cette personne puisque l'instance étant référencée (référence forte) dans le groupe de travail elle ne sera pas détruite par sa simple suppression de la liste de base des personnes…

En fait, on souhaitera dans un tel cas que toute personne supprimée de la liste principale n'apparaisse plus dans les groupes de travail dans lesquels elle a pu être référencée.

Cela peut se régler par une gestion d'évènement : toute suppression de la liste des personnes entraînera le balayage de tous les groupes de travail pour supprimer la personne. Cette solution n'est pas toujours utilisable. Les références faibles deviennent alors une alternative intéressante, notamment parce qu'il n'y a pas besoin d'avoir prévu un lien entre la liste principale et les listes secondaires qui peuvent être ajoutées après coup dans la conception de l'application et parce que la suppression d'une personne n'impose pas une attente en raison du balayage de toutes les listes secondaires (ou autre mécanisme similaire). Gain de temps, de performance, meilleure utilisation de la mémoire, faible couplage, bref les Weak References ont un intérêt réel !

Mise en œuvre

Il est temps de voir comment implémenter les références faibles.

Finalement, vous allez le constater, c'est assez simple. Les explications qui précèdent permettent de comprendre pourquoi les références faibles sont utiles. Les utiliser réclame moins de mots…

La classe WeakReference

Cette classe appartient à l'espace de nom System du framework. Son constructeur prend en paramètre l'instance que l'on souhaite référencer (la cible).

Elle expose trois propriétés caractéristiques, les autres propriétés et méthodes étant celles héritées de la classe mère System.Object :

IsAlive Indique si l'instance référencée est vivante ou non.

Target Permet de réacquérir une référence forte sur la cible.

TrackResurrection Pour dés/activer le pistage de résurrection de la cible.

Un mot sur cette dernière propriété : Lorsque l'on créé une référence faible on peut indiquer dans le constructeur, en plus de l'objet ciblé, un paramètre booléen qui fixera la valeur de TrackResurrection. Lorsque la valeur est « false » (par défaut) on parle de référence faible « courte », lorsque la valeur est « true » on parle de référence faible « longue ». Si l'objet possède un finaliseur, c'est dans celui-ci qu'il sera possible d'indiquer ou non si l'instance doit rester en vie (être ressuscitée) ou pas, notamment par un appel à GC.ReRegisterForFinalize(this). L'intérêt se trouve surtout dans les gestions de cache, car un objet « finalisable » survivra à au moins un cycle du GC en étant promu de la génération 0 à la génération 1. En retardant sa finalisation il sera poussé en génération 2 où il restera certainement un bon moment, améliorant ainsi grandement les chances de pouvoir le récupérer plus tard, donc rendant la gestion du cache encore plus efficace.

Le code

Le code qui suit est auto-documenté par sa simple exécution. S'agissant de projets console il vous suffit de créer une nouvelle application de ce type (que ce soit sous VS ou LinqPad) et de faire un copier / coller du code proposé. Lancez l'exécution et laissez-vous guider à l'exécution en jetant un œil sur le code…

```
1. public class Personne
2. {
3.     private string nom;
4.     public string Nom
5.     {
```

```
6.          get { return nom; }
7.          set { nom = (value != null) ? value.Trim() : string.Empty; }

8.      }
9.
10.          public Personne(string nom)
11.          { this.nom = nom.Trim(); }
12.      }
13.
14.
15.      public static ArrayList Employés;
16.      private static string line = new string('-', 60);
17.
18.      private static void Return()
19.      {
20.          Console.WriteLine("<return> pour continuer...");
21.          Console.ReadLine();
22.      }
23.
24.      public static void ListeEmployés()
25.      {
26.          Console.WriteLine(line);
27.          foreach (Personne p in Employés)
28.              Console.WriteLine(p.Nom);
29.          Console.WriteLine(line);
30.          Return();
31.      }
32.
33.      static void Main(string[] args)
34.      {
35.          Employés = new ArrayList();
36.          Employés.Add(new Personne("Olivier"));
37.          Employés.Add(new Personne("Barbara"));
38.          Employés.Add(new Personne("Jacky"));
39.          Employés.Add(new Personne("Valérie"));
40.
41.          Console.WriteLine("Liste originale");
42.          ListeEmployés();
43.
44.          Personne p = (Personne)Employés[0]; // pointe "olivier"
45.          Employés.RemoveAt(0); // suppresson de "olivier" dans la li
    ste
46.          Console.WriteLine("p pointe : " + p.Nom);
47.          Console.WriteLine("L'élément 0 de la liste a été supprimé")
    ;
48.          Console.WriteLine();
49.          ListeEmployés();
50.          Console.WriteLine("Mais l'objet pointé par p existe toujour
    s : " + p.Nom);
51.          Return();
52.
53.          WeakReference wr = new WeakReference(Employés[0]); // point
    e "barbara"
```

```
54.          Console.WriteLine("wr est une référence faible sur: " + ((P
   ersonne)wr.Target).Nom);
55.          Return();
56.
57.          Console.WriteLine("La cible de wr est vivante ? : " + wr.Is
   Alive.ToString());
58.          Return();
59.          Employés.RemoveAt(0); // suppression de "barbara"
60.          Console.WriteLine("L'élément 0 ('barbara') a été supprimé.
   La liste devient :");
61.          ListeEmployés();
62.
63.          Console.WriteLine("La cible de wr est vivante ? : " + wr.Is
   Alive.ToString());
64.          Console.WriteLine("On peut réacquérir la cible : " + ((Pers
   onne)wr.Target).Nom);
65.          Return();
66.
67.          Console.WriteLine("Mais si le GC passe par là...");
68.          GC.Collect(GC.MaxGeneration);
69.          Console.WriteLine("La cible de wr est vivante ? : " + wr.Is
   Alive.ToString()); // false !
70.          Console.WriteLine("La référence faible n'a pas interdit sa
   destruction totale.");
71.          Return();
72.      }
```

La sortie console ressemble à cela :

```
Liste originale

------------------------------------------------------------

Olivier

Barbara

Jacky

Valérie

------------------------------------------------------------

<return> pour continuer...

p pointe : Olivier

L'élément 0 de la liste a été supprimé


------------------------------------------------------------

Barbara
```

```
Jacky
Valérie
------------------------------------------------------------
<return> pour continuer...
Mais l'objet pointé par p existe toujours : Olivier
<return> pour continuer...
wr est une référence faible sur: Barbara
<return> pour continuer...
La cible de wr est vivante ? : True
<return> pour continuer...
L'élément 0 ('barbara') a été supprimé. La liste devient :
------------------------------------------------------------
Jacky
Valérie
------------------------------------------------------------
<return> pour continuer...
La cible de wr est vivante ? : True
On peut réacquérir la cible : Barbara
<return> pour continuer...
Mais si le GC passe par là...
La cible de wr est vivante ? : False
La référence faible n'a pas interdit sa destruction totale.
<return> pour continuer...
```

Conclusion

Les Weak References permettent d'écrire un code souple et découplé sans perdre le bénéfice évident de pouvoir référencer des objets, mais sans le risque de les voir tous perdurer en mémoire et l'engorger inutilement. Sans avoir besoin de mettre en place des mécanismes de surveillance des objets pointés qui ralentirait et complexifieraient le code il est possible d'avoir l'assurance de retrouver une instance (ou pas) sans bloquer sa collecte par le GC.

Cela est souvent très utile quand on écrit une librairie de code. On doit parfois garder des références sur des objets qu'on ne créée pas (ils sont créés par le développeur qui utilise votre librairie) sans gêner le code utilisateur. C'est le cas par exemple de MVVM Light qui utilise des Weak References dans les RelayCommand et qui pousse même jusqu'à utiliser des "weak events" pour gérer des évènements faiblement couplés.

Les références entre les objets font que même les langages managés peuvent présenter des problèmes de gestion de mémoire. Les Weak References permettent de contourner ces problèmes de façon élégante.

Ne pas connaitre les Weak References c'est soit écrire un code de niveau très faible soit faire des grosses bêtises à un moment ou un autre, et sans le savoir !

Faible ? c'est votre code qui peut l'être sans les Weak References…

Après les Weak References, les Weak Events !

Tous faibles pour une code plus fort ! Tel pourrait être la devise des Weak References et des Weak Events ! Mais qu'est-ce que les Weak Events ? Et à quoi cela sert-il ? Et comment les mettre en œuvre ? Plein de questions auxquelles je vais répondre…

La notion "Weak"

Weak, prononcer "ouik", veut dire "faible". A quoi est du cette "faiblesse" ?

En matière de code vous l'avez compris le dogme qui s'est imposé, non sans bonnes raisons, est celui du "decoupled" (découplé). Quand on dit découplé, on s'oppose à couplé, logique.

Le couplage consiste à lier deux choses entre elles.

Si j'ai deux objets a et b et que dans la définition de la classe A dont est issue l'instance a j'ai un code du type "private B instanceDeB = new B();" cela signifie que les instances a de A se lient à une instance de B.

Ce type de couplage peut être trivial comme cet exemple ou être plus indirect, le résultat étant le même, l'instance a de type A possède une référence sur une instance b de B.

Ce type de couplage est dit "fort", non pour qualifier sa valeur intrinsèque mais juste pour signifie que le lien est fort.

Les environnements managés et les liens forts

La base même du principe des environnements managés comme .NET repose sur le fait que le développeur n'a plus à se préoccuper de la libération des objets (ce n'est pas aussi simple mais on considèrera cela comme vrai en première approximation).

Pour ce faire le code qui est exécuté est "surveillé" par une couche, le CLR dans .NET. Et lorsque cela est nécessaire un objet particulier de .NET fait le ménage, c'est le Garbage Collector. Je schématise beaucoup mais c'est l'essentiel du processus.

Pour faire son travail le GC utilise un "graphe des objets", c'est à dire qu'il se fait une représentation mémoire des liens entre chaque objet pour supprimer ceux qui n'ont plus de liens avec les autres car dans ce cas cela signifie qu'ils n'ont plus d'utilité. Le raisonnement est simple (et faillit quelques fois mais c'est une autre histoire dont je parlerai un jour certainement).

