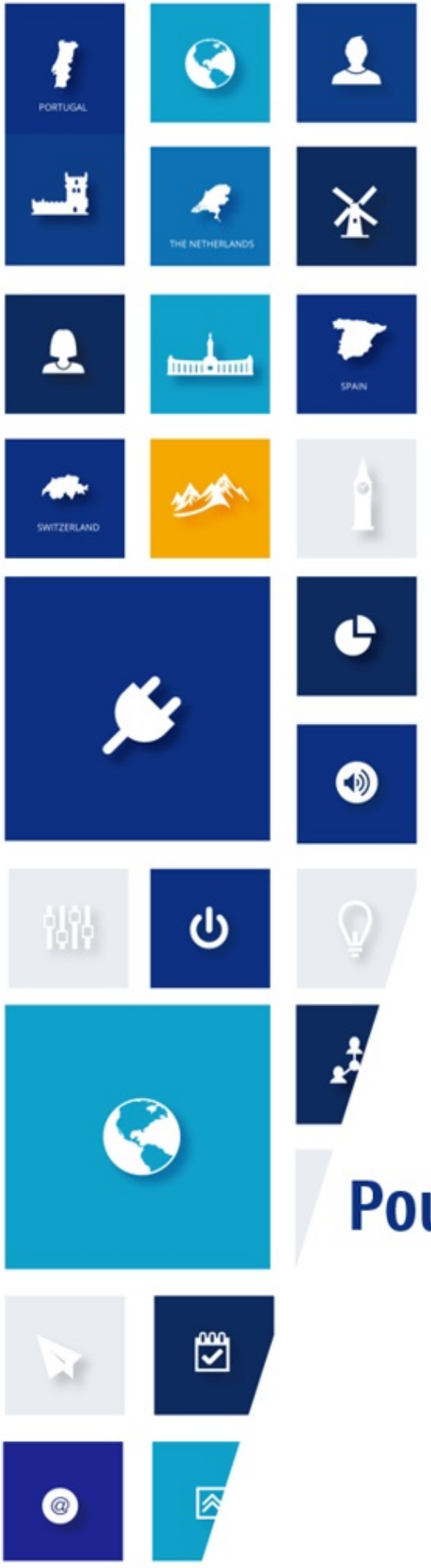


AAA Programming

Pour les développeurs .Net

Table des matières

Pourquoi ce livre

Pendant plusieurs années j'ai été éditeur de logiciels pour les industries graphiques et plus particulièrement pour les éditeurs de magazines et les imprimeurs. J'ai développé en .Net une solution logicielle pour automatiser l'impression depuis l'éditeur de magazine jusqu'à l'imprimeur. Cette solution logicielle ne devait en aucun cas être à l'origine de l'arrêt de l'outil industriel de l'éditeur ou de l'imprimeur. Sachant que l'outil industriel d'un imprimeur se chiffre en moyenne à quelques dizaines de millions d'euros, la solution devait fonctionner 7j/7, 24h/24 sans aucun support compte tenu du fait que l'impression d'un magazine ou d'un quotidien se fait le plus souvent en dehors des heures ouvrées traditionnelles.

Pour satisfaire ce très haut niveau d'exigence, j'ai dû changer ma façon de coder.

Au fil des années j'ai mis au point un ensemble de techniques de programmation permettant de livrer rapidement une application sans bug et ne nécessitant aucun support une fois mise en production.

Puis en tant que consultant, j'ai partagé ces techniques avec d'autres développeurs lors des missions que j'ai menées à bien. A chaque fois j'ai été étonné de l'impact positif lié à l'application de ces méthodes:

- Amélioration de la lisibilité du code;
- Convergence plus rapide vers le zéro bug;
- Augmentation de la vélocité de l'équipe;
- Accroissement de la qualité du produit livré.

Partager ces techniques avec d'autres développeurs m'a aidé à les formaliser puis m’a incité à les présenter dans cet ouvrage.

A qui est destiné ce livre

Ce livre est destiné à un double public:

- A tous les développeurs .Net, du développeur débutant au développeur confirmé, qui ont l'ambition de développer des applications critiques ou grand public, qui ont l'ambition de fournir un code simple à comprendre, facile à lire, facile à maintenir, facile à faire évoluer;
- A tous les responsables qui ont les objectifs suivants pour leur équipe :
    - Augmenter la vélocité de l'équipe;

- Faire en sorte qu'un développeur puisse enrichir, modifier, maintenir le code d'un autre sans qu'on puisse distinguer qui à écrit quoi;
- Faire en sorte qu'une équipe de N développeurs agisse comme un seul développeur à la puissance N;
- Maintenir la maintenabilité;
- Répondre dans les plus brefs délais aux évolutions du métier ou du marché.