Tous les liens forts permettent au GC de relier des objets entre eux dans son graphe. Ces objets là seront conservés (poussés de la zone dite génération 0 à celle appelée génération 1 puis pour les objets à très longue durée de vie dans la génération 2 qui est scrutée tellement moins souvent que souvent les objets même libérés y vivent jusqu'à l'arrêt de l'application et sont détruits sans appel à leur finalseur).

Cette façon de procéder a démontré sa supériorité sur les langages non managés, les informaticiens ayant gardé un très mauvais souvenir des bogues que des langages tels que C facilitent. Les pointeurs fous, les zones mémoire partiellement libérées, les débordements de buffers, etc, sont même à la base de la plupart des malwares, des attaques de sites web, sans parler des écrans bleus de la mort et ce encore aujourd'hui puisque des acharnés restent scotchés à ces langages primitifs et dangereux.

Well, tout est beau et chouette dans le monde managé… Sauf un petit détail : tous les objets n'ont pas vocation à avoir une vie infinie et tous ne possèdent pas un point unique de responsabilité de gestion de leur cycle de vie. Leur destruction, via des patterns comme Dispose ou autres, ne peut donc pas être encadrée et perd son caractère déterministe. Tout repose ainsi sur l'intelligence du GC pour faire le ménage. Intelligence très limitée.

Or, même comme cela les choses ne sont pas si simples…

Dans le cas de l'exemple évoqué plus haut, pour que l'instance de B puisse être collectée par le GC il faut absolument que l'instance de A relâche la référence qu'elle possède sur cette dernière. Donc sans logique particulière l'instance de B vivra aussi longtemps que celle de A qui la possède.

Si le besoin de référencer b dans a est une obligation durant toute la vie de a, cela n'est pas gênant. Mais si l'utilisation que a fait de b n'a de sens que ponctuellement, b engorgera la mémoire pour rien l'essentiel du temps.

Les applications managées ont ainsi une tendance fâcheuse à consommer de la RAM plus qu'il n'en faudrait. Sauf si le code est parfaitement développé, mais la perfection n'étant pas de ce monde il y a toujours un peu de perte. Ce qui n'est pas grave en soi

car la plupart des développeurs aujourd'hui travaillent pour des machines gorgées de RAM.

Pour les applications utilisées sporadiquement dans une journée cela ne pose aucun problème en général. Pour des services Windows tournant en permanence les problèmes peuvent survenir comme pour de très grosses applications brassant beaucoup de données.

D'où l'importance même en managé d'apporter un soin particulier à l'écriture des classes et des relations qu'elles entretiennent, d'implémenter Dispose même s'il n'y a pas de référence extérieur pour permettre le nettoyage des références managées, ajouter une méthode de type Close() ou équivalent pour marquer la fin de l'utilisation d'une instance et nettoyer les références qu'elle possède, etc…

Affaiblir les liens

Donc pour pallier ce problème il faut affaiblir les liens entre les instances.

Mais ce n'est qu'une phrase, en réalité cela veut dire quoi ? Comment rendre un lien "faible" ?

Il faut d'abord penser qu'il y a deux types de liens dans un environnement comme .NET : les références entre objets, comme l'exemple utilisé jusqu'ici, et les évènements. C'est un peu simplificateur mais du point de vue du développeur ce sont ces deux situations qui vont lui poser problème.

Pour ce qui est des liens de type référence je vous ai déjà parlé des Weak References et je vous incite à lire cet article si ce n'est pas déjà fait.

Concernant les évènements .. c'est le sujet du jour !

Les Weak Events

Comme nous l'avons vu les liens forts entre deux objets peuvent être "affaiblis" par l'utilisation des Weak References. Ce mécanisme permet à une instance a de A de référencer une instance b de type B d'une telle façon que b peut être collecté à tout moment. La classe A est écrite pour savoir se débrouiller dans ce cas là et l'utilisation des Weak References lui permet de savoir si l'instance b existe toujours ou non.

Pour les évènements l'affaire est beaucoup moins simple. Quand a référence l'instance b, c'est clair. L'instance a est responsable de ce lien qu'elle noue avec b.

Mais dans la gestion des évènements le lien est plus indirect.

Si on pose maintenant qu'il n'y a plus de lien direct entre A et B et que la classe B expose un évènement E, l'instance a de type A doit posséder une méthode M et l'inscrire auprès de b pour que b puisse l'appeler lorsque cela sera nécessaire. Cette fois-ci a n'est plus responsable de tout. Au contraire, c'est b qui pour invoquer M doit garder une référence sur a. Ce mécanisme est évident mais il n'est matérialisé par aucun morceau de code.

Quand a s'abonne à l'évènement E de b, il passe la référence à sa méthode M, mais cette référence inclut implicitement une référence à a…

Ce qui créée un lien fort entre b et a.

b possède désormais une référence forte sur a par l'intermédiaire de celle qu'elle a enregistrée sur M.

Ce circuit plus complexe rend forcément le problème des références fortes plus difficile à gérer que dans le cas des références directes entre deux objets.

Les problèmes soulevés sont nombreux et sont la cause de fuites mémoires sous .NET.

On croyait avoir résolu le problème des fuites en passant au managé mais ce n'est pas totalement vrai… Il faut ici aussi développer un soin particulier pour éviter de perdre des bouts de mémoire, ce qui revient un peu au même que les pointeurs non libérés en C… Enfin pas tout à fait. Car ici nous connaissons le mécanisme responsable de ces pertes, toujours le même, celui lié aux références fortes alors qu'en C on ne peut pas dégager de solution globale puisque c'est inhérent au code lui-même et à sa logique (l'oubli de libération d'un pointeur ou libérer une quantité de mémoire différente de celle allouée ne peuvent être réglés en C par du C, il faut passer au managé justement).

Il existe des références de deux types et le problème est réglé pour les références directes avec les Weak References. Reste donc à régler le problème pour les évènements (et plus généralement pour tout ce qui fonctionne sur le mode d'un abonnement comme les Commandes sous MVVM, raison pour laquelle MVVM Light utilise des Weak Events dans son implémentation de RelayCommand).

Avant .NET 3.5

Ce n'était pas la préhistoire mais concernant les Weak Events il fallait le faire à la main. Le Framework ne prévoyait pas de solution particulière. Il faut comprendre qu'à cette époque les évènements sont principalement utilisés pour liés par exemple le Click d'un bouton à du code-behind majoritairement sous Windows Forms. Le code behind existe aussi longtemps que la Form existe, ce lien fort entre ces deux objets ne posait donc pas de souci en soi.

Les problèmes sont apparus dès lors qu'on a mis l'accent sur la programmation découplée et des architectures basées sur ce concept du couplage faible, comme MVVM avec Silverlight ou WPF.

A Partir de .NET 3.5

Le problème devient assez voyant et le Framework propose alors sa solution.

Il s'agit d'un couple :

- WeakEventManager
- IWeakEventListener

C'est donc une solution en deux étapes un peu compliquée à comprendre.

En gros il faut créer un évènement personnalisé en dérivant de WeakEventManager puis dans la casse qui écoute supporter l'interface IWeakEventListener.

C'est assez lourd et c'est heureusement dépassé.

A partir de .NET 4.5

Le besoin étant toujours aussi pressant mais la version en kit de .NET 3.5 n'ayant pas convaincu, il fallait que le Framework nous propose un peu mieux. C'est avec la version 4.5 que la bonne nouvelle arrivera sous la forme d'un WeakEventManager, le même oui, mais en générique ce qui change tout !

Avec `WeakEventManager<TEventSource,TEventArgs>` l'écriture devient autrement plus légère puisqu'il n'y a plus besoin d'écrire un descendant de WeakEventManager pour chaque type d'évènement. Mieux encore, la source de l'évènement n'a plus besoin d'implémenter l'interface IWeakEventListener !

On retrouve ainsi une gestion d'évènement presque aussi légère que la version non protégée contre les fuites mémoires.

Ce qui peut se résumer par le petit exemple ci-dessous :

```
1.  void Main()
2.  {
3.
4.      var source = new SourceDEvenement();
5.      Ecouteur listener = new Ecouteur(source);
6.
7.      source.Raise();
8.
9.      Console.WriteLine("Ecouteur à null:");
10.     listener = null;
11.
```

```
12.     ViderMemoire();
13.
14.     source.Raise();
15.
16.     Console.WriteLine("Source à null :");
17.     source = null;
18.
19.     ViderMemoire();
20. }
21.
22. static void ViderMemoire()
23. {
24.     Console.WriteLine("**Vidage en cours**");
25.
26.     GC.Collect();
27.     GC.WaitForPendingFinalizers();
28.     GC.Collect();
29.
30.     Console.WriteLine("**Nettoyé!**");
31. }
32.
33.
34. public class SourceDEvenement
35. {
36.     public event EventHandler<EventArgs> Event = delegate { };
37.
38.     public void Raise()
39.     {
40.         Event(this, EventArgs.Empty);
41.     }
42. }
43.
44. public class Ecouteur
45. {
46.     private void OnEvent(object source, EventArgs args)
47.     {
48.         Console.WriteLine("L'écouteur à reçu un évènement de la source.");

49.     }
50.
51.     public Ecouteur(SourceDEvenement source)
52.     {
53.         WeakEventManager<SourceDEvenement, EventArgs>.AddHandler(source, "E
    vent", OnEvent);
54.     }
55.
56.     ~Ecouteur()
57.     {
58.         Console.WriteLine("Finaliseur de l'écouteur exécuté.");
59.     }
60. }
```

Code qui va produire la sortie suivante (testé sous LinqPad) :

```
L'écouteur à reçu un évènement de la source.
Ecouteur à null:
**Vidage en cours**
Finaliseur de l'écouteur exécuté.
**Nettoyé!**
Source à null :
**Vidage en cours**
**Nettoyé!**
```

Comme on le voit le code est très simple puisqu'une seule ligne permet à l'écouteur de s'abonner à la source d'évènement.