A propos des exemples de code montrés dans ce livre

Ce livre contient des exemples de code qui sont tous tirés d'applications réelles. Toutefois, ces exemples ont été retravaillés de façon à apparaître comme des "codes snippets" sans lien avec l'application et le développeur d'origine. Votre feedback est très important : je suis toujours à la recherche d'exemples de code réel à partir desquels je peux montrer comment appliquer les techniques décrites dans ce livre.

Pré-requis

Tous les exemples de code sont écrits en .Net C#.

Si vous souhaitez expérimenter vous mêmes les techniques montrées dans ce livre, je vous invite à installer Visual Studio 2015 Community Edition sur votre poste.

Work in Progress

Ce livre est en cours d'écriture. J'ai besoin de votre feedback pour l'améliorer : n'hésitez pas à commenter.

Code Companion

Vous pouvez voir en action les techniques du aaaProgramming en allant sur le projet GitHub associé à cet ouvrage. Un package NuGet est également disponible pour exploiter ces techniques directement dans Visual Studio.

Nommer correctement les choses

Nommer correctement les choses signifie donner un nom suffisamment évocateur à la variable que vous vous apprêtez à déclarer, à la méthode, à la classe, à la propriété, à l'interface, bref à tous les objets que vous allez manipuler au sein de votre application.

Cette activité de nommage permet de rendre son code expressif. Un code expressif raconte une histoire.

L'intérêt de raconter une histoire est qu'une relecture rapide permet de détecter très tôt les éventuelles incohérences, les éventuelles erreurs de conception , permet de se poser très rapidement les bonnes questions et permet d'échanger plus facilement avec les autres développeurs de son équipe.

L'objectif de ce chapitre est de vous montrer les méthodes qui vous permettront de trouver le nom le plus évocateur possible pour définir une variable, une méthode, une propriété ou une classe.

Trouvez le bon nom peut se révéler être une activité difficile ou chronophage.

Chaque fois que j'ai rencontré une réelle difficulté à trouver le bon nom, je me suis rendu compte que pour résoudre ce problème, il fallait modifier la conception ou les choix effectués pour développer l'application: dans tous les cas de figure il est apparu que la nouvelle conception ou les nouveaux choix étaient meilleurs que ceux initialement prévus.

Autrement dit nommer correctement les choses a une vertu inédite : détecter très rapidement si un choix technique ou un choix de conception est pertinent.

Si vous êtes responsable d'équipe, je vous recommande d'être à l'écoute des difficultés rencontrées par un développeur quand il doit donner un nom expressif aux différents éléments de son code : c'est un signal très fort de la présence d'un choix de conception ou d'un choix technique qui doit être retravaillé avec l'équipe.

Pour nommer correctement les choses, il est nécessaires de connaître et de respecter dans un premier temps les conventions de nom qui existent au sein du Framework .Net.

Les conventions de nom du Framework .Net

L'ensemble du Framework .Net a été écrit en suivant les techniques décrites dans le livre :

Framework Design Guidelines : Conventions, Idioms and Patterns for Reusable .NET libraries

Une partie de cet ouvrage est également disponible en ligne sur MSDN.

Je recommande fortement la lecture de ce livre en préambule à l'application des méthodes décrites plus loin dans cet ouvrage.

Chez Microsoft, tous les développeurs suivent les conventions décrites dans ce *Framework Design Guidelines*.

En tant que développeur .Net, vous allez utiliser bon nombre d'outils mis à votre disposition dans le Framework .Net.

Si vous utilisez les conventions décrites dans l'ouvrage *Framework Design Guidelines*, votre propre code semblera être une extension naturelle du Framework .Net. Vous allez pouvoir écrire votre application sous la forme d'une histoire en vous appuyant sur les mots et les verbes du Framework .Net.

La première règle à suivre concerne les conventions de nommage.

Nommer correctement les choses passe par la connaissance et la mise en pratique des conventions de nom existantes dans le Framework .Net.

Ces conventions de nom sont au nombre de deux:

- Pascal Casing;
- Camel Casing.

Pascal Casing

Au début du développement du Framework .Net, vers 1996, les développeurs tombèrent d'accord sur deux conventions de nom pour nommer les propriétés, les méthodes, les variables et les classes.

Au départ, ces deux conventions de nommage étaient ... sans nom.

La première convention de nom était utilisée pour nommer les méthodes et les propriétés. Aujourd'hui, le nom d'une méthode ou d'une propriété suit toujours le même pattern au sein du Framework .Net:

- Le nom est une concaténation de mots;
- La première lettre de chaque mot est en majuscule (par exemple ToString).

En 1996, Anders Hejlsberg rejoint Microsoft. Il est le père fondateur du Framework .Net et du langage CSharp. C'est également le père fondateur de Typescript.

Avant de rejoindre Microsoft, Anders Hejlsberg a développé chez Borland, le compilateur Turbo C et Turbo Pascal.

Dans turbo Pascal, les méthodes et les propriétés suivaient la convention de nom suivante:

- Le nom est une concaténation de mots;
- La première lettre de chaque mot est en majuscule.

L'équipe de développement du Framework .Net a choisi de nommer cette convention *Pascal Casing*, en hommage à Anders Hejlsberg et en référence à Turbo Pascal.

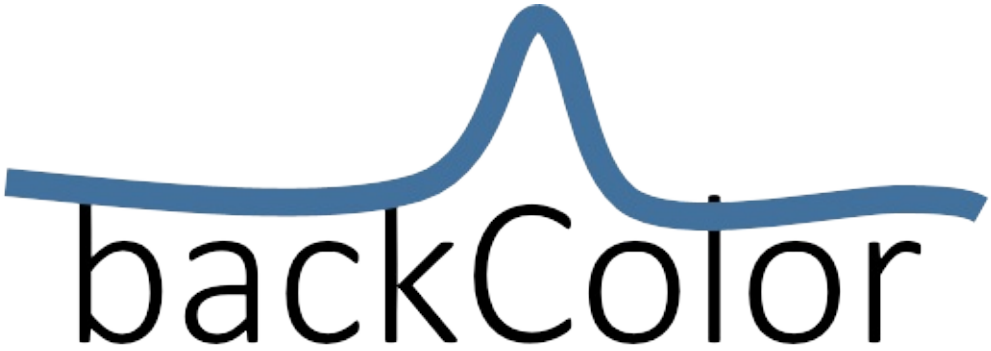
Camel Casing

La deuxième convention de nom était utilisé, en 1996, pour donner un nom aux variables locales :

- Le nom est une concaténation de mots;
- La première lettre de chaque mot est en majuscule sauf pour le premier mot (par exemple backColor).

L'équipe de développement du Framework .Net a eu beaucoup de mal pour trouver un nom à cette convention qui soit en même temps une référence à un homme exceptionnel ou à une technique connue de tout le monde.

Par contre si vous dessinez une ligne au dessus d'un nom écrit avec cette convention, la forme obtenue fait penser à la bosse d'un chameau.



L'usage des conventions de nom Camel Casing et Pascal Casing est spécifique au Framework .Net. Dans le langage JavaScript, même si vous retrouvez ces deux conventions de nom, leurs conditions d'application sont différentes de celle du Framework .Net.

Si vous développez votre application en .Net, vous devez savoir quand appliquer l'une ou l'autre de ces conventions. C'est l'objet du paragraphe suivant.

Quand utiliser Pascal Casing

La convention de nom Pascal Casing doit être utilisée pour nommer les éléments suivants:

- Classe;
- Énumération;
- Valeur d'une énumération;
- Événement;
- Membre public static et read-only;
- Méthode;
- Propriété;
- Namespace;
- Interface.

> L'interface est un cas particulier dans la mesure où, par convention, le nom doit être préfixé par la lettre **I** comme c'est le cas pour les interfaces disponibles dans le Framework .Net comme par exemple:
>
> - IEnumerable;
> - IQueryable;
> - IDisposable.

Je vous propose, ci-dessous, d'étudier quelques exemples d'application de la convention de nom Pascal Casing. L'objectif de ces exemples est de vous amenez à découvrir comment déterminer si un nom est correct.