En procédant de cette façon plus aucun risque de fuite mémoire, si l'écouteur disparait (cela est forcé dans l'exemple ci-dessus) la référence n'est plus maintenue par la source et l'appel à Raise sur cette dernière ne provoque pas non plus d'erreur d'exécution. L'abonné a disparu, la source n'en a que faire et poursuit son existence paisiblement…

Conclusion

Les Weak Events de .NET 4.5 sont une grande simplification de ce qui était proposé avant.

Les utiliser est tellement simple qu'il n'y a plus d'excuse aux memory leaks d'antan, et pourtant les Weak Events sont encore très peu utilisés.

Mais après cette saine lecture vous serez plus vigilants j'en suis certain !

Programmation défensive : faut-il vraiment tester les paramètres des méthodes ?

La programmation défensive c'est bien. Mais faut-il vraiment tester la validité de tous les paramètres des méthodes, n'y-a-t-il pas plus subtile ?

Self-Défense

Il y a deux types de code, celui qui ne teste rien et laisse voguer la galère en se disant que les exceptions finiront bien par remonter jusqu'à un message d'erreur, et celui qui s'attache à tout tester en permanence pour éviter les problèmes.

La programmation défensive consiste à anticiper les ennuis. Par exemple vérifier qu'un champ n'est pas null avant de s'en servir, qu'une donnée numérique n'est pas à zéro avant de l'utiliser comme dénominateur dans une division, etc.

Se défendre c'est très bien, anticiper c'est génial, mais à la fin le code est constellé de tests qui en brouillent la compréhension et qui multiplient la maintenance. Pire cela gonfle le code des objets et de leurs méthodes et ralentit systématiquement les traitements là où parfois on préfèrerait plus de vitesse quitte à tester en amont. Par exemple une méthode appelée 100000 fois dans une boucle peut fort bien avoir ses paramètres validés en entrée de boucle si cela est possible, ce qui fera gagner du temps de calcul. Hélas avec la programmation défensive classique le développeur n'a pas ce choix. Si la méthode est "blindée" elle sera 100000 fois appelée avec son blindage même si ponctuellement cela n'était pas nécessaire.

A la place de cette approche classique limitée je vous propose de découvrir les décorateurs de validation.

Blindage de code

Prenons un code typique tel qu'on en écrit tous les jours afin de mieux comprendre le changement de mode de pensée que les décorateurs de validation impliquent.

Par exemple un cas (fictif pour simplifier le code) d'une classe générant un fichier de données sur disque :

```
1. public class Report
2. {
3.     public void Export(string fileName)
4.     {
5.         if (string.IsNullOrWhiteSpace(fileName))
```

```
6.          {
7.              var msg = "Le nom du fichier ne peut être null, vide ou u
   niquement fait d'espaces";
8.              Logger.Log(msg);
9.              throw new Exception(msg);
10.             }
11.
12.             if (File.Exists(fileName))
13.             {
14.                 var msg = "Le fichier existe déjà !";
15.                 Logger.Log(msg);
16.                 throw new Exception(msg);
17.             }
18.
19.             // le code intéressant commence ici...
20.             // ...
21.         }
22.     }
23.
24.     public static class Logger
25.     {
26.         public static void Log(string message)
27.         {
28.             Console.WriteLine($"** Erreur: {message} **");
29.         }
30.     }
```

Voilà un code qui sait se défendre !

Log des erreurs, exceptions avec messages pertinents, tout y est.

Si le nom du fichier est vide, plein d'espaces ou null une erreur spécifique est générée et même loggée ! Mieux si le fichier existe déjà une autre erreur sera déclenchée avec son log aussi.

On pourrait aussi trouver dans un tel code des variantes utilisant *ArgumentNullException* par exemple.

Bref, c'est bien, c'est protégé mais cela possède plusieurs inconvénients évoqués en début d'article mais dont on peut palper ici la réalité :

- C'est lourd ! Avant de lire le code utile de la méthode il faut sauter des tas de lignes qui n'ont rien à voir avec le traitement.
- C'est lent ! Ok il s'agit de l'écriture d'un fichier et les I/O sont lentes par nature, donc c'est peu crucial ici. Mais même ici, si on génère des noms de fichiers sous la forme de GUID assurément uniques dans une boucle d'un million d'appel, quel temps perdu dans tous ces tests qui ne serviront en plus à rien

dans ce cas particulier (le nom de fichier est non nul, non vide, et forcément unique donc le test d'existence assez long n'est pas utile).

- Ce n'est pas modulable. Si dans ma boucle je sais que le nom de fichier sera unique et non vide, je n'ai pas la possibilité d'utiliser une version moins protégée mais plus rapide de la méthode, sauf à en écrire une autre et à commencer du code spaghetti…

Des décorateurs pour valider

C'est là qu'entre en scène les décorateurs de validation.

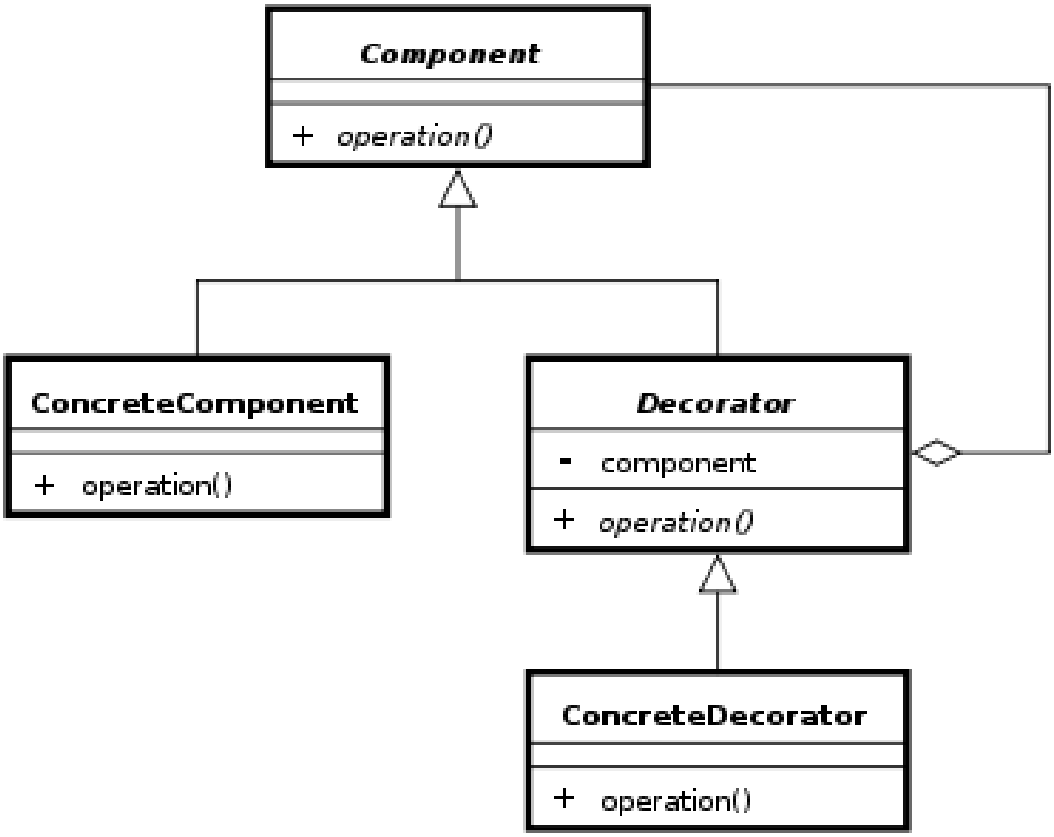
Dans l'un de mes derniers articles je vous parlais des Design Pattern, notamment celles du Gang Of Four, et de l'importance de toutes les connaitre pour savoir les utiliser avec inventivité. Voici aujourd'hui une mise en pratique très parlante.

Les décorateurs ? Dans la classification du GoF on les trouve dans la catégorie des patterns structurels. Pour rappel : *Avec ce pattern on dispose d'un moyen efficace pour accrocher de nouvelles fonctions à un objet dynamiquement lors de l'exécution. Cela ressemble à la fois un peu aux interfaces et à l'héritage mais c'est une troisième voie offrant des avantages différents. Les nouvelles fonctionnalités sont contenues dans des objets, l'objet parent étant dynamiquement mis en relation avec ces derniers pour en utiliser les capacités.*

Connaitre et pratiquer les patterns ouvre l'esprit. C'est ce qui fait d'un développeur moyen un bon développeur, entre autres choses.

Car celui qui a pigé ce qu'est un décorateur, pas seulement en comprenant ma petite définition au coin d'un article mais en pratiquant, testant, en intégrant mentalement toute la portée de ce Design pattern, celui-là trouve ici une idée géniale de les utiliser pour simplifier son code, le rendre plus rapide, plus efficace, plus facilement maintenable.

Peut-être même que le développeur lui-même en sortira plus beau, plus fort et plus séduisant ! (Bon là j'exagère un peu, mais pas tant que ça… écrire du bon et beau code rend sûr de soi, et les grands séducteurs(rices) sont avant tout des gens sûrs d'eux !).



Aller, un peu d'UML ça fait du bien aussi. Ci-dessus on voit le schéma général d'application du pattern Décorateur.

Mais quel est le rapport avec la validation d'un nom de fichier (ou de n'importe quoi d'autre d'ailleurs) ?

La réponse est dans l'abandon des vieux réflexes de la programmation défensive et l'application inventive des DP : les décorateurs de validation.

Comment ?

Ceux qui suivent le mieux ont compris ou commencent à comprendre… Les décorateurs ajoutent dynamiquement au runtime des possibilités nouvelles à des classes. Et on peut décorer un décorateur par un autre décorateur ce qui permet de composer une chaîne selon ses besoins… ça y est vous y êtes ?

Non ? Pas grave, je vais expliquer !

Dans la version classique c'est la méthode Export de notre exemple qui s'occupe de valider systématiquement les arguments qui lui sont passés, ici un string contenant le nom du fichier. Cette validation, dans notre exemple, s'assure que le nom de fichier est non null, non vide, non constitué d'espaces et que le fichier n'existe pas déjà. On pourrait selon le fonctionnel ajouter d'autres tests comme tester la validité du nom (il peut satisfaire toutes les conditions listées mais être non valide pour l'OS). Et notre méthode s'alourdirait encore plus, devant de moins en lisible, de moins en moins maintenable et surtout de plus en lente systématiquement.