Exemple 1

```
public class Class1
{
}
```

Par défaut Visual Studio applique automatiquement la convention de nom Pascal Casing quand vous ajoutez un nouveau fichier de classe dans votre projet. *Class1* est conforme à la convention Pascal Casing : il s'agit d'un nom constitué d'un seul mot et la première lettre de ce mot est en majuscule. Il est donc parfaitement valide de conserver ce nom, mais en général vous, comme moi, changez *Class1* en un nom plus explicite. Conserver le nom par défaut donné par Visual Studio aurait pour résultat d'obfusquer votre code.

> A mes yeux, le nom *Class1*, bien que conforme à la convention de nom Pascal Casing, ne véhicule rien de spécifique quant à l'usage ou l'objectif de cette classe. *Class1* est un nom qui manque d'expressivité, il contribue à rendre plus difficile l'usage et la compréhension de cette classe pour un développeur externe qui serait amené à co-développer votre projet.

Exemple 2

```
public class BasePage : System.Web.UI.Page
{
  ...
}
```

La classe *BasePage* a été définie dans un projet Web ASP.NET WebForm.

Ce nom est conforme à la convention de nom Pascal Casing. Il est aussi suffisamment évocateur pour qu'on comprenne que la classe *BasePage* sera utilisée comme classe de base pour toutes les nouvelles pages qui seront définies dans le projet Web.

Cependant, ce nom pose problème pour deux raisons :

- la classe *System.Web.UI.Page* est elle même la classe de base par défaut pour toutes les pages Web d'un projet Web ASP.NET Web Form. Le nom choisi, *BasePage*, semble plus générique que *Page*, ce qui est contradictoire avec la notion d'héritage.

  En effet, l'héritage permet de spécialiser de plus en plus une classe, par conséquent d'un point de vue sémantique on s'attend plutôt à avoir ce type de déclaration:

  ```
  public class Page : BasePage
  {
  ...
  }
  ```

  Voici quelques exemples de classes du Framework .Net qui illustrent la relation qui existe entre une classe nommée *Xyz* et la classe nommée *XyzBase*:

  ```
  public class Attachment : AttachmentBase
  {
  ...
  }
  ```

```
public class ColorAnimation : ColorAnimationBase
{
...
}
```

```
public class Setter : SetterBase, ISupportInitialize
{
...
}
```

```
public class Button : ButtonBase
{
...
}
```

- Le nom *BasePage* diffère de la convention communément établie au sein du Framework .Net, à savoir que le mot *Base* doit être le dernier mot à être concaténé, comme l'illustre les quelques classes ci-dessous du Framework .Net:
  - System.Collections.DictionaryBase
  - System.Configuration.Settingsbase
  - System.Web.HttpContextBase
  - System.Web.HttpResponseBase
  - System.Windows.Controls.Primitives.ButtonBase

> A mes yeux, le nom *BasePage* est un nom de classe suffisamment évocateur. Cependant ce nom aurait pu être parfaitement correct si :
>
> - Sa sémantique avait été conforme à son usage (car si vous demandez à un développeur du Framework .Net de classer *BasePage* par rapport à Page, il vous dira tout de suite: *Page* hérite de *BasePage*);
> - La façon de concaténer les mots avait été conforme à une concaténation similaire au sein du Framework .Net.
>
> Pour que le nom soit correct il faudrait le remplacer par un nom comme *MyAppPageBase, ou plus simplement par un nom comme MyAppPage,* où *MyApp* serait le nom interne de votre application.

Autrement dit quand vous donnez un nom à une classe, une propriété ou une méthode, cherchez à savoir si il existe des noms similaires dans le Framework .Net et comparez toujours avec ce qui existe dans le Framework .Net pour déterminer la pertinence de votre choix.

IlSpy est un outil qui peut vous aider dans cette démarche d'analyse.

Quand vous estimez que le nom choisi rempli tous les critères à savoir:

- Le nom est expressif
- Le nom est conforme à ce qui existe déjà dans le Framework .Net

Validez toujours votre choix de la façon suivante:

- Obtenez le consensus au sein de l'équipe;
- Appliquer la méthode TDD (Test Development Driven) à votre nouvelle classe, méthode ou propriété pour vérifier que le code est facile à écrire, facile à comprendre, facile à modifier avec le nom que vous avez choisi.

Si vous rencontrez la moindre difficulté pour obtenir le consensus ou pour écrire simplement des test unitaires, je vous recommande fortement de changer de nom.