Grâce au DP Décorateur nous allons changer la façon de penser la programmation défensive. Nous allons écrire un code pur sans test, rapide, maintenable, puis nous allons le décorer pour assurer les tests. A l'utilisation nous pourrons composer une chaîne de décorateurs ajoutant chacun un test, chaîne dont la lecture indiquera clairement nos intentions. Et si nous ne voulons appliquer qu'un test ou aucun pour gagner en vitesse, cela sera possible.

D'abord partons d'une Interface. Les interfaces sont partout dans la programmation moderne, c'est normal c'est un concept aussi simple qu'il est génial…

```
1. public interface IReport
2. {
3.     void Export(string fileName);
4. }
5. Rien de compliqué.
6. Maintenant construisons une classe qui supporte cette interface :
7.
8. public class DefaultReport : IReport
9. {
10.          public void Export(string fileName)
11.          {
12.              // Code utile uniquement !
13.          }
14.      }
```

Ultra simple. Ce "DefaultReport" sera la version qui fera le travail d'écriture sur disque. Pas de test des paramètres (ici fileName). C'est la version "rapide".

Maintenant construisons une série de décorateurs pour assurer la validation des paramètres :

```
1. public class NoWriteOverReport : IReport
2. {
3.     private IReport origin;
4.
5.     public NoWriteOverReport(IReport report)
6.     {
7.         origin = report;
8.     }
9.
10.          public void Export(string fileName)
11.          {
12.              if (File.Exists(fileName))
13.              {
14.                  var msg = "Le fichier existe déjà !";
15.                  Logger.Log(msg);
16.                  throw new Exception(msg);
17.              }
18.              origin.Export(fileName);
19.          }
20.      }
21.
22.      public class NoNullReport : IReport
23.      {
24.          private IReport origin;
25.
26.          public NoNullReport(IReport report)
27.          {
28.              origin = report;
29.          }
30.
31.          public void Export(string fileName)
32.          {
33.              if (string.IsNullOrWhiteSpace(fileName))
34.              {
35.                  var msg = "Le nom de fichier ne peut être vide, nul
    ou fait d'espaces.";
36.                  Logger.Log(msg);
37.                  throw new Exception(msg);
38.              }
39.              origin.Export(fileName);
40.          }
41.      }
```

Ici nous avons créé deux classes implémentant l'interface IReport. Le principe est celui du décorateur : le constructeur de ces classes prend un IReport en paramètre, celui qui va être décoré. Le décorateur est lui-même un IReport car le décorateur se substitue à la classe décorée de façon transparente.

Un premier décorateur permet de valider le nom du fichier, un second s'occupe de vérifier si le fichier existe ou non.

Chaque classe a une responsabilité clairement définie, elle ne réinvente pas la roue et exploite le savoir faire de la classe décorée en y ajoutant son propre code de portée limitée et facilement maintenable.

Le développeur va ainsi avoir la liberté d'utiliser soit la classe par défaut, rapide et simple mais sans tests, soit les décorateurs, ceux qu'ils veut, quand il le veut, et dans l'ordre qu'il préfère.

Ainsi pourra-t-on écrire :

```
IReport myReport;
    myReport = new DefaultReport();
    myReport.Export("toto.dat");

    myReport = new NoNullReport(new DefaultReport());
    myReport.Export("toto.dat");

    myReport = new NoWriteOverReport(new DefaultReport());
    myReport.Export("toto.dat");

        myReport = new NoNullReport(new NoWriteOverReport(new DefaultReport()));
        myReport.Export("toto.dat");

        myReport = new NoWriteOverReport(new NoNullReport(new DefaultReport()));
        myReport.Export("toto.dat");
```

Dans le premier cas le développeur assume les tests et préfère la vitesse, dans le deuxième et troisième cas un seul test est choisi, dans les deux derniers les deux tests sont composés mais dans un ordre différent.

Au final le code d'exécution est toujours le même – myReport.Export(filename) – ce qui est clair et simple.

De la même façon la composition des opérateurs rend les intentions du développeur tout aussi claires et simples, lisibles.

Conclusion

La maitrise des Design Pattern est l'une des flèches de l'arc du bon développeur. L'inventivité en est une autre.

Grâce à un simple DP datant du GoF que tout le monde croit connaitre on peut inventer une nouvelle façon d'écrire du code défensif plus léger, plus clair, plus lisible et plus efficace pouvant s'adapter, se moduler selon les besoins. Les intentions du développeur sont tout aussi lisible et accessibles du premier coup d'œil.

Le code résultant est aussi bien plus facile à réutiliser. D'abord parce qu'il utilise une interface permettant un découplage fort entre code et contrat. Mais aussi parce qu'il devient très facile d'ajouter des validations non prévues au départ sans jamais modifier le code du DefaultReport qui fait le travail efficace. On minimise les risques de régression au fil de la maintenance qui est rendue plus simple car les classes sont plus petites, mieux ciblées sur une responsabilité unique.

Voici donc un bon exemple de l'importance des DP et du rôle primordial que l'imagination et la créativité jouent dans notre métier, comme dans tout métier d'art.

Un mauvais développeur sera facilement remplacé un jour par un robot. Un bon développeur possèdera toujours ce petit plus qui nous différencie des machines. Et malgré les délires à la mode, l'avènement d'une IA capable de totalement remplacer l'humain dans ce qu'il a de plus complexe, comme la créativité, est vraiment loin d'arriver. Mais il est vrai que ceux qui se contentent d'être des pions interchangeables ont du souci à se faire. Heureusement les lecteurs de Dot.Blog ne sont pas de cette espèce ! Clignement d'œil

Au passage j'aime toujours rendre à César ce qui lui appartient et cette merveilleuse idée de décorateurs de validation me vient d'un article de Yegor Bugayenko à propos de Java… Comme quoi un bon développeur doit aussi savoir garder son esprit ouvert et ne pas se contenter de son petit monde ! Vive la créativité, vive la diversité du monde …

L'intéressant problème du diamant !

Le problème dit du diamant concerne principalement C++ et aussi Java mais pas C# qui résout le problème de façon intelligente mais qui peut surprendre…

Le problème du diamant

Je ne dirais pas que dans une interview d'embauche la question vaudrait de l'or… mieux elle vaudrait du diamant !

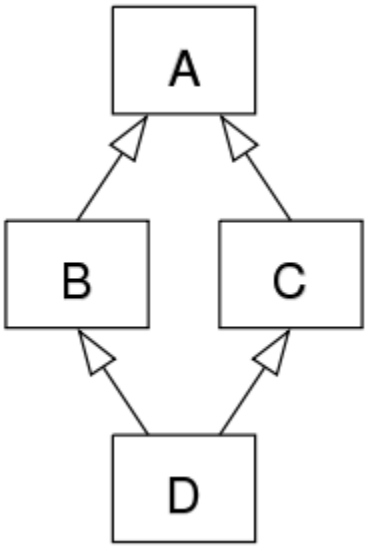
Qu'est-ce que cette histoire de brillants et que vient faire la gemmologie dans C# ?

L'histoire est simple, c'est une banale histoire de famille comme il y en a hélas partout. Une histoire d'argent, enfin de diamant, tout cela pour un héritage.

Le problème du diamant concerne vous l'avez compris l'héritage des classes ou des interfaces, deux entités supportant cette notion de filiation.

Le plus simplement du monde, imaginez une classe appelée Base, puis deux classes filles ClassA et ClassB, chacune proposant une méthode Foo(). Supposons maintenant une nouvelle classe ClassC qui hérite de ClassA et ClassB. L'implémentation de Foo() dans ClassC va poser un petit problème, de quel "Foo()" parlera-t-on ? Celui hérité de ClassA ou de celui provenant de ClassB. Si je créée une instance x de ClassC, écrire "x.Foo();" appellera-t-il la méthode définie dans ClassA ou dans ClassB ?

Ce problème se résume au schéma suivant qui figure… un diamant :



Multi-héritage ? c'est du C++ pas du C# !

Bien entendu ce problème du diamant est lié à l'héritage multiple typique de C++. Bienheureusement et avec sagesse C#, tout comme Java, n'acceptent tout simplement pas le multi-héritage la question semble donc sans fondement vous dîtes-vous …

C'est vrai, mais pas si vite petit bolide !

Les interfaces supportent le multi-héritage sous C# !

Si la question n'a pas de sens sous C# pour les classes, en revanche dans le petit schéma ci-dessus, si nous ne parlons plus de classes mais d'interfaces, tout cela fait sens… car l'héritage multiple existe bien en C# pour les interfaces.

Le problème du diamant existe-t-il alors en C# ?

Oui et non.

Oui car bien entendu le multi-héritage étant autorisée le problème va se poser. Et Non car en réalité C# gère les choses avec une grande logique comme nous allons le voir.

Sachant que Java est de ce point de vue dans la même situation que C# pourquoi ai-je dit en introduction que le problème du diamant ne concernait que C++ et Java ? C'est bête mais dans le cas que nous allons étudier en C# Java donne une erreur de compilation. Ecrivez en Java le code suivant :

```
1.  //Diamond.java
2.  interface Interface1 {
3.      default public void foo() { System.out.println("Interface1's foo"); }
4.  }
5.  interface Interface2 {
6.      default public void foo() { System.out.println("Interface2's foo"); }
7.  }
8.  public class Diamond implements Interface1, Interface2 {
9.      public static void main(String []args) {
10.         new Diamond().foo();
11.     }
12. }
```

… Vous obtiendrez une erreur de compilation "Error:(9, 8) java: class Diamond inherits unrelated defaults for foo() from types Interface1 and Interface2".

Quelle horreur du Java sur Dot.Blog ! Oui oublions vite cette erreur car d'erreur il n'y a pas en C# dans un tel cas !

Java n'a été que le brouillon que C# et cela se voit à ce genre de choses. Mais point de querelle de langage, c'est une simple et évidente constatation, preuve en est avec le problème du diamant.

En C# le problème ne se pose pas, tout simplement.

C# et le diamant

C# interdit l'héritage multiple des classes, ce qui est une bonne chose donc. Il l'autorise en revanche pour les interfaces. Pourquoi donc l'un et pas l'autre de ces deux cas ?

Je ne sais pas pourquoi Java se sent obligé de lever une erreur de compilation dans le cas présenté car la logique permet d'éviter de se poser des questions. En effet, lorsqu'on parle d'héritage de classe cela implique l'héritage du code de la classe. Donc dans le cas du diamant (revenir au petit schéma plus haut) si les classes B et C implémentent une méthode publique Foo() tout le problème est de savoir comment la VMT de la classe D va résoudre le conflit… La VMT (Virtual Method Table, table des méthodes virtuelles) est le mécanisme par lequel la POO gère l'héritage et les méthodes virtuelles. Cette table pointe les méthodes qui doivent être appelées. Pour créer la VMT de la classe D héritant de B et C le compilateur ne saura s'il doit pointer pour Foo() l'implémentation de B ou de C, conflit qui donnera lieu à un arbitrage du développeur.

Mais pour les interfaces le problème n'est pas de même nature. Une interface est une promesse, un contrat. Elle ne contient aucun code. Si dans le schéma plus haut on considère que A, B, C et D sont des interfaces et non des classes, la seule chose que "promet" l'interface D c'est de proposer la liste des opérations (et propriétés) des interfaces B et C. Rien de plus. Donc si j'implémente une classe Z qui supporte l'interface D, et en reprenant le conflit sur Foo() évoqué en Java, Puisque B et C "promettent" d'implémenter Foo(), D aussi, ma classe Z se devra juste d'offrir une méthode Foo(). Aucun conflit ne se pose pour des interfaces. Raison pour laquelle C# fait l'économie d'une erreur. Le contrat est rempli, il n'y a pas de duplication de code ou d'ambigüité particulière.

Pourquoi Java produit-l une erreur de compilation dans le même cas ? Connaissant l'esprit de Java je suppose que cette erreur "inutile" de prime abord est là pour souligner une situation malgré tout embarrassante. Il y a une erreur de conception dans le cas du diamant, c'est une évidence. Java préfère avertir le développeur. Mais un warning aurait été suffisant. C# considère que puisqu'il n'y a aucun problème rien ne sert de bloquer la compilation. C'est une autre vision des choses.

Régler les conflits

On vient de le voir avec les interfaces il n'y a en réalité par d'ambigüité dans le cas du diamant. Mais cela ne veut pas dire qu'il n'y a pas de conflit au moins conceptuel. Comment C# le gère-t-il ?

Comme d'habitude (et lorsque cela est possible) mes exemples de code fonctionnent sous LinqPad qui est bien plus léger pour de tels exercices que VS. Vous pouvez donc copier/coller le programme ci-dessous directement dans cet utilitaire précieux (surtout si vous vous acquittez de la petite licence qui débloque entre autres l'Intellisence ce qui est vraiment bien pratique).

```
1.  void Main()
2.  {
3.      var c = new RealStringChannel();
4.
5.      var b = c as ReadChannel;
6.      var d = c as WriteChannel;
7.
8.      b.Open();
9.      d.Open();
10.     c.Open();
11.
12.     var s = new StrangeChannel();
13.     var sr = s as ReadChannel;
14.     var sw = s as WriteChannel;
15.     s.Open();
16.     sr.Open();
17.     sw.Open();
18. }
19.
20. public interface Channel
21. { int Number { get; set; } }
22.
23. public interface ReadChannel : Channel
24. { string ReadString(); void Open(); }
25.
26. public interface WriteChannel : Channel
27. { void WriteString(); void Open(); }
28.
29. public interface StringChannel : ReadChannel, WriteChannel
30. { Encoder StringEncoder { get; set;} }
31.
32. public class RealStringChannel : StringChannel
33. {
34.     public int Number { get; set;}
35.
36.     public string ReadString() { return ""; }
37.
38.     public void WriteString() { }
39.
40.     void ReadChannel.Open() { Console.WriteLine("Opened for Read."); }
41.
42.     void WriteChannel.Open() { Console.WriteLine("Opened for Write."); }
43.
```

```
44.     public void Open()
45.     {
46.     Console.WriteLine("---- RW Channel");
47.     (this as ReadChannel).Open();
48.     (this as WriteChannel).Open();
49.     Console.WriteLine("----");
50.     }
51.
52.     public Encoder StringEncoder { get; set; }
53. }
54.
55. public class StrangeChannel : ReadChannel, WriteChannel
56. {
57.     public int Number { get; set; }
58.
59.     public string ReadString() { return ""; }
60.
61.     public void WriteString() { }
62.
63.     public void Open()
64.     {
65.         Console.WriteLine("*Open Strange Channel");
66.     }
67.
68. }
```

Ce qui produira la sortie suivante :

```
Opened for Read.
Opened for Write.
---- RW Channel
Opened for Read.
Opened for Write.
----
*Open Strange Channel
*Open Strange Channel
*Open Strange Channel
```

Je vais décomposer un peu car le code exemple fait plusieurs choses et montrent surtout différents aspects du problème du diamant appliqué aux interfaces sous C#.

Où se trouve le diamant ?

Cherche et tu trouveras disent les Ecritures… Enfin c'est mieux si on vous montre au lieu de jouer bêtement au Sphinx ! Et comme je sais où se cache le diamant, voici où le trouver : C'est le groupe de quatre interfaces Channel, ReadChannel, WriteChannel et StringChannel.

J'ai un peu joué au Sphinx quand même car je n'ai pas mis les "I" devant les noms des interfaces laissant croire qu'il peut s'agir de classes… Je suis resté joueur … (en réalité non, j'ai simplement oublié comme une cruche !).

Donc Channel est une interface qui imaginons permet de décrire un canal de communication numéroté. Cette interface n'oblige qu'à une seule chose : offrir une valeur entière Number qui permettra de fixer le numéro du canal utilisé.

ReadChannel hérite de Channel et propose une méthode ReadString() ainsi qu'une méthode Open(). Cette dernière sert à ouvrir le canal de communication pour y lire des strings avec la première (tout cela est fictif).

WriteChannel hérite aussi de Channel et propose aussi une méthode Open() ainsi qu'une méthode WriteString() permettant d'écrire une string dans le canal.

Le diamant se forme avec StringChannel qui hérite à la fois de ReadChannel et WriteChannel pour créer un canal "complet" pouvant lire et écrire des strings avec un numéro de canal (hérité de Channel).

Là où ça pourrait coincer (mais pas en C#) c'est que ReadChannel autant de WriteChannel proposent une méthode Open() !

D'un point de vue conceptuel on pourrait régler la chose en déplaçant Open() dans Channel (et ajouter un Close() d'ailleurs) ce qui semblerait un peu plus "sain". Mais parfois en développement on ne choisit pas les interfaces ni le code dont on .. hérite !

Supposons ainsi que Open() ne peut pas être déplacé. Que va-t-il se passer ?

Le cas simple

Dans le code exemple regardez d'abord la classe StrangeChannel. Elle supporte directement ReadChannel et WriteChannel. L'effet aurait le même si elle avait supporté directement StringChannel (sauf que cette dernière ajoute la propriété Encoder que StrangeChannel ne reprend pas donc).

Cette classe, StrangeChannel, puisqu'elle supporte les deux interfaces Read/WriteChannel supporte aussi Channel, elle offre donc une propriété Number. Puis les méthodes spécifiques ReadString() et WriteString().

Pour Open() elle n'offre bien entendu qu'une seule implémentation.

Channel est respecté, Read et WriteChannel aussi.

Pour preuve le code de Main que je redonne ici pour clarifier :

```
1. var s = new StrangeChannel();
2. var sr = s as ReadChannel;
3. var sw = s as WriteChannel;
4. s.Open();
5. sr.Open();
6. sw.Open();
```

Qui donne la sortie suivante :

```
*Open Strange Channel
*Open Strange Channel
*Open Strange Channel
```

Et oui, il n'existe bien qu'une seule implémentation de Open(), que l'on voit l'instance "s" de StrangeChannel sous sa forme directe (s.Open()) ou bien sous l'aspect de l'une ou l'autre des interfaces qu'elle supporte, c'est toujours le code unique de la méthode publique Open() qui est appelé.

Pas de conflit, pas de problème, pas d'erreur de compilation. Et c'est logique puisqu'avec les interfaces l'héritage de code ne se pose pas. On hérite juste de la "promesse" d'offrir certaines méthodes et propriétés. StrangeChannel répond bien sans tromperie ni ruse aux promesses que cette classe a faites lorsqu'elle a prêté serment de supporter les contrats de ReadChannel et WriteChannel !

Un cas plus tordu

Regardons maintenant le code de la classe RealStringChannel.

Cette classe supporte l'interface StringChannel, le bas du diamant. Elle promet ainsi de supporter à la fois Channel, ReadChannel, WriteChannel et StringChannel.

Pour l'essentiel nous sommes dans le même cas que précédemment. Nous devons juste ajouter la propriété StringEncoder que StringChannel ajoute. Rien de gênant ici.

Mais imaginons que souhaitions proposer un code différent pour Open() selon la forme sous laquelle se présentera l'instance …

C'est ce que montre le code de Main :

```
1. var c = new RealStringChannel();
2.
3. var b = c as ReadChannel;
4. var d = c as WriteChannel;
5.
6. b.Open();
7. d.Open();
8. c.Open();
```

Code qui donne la sortie :

```
Opened for Read.
Opened for Write.
---- RW Channel
Opened for Read.
Opened for Write.
----
```

Le code de Main créée une instance "c" de RealStringChannel. Puis comme pour l'exemple précédent sont ajoutées des variables qui capture cette instance sous la forme plus restreinte de l'une ou l'autre des interfaces supportées (Read et WirteChannel).

L'appel via ReadChannel nous indique "Opened for Read".

L'appel via WriteChannel nous indique "Opened for Write"

Et l'appel via l'instance "complète" de la classe RealStringChannel nous indique le passage "RW Channel" avec une ouverture pour la lecture et autre pour l'écriture.

Diable !

Trois sorties différentes pour l'appel à la même méthode de la même instance !

Par quelle magie cela arrive-t-il ?