Si a la troisième itération, le nom choisi cause toujours des difficultés, il convient de s'arrêter et de réfléchir, avec l'ensemble de l'équipe, à la pertinence du ou des choix techniques qui ont amené à la création de la classe, de la méthode ou de la propriété en question.

Quand utiliser Camel Casing

La convention de nom Camel Casing doit être utilisée pour nommer les éléments suivants:

- Variable locale;
- Paramètre de méthode;

A compléter.

Cas particulier des variables qui sont associées à une propriété

Les membres privés d'une classe sont par définition des variables locales à la classe où ils sont définis.

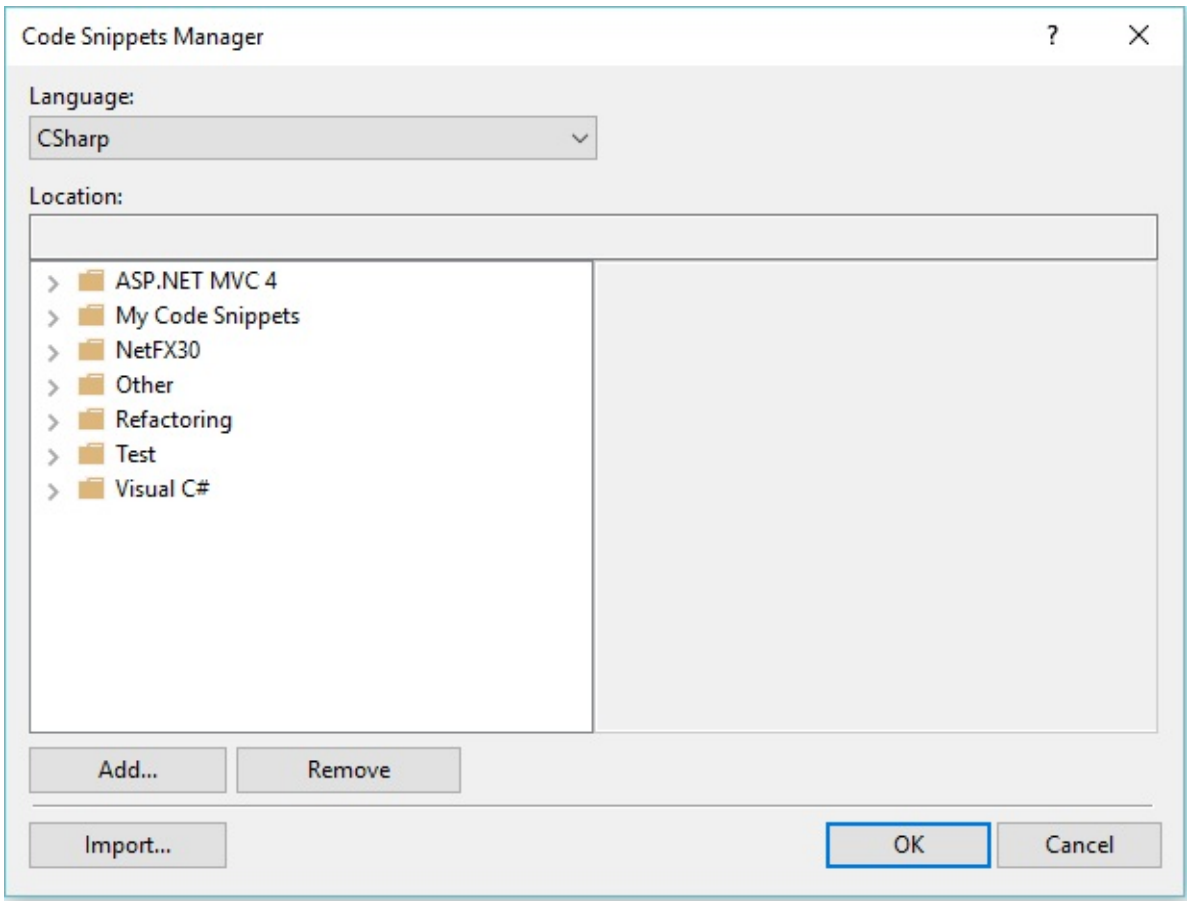
Le nom d'un membre privé doit par conséquent être conforme à la convention de nom Camel Casing.