C'est dans le code de la classe RealStringChannel qu'il faut aller regarder, allez, je vous le redonne pour éviter d'avoir à scroller :

```
1. public class RealStringChannel : StringChannel
2. {
3.     public int Number { get; set;}
4.
5.     public string ReadString() { return ""; }
6.
7.     public void WriteString() { }
8.
9.     void ReadChannel.Open() { Console.WriteLine("Opened for Read."); }
10.
11.    void WriteChannel.Open() { Console.WriteLine("Opened for Write."); }
12.
13.    public void Open()
14.    {
15.    Console.WriteLine("---- RW Channel");
16.    (this as ReadChannel).Open();
17.    (this as WriteChannel).Open();
18.    Console.WriteLine("----");
19.    }
20.
21.    public Encoder StringEncoder { get; set; }
22.}
```

Regardez comment Open() est définies trois fois… Une fois sans "public" et préfixée de ReadChannel, une autre fois similaire mais préfixée de WriteChannel et une troisième fois "normalement" si je peux dire.

Selon comment l'instance est vue, soit directement comme une instance de RealStringChannel ou comme une implémentation de Read ou WriteChannel, la méthode Open() qui sera fournie à l'appelant sera différente !

Et oui cela est possible en C#.

Conclusion

Possible veut-il dire "souhaitable" ? Certes non.

Je ne vous conseille vraiment pas de créer du code qui fonctionne de cette façon.

Mais comme toute possibilité il est probable qu'il existe des cas où cela peut soit être très utile, soit aider à se sortir d'une mauvaise passe en réutilisant du code qu'on n'a pas écrit à la base.

Le problème du diamant n'existe pas techniquement sous C# mais il existe tout de même au niveau conceptuel avec les interfaces. Il n'y a pas d'incohérences, pas d'erreur de compilation et cela est normal, mais il est possible de créer des situations alambiquées qui peuvent dérouter.

En tout cas si je vous avais demandé au départ de prédire les sorties du code, ou comment avoir trois sorties différentes pour Open() avec la même classe, je ne suis pas certain que tout le monde aurait su quoi répondre.

Ouf! vous avez lu Dot.Blog.

File à priorité (priority queue)

La File à priorité (ou de priorité) n'existe pas dans .NET et pourtant elle peut rendre d'immenses services. Comment l'implémenter et s'en servir ?

Priority Queue

Cette structure de donnée est une "queue" ou plus élégamment en français un "file" (d'attente). C'est aussi une sorte de pile dont les deux représentantes les plus célèbres sont les types LIFO (Last In First Out) et FIFO (First in First Out).

L'enjeu étant de stocker de façon temporaire des objets et de les récupérer ultérieurement, mais pas n'importe comment.

Dans le cas des piles LIFO, c'est la dernière entrée et qui ressort en premier, comme une pile d'assiettes. Le plongeur lave une assiette et la pose sur la pile un fois séchée, le chef prend l'assiette du dessus pour la dresser. Il ne va pas soulever toute la pile pour prendre celle d'en dessous… Le chargeur d'un pistolet automatique (ou de mitraillette).marche de la même façon (la dernière balle insérée sera la première chargée).

La file FIFO offre un fonctionnement assez proche mais comme son nom le laisse supposer c'est l'assiette la plus ancienne posée sur la pile qui ressort en premier. Si cette situation ne convient pas à une pile d'assiettes elle s'adapte en revanche bien mieux à celle d'un file d'attente au cinéma. En dehors d'éventuels resquilleurs ce sont les premiers arrivés qui achèteront leur billet en premier et le dernier client en bout de file qui aura la surprise de savoir s'il reste ou non de la place…

Si je parle de resquilleurs c'est parce que cela existe… même en informatique.

Et si cela est fort mal vu dans une file d'attente nous savons tirer profit de ce principe en programmation. Le resquillage s'appelle alors plus joliment une "priorité".

Par exemple les handicapés peuvent bénéficier d'un accès plus rapide à la caisse du cinéma même s'ils sont arrivés en dernier.

Bref resquiller c'est mal, mais prioriser ça peut être tout à fait légitime et ce sans remettre en cause le fonctionnement global de la file d'attente.

C'est exactement à cela que servent les files à priorité en programmation : ce sont avant tout des files LIFO comme les autres mais les objets stockés (plus généralement pointés que stockés d'ailleurs) peuvent se voir attribuer un niveau de priorité les

faisant avancer plus vite dans la file. La majorité des implémentations fonctionnent un peu à l'inverse, plus le chiffre précisant la priorité est petit plus la priorité est grande (une priorité de 1 donne plus d'avantages dans la file qu'une priorité de de 2).

Beaucoup de problèmes peuvent être résolus avec une file à priorité. On peut supposer par exemple une pile de messages parmi lesquels certains doivent être traités très vite même s'ils ont été empilés plus tard.

.NET ne propose pas de file de ce genre ce qui est assez étonnant puisque les collections de tout genre sont plutôt bien représentées dans le Framework.

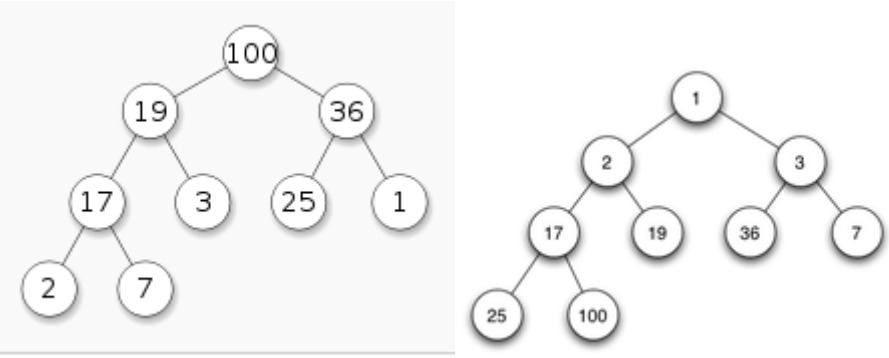
Il faudra donc l'implémenter soi-même.

Les tas binaires

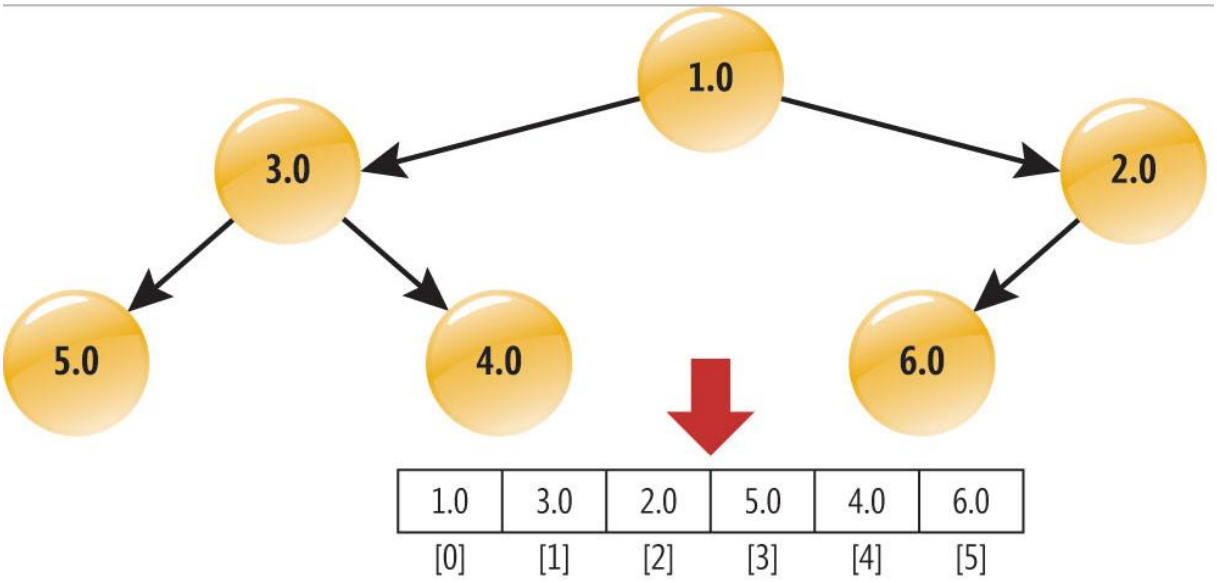
Les tas binaires sont des structures de données parfaitement adaptées à la mise en œuvre des files à priorité (ou parle aussi de "file de priorité") car l'accès à la valeur maximale s'effectue en temps constant.

Gérer une file de priorité pose en effet le problème des multiples insertions et suppressions qui impliquent une réorganisation des données pour gérer les priorités. On imagine bien une sorte de tri mais la mise en œuvre d'un tri à chaque opération serait très long. Les listes chainées peuvent offrir une autre voie mais elles sont délicates à maintenir. Les tas binaires sont plus simples et plus efficaces.

Au départ on peut voir un tas binaire comme un arbre binaire parfait, tous les nœuds de tous les niveaux sont remplis. Et c'est un tas, la valeur de priorité de chaque nœud est inférieure ou égale à celle de chacun de ses nœuds enfants (ou supérieure, tout dépend de l'algorithme retenu, on créé alors un "tas-max" ou un "tas-min").



tas-max et tas-min



L'implémentation que nous choisirons utile un tas-min

Les tas-binaires qui sont des graphes ordonnés sont de précieux alliés du développeur en de nombreuses situations car ils sont très efficaces, rechercher une valeur est proche du principe de la recherche dichotomique sur une liste triée et les temps d'accès sont parmi les meilleurs. Insérer une valeur est en revanche bien moins couteux que de trier une liste pour faire une recherche dichotomique puisqu'on ne fait qu'insérer un élément au bon emplacement ce qui ne concerne qu'un endroit très localisé de la structure de données et non celle-ci dans son ensemble.

Une liste pour support

Lorsqu'on parle d'arbre, de graphe ou de tas binaire on pense donc immédiatement à des pointeurs (ou des références managées ce qui revient un peu au même). Or l'informaticien rusé et soucieux de faire le moins de bogues a comme une résistance naturelle à utiliser ces structures de pointeurs. Elles sont difficiles à déboguer, un peu évanescentes, changeantes…

La paresse et la peur sont plutôt de bonnes conseillères dans notre métier si on sait en tirer parti de façon inventive…

C'est là que les listes font leur entrée.

On ne l'imagine pas tout de suite mais une liste peut parfaitement servir de support à un tas binaire. Cela permet d'exploiter une structure existante et bien testée dans le Framework, cela facilité le débogue (dumper une liste se fait avec un simple foreach), la libération de la mémoire de chaque item dans la liste est déjà pris en compte, bref il n'y a que des avantages.

Reste un détail… Comment transformer une simple liste en un tas binaire ?

L'astuce est algorithmique comme le plus souvent. Pour tout nœud parent qui se trouve à l'index PI (parent index) ses enfants seront stockés à (2 * PI + 1) et (2 * PI + 2). Ce qui permet aussi de savoir que pour tout nœud enfant à l'index CI (child index) le parent se trouve ((CI-1)/2).

La beauté algorithmique de cette solution n'a d'égale que sa simplicité de mise en œuvre … (mais ce qui est beau mathématiquement est souvent simple à formuler donc à coder).

Implémentation de démonstration

Ecrire un code de démonstration est une chose, s'en servir en production en est une autre. Il faudra ajouter des tests, lever des exceptions, s'assurer du bon fonctionnement en multithreading, etc. Mais chacun pourra ajouter ce qui lui semble indispensable.

Je n'ai pas trouvé beaucoup de librairies offrant une liste de priorité, il y a celle-ci (http://bit.ly/HSPQNET) que ne j'ai pas testée, si vous le faites ou si vous en connaissez d'autres n'hésitez pas à laisser un commentaire !

Les items

Pour tester notre code il nous faut des items à gérer dans la file d'attente. Pour les rendre plus facile à gérer ils implémentent IComparable<>.

```
1. public class Item : IComparable<Item>
2. {
3.     public string Name { get; set; }
4.     public int Priority {get; set;} = int.MaxValue; // plus petit = p
   lus grande priorité
5.
6.     public Item(string name, int priority)
7.     {
8.         Name = name;
9.         Priority = priority;
10.          }
11.
12.          public override string ToString()
13.          {
14.              return $"({Name}, {Priority})";
15.          }
16.
17.          public int CompareTo(Item other)
18.          {
```

```
19.             if (Priority < other.Priority) return -1;
20.             else if (Priority > other.Priority) return 1;
21.             else return 0;
22.         }
23.     }
```

Un nom, une priorité dont la valeur est la plus grande possible par défaut (puisque nous gérons ici un tas-min).

La file de priorité

Comme vous allez le voir le code de la file est très simple puisque toute la magie se trouve dans la conceptualisation, l'algorithme et non dans de nombreuses lignes de code complexe :

```
1.  public class PriorityQueue<T> where T : IComparable<T>
2.  {
3.      private List<T> data;
4.
5.      public PriorityQueue()
6.      {
7.          this.data = new List<T>();
8.      }
9.
10.     public int Count => data.Count;
11.
12.     public bool IsConsistent()
13.     {
14.         if (data.Count == 0) return true;
15.         int li = data.Count - 1; // dernier index
16.         for (int pi = 0; pi < data.Count; ++pi) // index parents
17.         {
18.             int lci = 2 * pi + 1; // index enfant gauche
19.             int rci = 2 * pi + 2; // index enfant droit
20.             if (lci <= li && data[pi].CompareTo(data[lci]) > 0) return fals
    e;
21.             if (rci <= li && data[pi].CompareTo(data[rci]) > 0) return fals
    e;
22.         }
23.         return true;
24.     }
25.
26.     public void Enqueue(T item)
27.     {
28.         data.Add(item);
29.         int ci = data.Count - 1;
30.         while (ci > 0)
31.         {
32.             int pi = (ci - 1) / 2;
33.             if (data[ci].CompareTo(data[pi]) >= 0)
34.                 break;
35.             T tmp = data[ci]; data[ci] = data[pi]; data[pi] = tmp;
36.             ci = pi;
37.         }
38.     }
```

```
39.
40.     public T Dequeue()
41.     {
42.         if (data.Count==0) return default(T);
43.         int li = data.Count - 1;
44.         T frontItem = data[0];
45.         data[0] = data[li];
46.         data.RemoveAt(li);
47.
48.         --li;
49.         int pi = 0;
50.         while (true)
51.         {
52.             int ci = pi * 2 + 1;
53.             if (ci > li) break;
54.             int rc = ci + 1;
55.             if (rc <= li && data[rc].CompareTo(data[ci]) < 0)
56.                 ci = rc;
57.             if (data[pi].CompareTo(data[ci]) <= 0) break;
58.             T tmp = data[pi]; data[pi] = data[ci]; data[ci] = tmp;
59.             pi = ci;
60.         }
61.         return frontItem;
62.     }
63. }
```

On pourrait partir de classes existantes dans le Framework ou suivre la façon dont les collections sont implémentées. Ce code n'est qu'une mise en œuvre minimaliste sans aucune dépendance.

Deux opérations sont essentielles : Enqueue et Dequeue. Toutes deux se fondent sur l'algorithme décrit plus haut.

La mise en file d'attente est plus simple à comprendre et reprend à la lettre le petit calcul évoqué dans la présentation de l'algorithme : l'item est d'abord ajouté (ce qui laisse la liste gérer sa taille et son extension si nécessaire, ce qui prépare un emplacement, etc, tout cela simplifie par force notre code). Puis on part de la fin de la liste (donc de l'item ajouté) et on calcule la place théorique de son parent selon sa priorité. Si on ne trouve pas on remontre l'arbre en suivant toujours le même parcours. Dès que l'on trouve un emplacement correct on y insère l'item par permutation.

L'ajout d'un nouvel élément à la file n'est pas instantané mais s'avère être très rapide car les opérations sont peu nombreuses. Le temps augmente en fonction de la taille de la liste mais d'une façon très légère puisque le parcours de la liste s'effectue par moitié à chaque fois (principe d'une recherche dichotomique). Le nombre maximum d'opération est ainsi de O(log n) ce qui est de très loin plus rapide que O(n log n) pour une implémentation plus simple qui n'utiliserait pas l'astuce du tas binaire…

Le code de test

Pour tester l'ensemble on ajoute une méthode à laquelle on passe le nombre d'opérations à réaliser. Elle tirera au sort l'empilage ou le dépilage des éléments. Avec une trace console on peut vérifier que les éléments ayant une plus grande priorité sorte de la file en premiers et que les éléments de même priorité sortent dans leur ordre d'entrée dans la file (ce qui est important pour conserver le sens global qui reste une gestion de file d'attente LIFO).

```
1.  static void TestPriorityQueue(int numOperations)
2.  {
3.      var rand = new Random(0);
4.      var pq = new PriorityQueue<Item>();
5.      for (int op = 0; op < numOperations; ++op)
6.      {
7.          int opType = rand.Next(0, 2);
8.
9.          if (opType == 0) // enqueue
10.         {
11.             string lastName = op + "toto";
12.             int priority = rand.Next(1,1000);
13.             pq.Enqueue(new Item(lastName, priority));
14.             Console.WriteLine($"Mise en file {lastName}, P={priority}");

15.             if (pq.IsConsistent() == false)
16.             {
17.                 Console.WriteLine($"Inconsistance après opération n° {op}")
    ;
18.             }
19.         }
20.         else // Dequeue
21.         {
22.             if (pq.Count > 0)
23.             {
24.                 var e = pq.Dequeue();
25.                 Console.WriteLine($"Dépile {e.Name}, P={e.Priority}");
26.                 if (pq.IsConsistent() == false)
27.                 {
28.                     Console.WriteLine($"Inconsistance après opération n° {o
    p}");
29.                 }
30.             }
31.         }
32.     }
33.     Console.WriteLine("\nTests ok");
34. }
```

Le code de Main est forcément très simple :

```
1.  void Main()
2.  {
3.      const int max = 30;
4.      Console.WriteLine($"Test de {max} opérations");
5.
6.      TestPriorityQueue(max);
7.
```

```
8.      Console.WriteLine("Fin de test");
9.      Console.ReadLine();
10. }
```

Conclusion

Les files de priorité sont des structures très intéressantes qui peuvent régler de nombreux problèmes dans des domaines très variés (gestion de ressources partagées, distribution de messages en réseau, etc). Il est étonnant et dommage que le Framework .NET si riche de collections en tout genre ne propose pas de telles files.

Toutefois on remarque qu'avec un minimum de ruse algorithmique on peut "vampiriser" une List<T> bien testée de .NET pour en faire un tas-binaire ultra efficace.

Je suis certain que vous y penserez maintenant que vous savez que cela existe et qu'il est facile d'en créer avec un peu d'astuce (ou par un copier / coller depuis Dot.Blog !).

Double check lock : frime ou vraie utilité ?

Il y a des modes en programmation, des “il faut” et des “il ne faut pas”, impératifs, sans discussion possible alors qu’ils sortent du néant pour y retourner tôt ou tard… Le Double Check sur un Lock est-il de ce type ?

Le Double Check

Le double contrôle. “C’est bien, la preuve c’est que même Google insiste sur la double authentification”. Je l’ai entendu. Sans rire.

Deux petites choses : d’une part cela n’a rien à voir, ce qui montre à quel point beaucoup d’informaticiens auraient mieux fait d’être charcutiers ou chauffeur-livreurs, de nobles métiers utiles qui demandent malgré tout des capacités intellectuelles moindres. D’autre part cela démontre parfaitement le snobisme qui peut régner dans notre métier où les comportements sont souvent plus ceux de midinettes fashion-victims que d’ingénieurs sérieux et réfléchis.

Bref, le double check c’est de la frime ou bien ? (à dire avec l’accent suisse).

Commençons par le début.

Le double contrôle consiste à contrôler deux fois une même variable, avant le Lock et à l’intérieur de celui-ci au cas où un petit lutin malicieux aurait glissé un thread sournois entre le premier test et l’acquisition du Lock. Dans les faits cela peut arriver, le monde du multithreading est plein de petits lutins malicieux dont il faut déjouer les plans diaboliques.

On se sert le plus souvent d’un double check lors de la création d’un singleton. On peut avoir à s’en servir dans d’autres cas mais celui qu’on rencontre le plus souvent c’est celui du singleton.

Voici un exemple typique de double check :

```
1.  public sealed class UsefulClass
2.  {
3.    private static object _synchBlock = new object();
4.    private static volatile UsefulClass_singletonInstance;
5.
6.    //Pas de création depuis l'extérieur de la classe
7.    private UsefulClass() {}
8.
9.    //Singleton property.
10.   public static UsefulClassSingleton
11.   {
12.     get
13.     {
14.       if(_singletonInstance == null)
```

```
15.       {
16.         lock(_synchBlock)
17.         {
18.           // Si un lutin malicieux glisse un autre thread entre le 1er
19.           // test et l'acquisition du lock.. donc on double check ici :
20.           if(_singletonInstance == null)
21.           {
22.             _singletonInstance = new UsefulClass();
23.           }
24.         }
25.       }
26.     }
27.   }
28. }
```

Dialogue propable :

Lui : *A quoi sert tout ce code ?*

L'autre : *A créer un singleton.*

Lui : *Mais pourquoi tout ce code ?*

L'autre : *Hmm pour permettre une instanciation "lazy" du singleton.*

Lui : *Oui mais pourquoi ?*

L'autre (faisant mine de partir) : *Heuu tu m'énerves avec tes questions là, bon j'ai un truc à finir !*

C'est un peu ça le problème… C'est qu'en réalité cela s'appelle de la micro-optimisation ou de l'optimisation prématurée.

Que cherche-t-on en réalité ? A créer un singleton mais uniquement lors de son premier accès. La raison est que cela évite de créer l'instance si elle n'est pas utilisée.

Ca tient debout à votre avis ?

On pourrait se dire que si le singleton n'est pas utilisé par l'application on voit mal à quoi ça sert de le coder … Mais admettons qu'il existe des cas où cela puisse se justifier.

Du coup pour éviter la création d'une instance (celle du singleton) on créée systématiquement une instance de object pour le Lock… Et on ajoute plein de code pour tester tous les cas de figure les plus vicieux.

Est-ce bien raisonnable ? Peut-être, mais il faut que la création de l'instance du singleton soit sacrément couteuse pour justifier à la fois le code de test et la création permanente d'une instance de object.

C'est là qu'on mesure le ridicule de la situation. Il n'est pas ridicule de faire des singleton, il n'est pas ridicule de vouloir les charger avec un peu de délai, il n'est pas ridicule de double contrôler la variable, non, rien de tout cela est ridicule "par essence". Chacune de ces choses peut se justifier dans des contextes donnés. Ce qui est ridicule c'est que dans 99% des cas cela ne sert à rien.

C'est de *la micro optimisation*, du vent, de la perte de temps.

La version sans contrôle du tout

S'il s'agit de créer un singleton et une fois qu'on a compris que le créer plus tard n'a que rarement du sens il existe un code beaucoup plus plus simple et plus fiable aussi :

```
1. public sealed class UsefulClass
2. {
3.       private static UsefulClass _singletonInstance = new UsefulClass();
4.
5.       //pas de création en dehors de UsefulClass.
6.       private UsefulClass() {}
7.
8.       //Accès au singleton.
9.       public static UsefulClass Singleton =>_singletonInstance;
10.     }
11.  }
12.}
```

Nettement plus court… Et plus fiable aussi car c'est ici .NET qui nous garantit que le code d'initialisation d'une classe statique n'est exécuté qu'une seule fois. Cela est vrai tout le temps même en multithreading. D'ailleurs c'est encore mieux que cela, c'est le CLR plus que .NET qui le garantit.

Il n'y a donc aucune raison de faire des double contrôles alors que le Framework peut gérer la situation encore mieux tout seul !

Et en mode lazy ?

Il se peut, mais c'est assez rare, que cela vaille réellement la peine de différer la création du singleton. Mais même là aussi le Framework nous fournit des outils qui évitent d'avoir à utiliser le double check :

```
1.  public sealed class Singleton
2.  {
3.      /// la Lambda sera exécutée au premier appel
4.      private static readonly Lazy<Singleton> lazy = new Lazy<Singleton>(() =
    > new Singleton());
5.      ///  l'instance en mode lazy
6.      public static Singleton Instance => lazy.Value;
7.
8.      /// pas de création hors de la classe
9.      private Singleton()     {  }
10. }
```

En utilisant la classe *Lazy* avec une expression lambda qui ne sera exécutée qu'au premier appel on s'appuie sur des mécanismes existants et fiables.

Le double check clap de fin ?

Je le disais le double check ne sert pas qu'aux singletons et on peut utiliser ce pattern dans d'autres occasions où cela peut avoir un intérêt.

D'ailleurs revenons un instant sur le double check lui-même. Car il n'est pas inutile, au contraire, au moins de le comprendre !

Ce pattern permet de s'assurer de plusieurs choses :

- Un seul thread peut entrer dans la section critique (grâce au Lock) ;
- Une seule instance est créée et uniquement une et quand cela est nécessaire ;
- La variable est déclarée volatile pour s'assurer que l'assignation de la variable d'instance sera complète avant tout autre accès en lecture ;
- On utilise une variable de verrouillage plutôt que de locker le type lui-même pour éviter les dead-locks ;
- Le double check lui-même de la variable permet de résoudre le problème de la concurrence tout en évitant d'avoir à poser un Lock à chaque lecture de la propriété d'accès à l'instance.

Le premier check évite de placer un lock à chaque lecture de la propriété d'instance ce qui fonctionne correctement puisque la variable est volatile. Le second check s'assure que la variable n'est instanciée qu'une seule fois. Mais pourquoi ? Les histoires de lutins malicieux c'est amusant mais peu technique, alors voici un scénario qui planterait le singleton s'il n'y avait pas de double check :

- Thread 1 acquiert le lock
- Thread 1 commence l'initialisation de l'objet singleton

- Thread 2 entre et se trouve bloqué par le lock, il attend
- Thread 1 termine l'initialisation et sort
- Thread 2 peut enfin entrer dans la section lock et débuter une nouvelle initialisation !

Comme on le voit le pattern lui-même est loin d'être inutile. Il est même crucial dans certains contextes pour éviter à la fois les dead-locks et les doubles entrées dans des sections critiques (lorsqu'elles dépendent d'une variable, c'est pourquoi le cas du singleton est celui où cela est le plus utilisé puisqu'il s'agit déjà d'un pattern visant à contrôler la création d'une instance via une variable ou propriété). On notera que la variable instance doit être *volatile* pour garantir le fonctionnement ici décrit.

Donc pas de clap de fin pour le pattern mais peut-être pour le singleton…

En effet, le Singleton c'est l'antithèse de la POO. Avant la POO on avait un fichier avec tout un tas de fonctions dedans. Avec le Singleton on a une classe avec tout un tas de méthodes dedans. C'est un peu la POO pour débutant. Pas besoin de savoir combien d'instances il faudra gérer, il n'y en aura qu'une, tout redevient linéaire et simple à comprendre, comme à l'ancien temps quoi. C'était mieux avant. Un grand classique !

Certes le Singleton est un pattern décrit par le Gang Of Four, donc c'est forcément bien… Pas sûr. Les exemples donnés pour justifier le Singleton sont de type driver d'impression. Or ce n'est pas tout à fait le type de programme qu'on écrit tous les jours… Les Singleton créée des zones d'étranglement dans un environnement multithread, ce qu'impose toute programmation moderne en raison de l'évolution des processeurs. Du Temps du GOF les processeurs étaient mono-cœurs et l'évolution se faisait par la monté en Ghz. Depuis au moins 20 ans la progression ne se fait plus de cette façon mais par la généralisation des multi-cœurs. Un bon développement utilise forcément de nos jours du multithreading. Dans ce contexte très différent de l'époque du GOF le Singleton devient un problème plus qu'une solution.

En général ils servent plus à rendre des services qu'à gérer une device (exemple du driver donné par le GOF). Toutefois pour rendre des services nous avons des constructions mieux adaptées qui n'imposent pas d'avoir de singletons et qui utilisent l'injection de dépendances principalement, la messagerie MVVM ou un service locator. On notera que ce dernier peut être un singleton car il ne joue qu'un rôle d'aiguillage et ne constitue pas une zone de blocage, en revanche le code qui est derrière se doit d'être thread safe.

Donc oui au double-check, bien entendu, là où cela est indispensable, mais pour le singleton… avant d'en créer … double-checkez votre réflexion avant de le faire !

Conclusion

Le double-ckeck locking possède de solides justifications techniques dans un environnement multitreadé. Il ne s'agit ni d'une mode ni de frime, c'est utile.

Mais c'est utile uniquement dans certains cas très limités. Notamment pour la création des singletons.

Or il existe d'autres façons de créer un singleton encore plus fiable et plus courtes…

Et il subsiste une question : "les singletons programmation à papa ou vraie utilité ?"

A vous d'y réfléchir, à deux fois !

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 et en avril 2017 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 2017, dix 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.

Malgré une révision systématique des textes 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.

L'auteur

Œuvrant sous le nom de société E-Naxos, Olivier Dahan est un conférencier et auteur connu dans le monde Microsoft. La société fondée en 2001 se vouait au départ à l'édition de logiciels. Héritière en cela de OBS (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 la suite Hippocrate (gestion de cabinet médical et de cabinet de radiologie) puis d'autres produits comme MK Query Builder (requêteur visuel SQL pour VB et C#).

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 et aujourd'hui pour la *9ème année* au sein de la communauté *Visual Studio & Development Technologies*. 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 Olivier continue à proposer ses services de Conseil, Audit, Formation et Développement, toutes ces activités étant centrées autour des outils et langages Microsoft, de WPF et UWP au cross-plateforme avec Xamarin sous Android, iOS, Mac OS et Windows.

N'hésitez pas à faire appel à Olivier, la compétence et l'expérience sont des denrées rares… Pourtant un expert indépendant ne coûte pas plus cher qu'un second couteau d'une grande SSII, pensez-y !