Quand vous insérez une propriété dans une classe en utilisant le code snippet nommé *propfull*, vous obtenez le résultat suivant:

```
private int myVar;
public int MyProperty
{
    get { return myVar; }
    set { myVar = value; }
}
```

Un code snippet est un mot clé qui permet d'insérer un bloc de code prédéfini. Pour mettre en action un code snippet, il suffit de cliquer à l'endroit où on souhaite ajouter le bloc de code prédéfini, de taper ensuite le mot clé associé au code snippet, comme par exemple *propfull*, puis de cliquer deux fois sur la touche TAB.

L'objectif du code snippet est de favoriser l'insertion rapide de code. Il s'agit donc d'un outil de productivité. Il existe plein de code snippets disponibles pour le language CSharp. Pour découvrir les codes snippets disponibles dans Visual Studio, allez dans le menu *Tools* de Visual Studio puis sélectionnez l'option *Code Snippets Manager...* :



Naviguez dans les dossiers proposés et découvrez les code snippets fournis par Microsoft.

La propriété fournie par le code snippet *propfull* déclare une variable locale à la classe et une propriété dont le getter et le setter s'appuient sur cette variable locale.

Le nom de la variable locale, *myVar*, est conforme à la convention de nom Camel Casing. Le nom de la propriété, *MyProperty*, est conforme à la convention de nom Pascal Casing.

Le nom myVar pose cependant un problème. En effet le nom myVar est très éloigné sémantiquement du nom MyProperty laissant ainsi penser que l'un peut être modifié indépendamment de l'autre. Un autre développeur, qui serait amené à modifier ou à faire évoluer votre code, pourrait parfaitement modifier la variable myVar au lieu de la propriété MyProperty (et vice versa) entraînant ainsi un comportement erroné de l'objet modifié. Cela peut avoir pour conséquence un dysfonctionnement de l'interface graphique de l'application notamment quand les objets doivent être observables comme par exemple dans une application WPF.

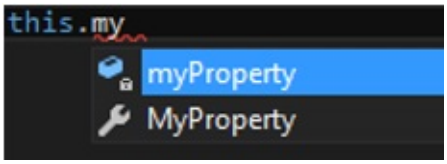
Pour résoudre ce problème une première approche consiste à réécrire la propriété ci-dessus de la façon suivante:

```
private int myProperty;
public int MyProperty
{
    get { return myProperty; }
    set { myProperty = value; }
}
```

Le nom de la variable et le nom de la propriété sont identiques d'un point de vues sémantique puisqu'ils ne diffèrent que la par la casse de la première lettre.

Cependant cela accroît le risque d'utiliser dans d'autre partie du code la variable au lieu de la propriété (et vice versa).

En effet l'IntelliSense propose maintenant les choix suivants quand vous commencez à taper le nom de la propriété:



L'intelliSense étant un outil de productivité, il est vraisemblable que vous choisissiez, sans même vous en apercevoir, la variable locale en lieu et place de la propriété.

C'est ce que j'appelle le bug de productivité : vous introduisez un bug dans votre application parce que vous avez choisi trop rapidement l'option proposée par l'IntelliSense.

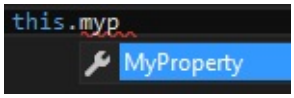
Comment résoudre ce problème posé par l'usage intensif de l'IntelliSense?

Il suffit de réécrire la propriété ci-dessus de la façon suivante:

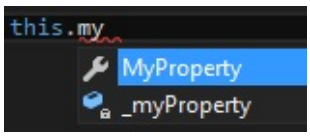
```
private int _myProperty;
public int MyProperty
{
    get { return _myProperty; }
    set { _myProperty = value; }
}
```

En ajoutant simplement le caractère _ devant le nom de la variable locale, l'IntelliSense apporte les avantages suivants:

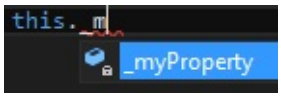
- Quand vous commencez à taper les trois premières lettres de la propriété, l'IntelliSense masque l'accès à la variable locale:

- Quand vous commencer à taper seulement les deux premières lettres de la propriété, l'IntelliSense vous positionne par défaut sur la propriété et non sur la variable locale:



- Quand vous commencez à taper le caractère _, l'Intellisense vous donne accès uniquement aux variables locales qui sont associées à des propriétés:

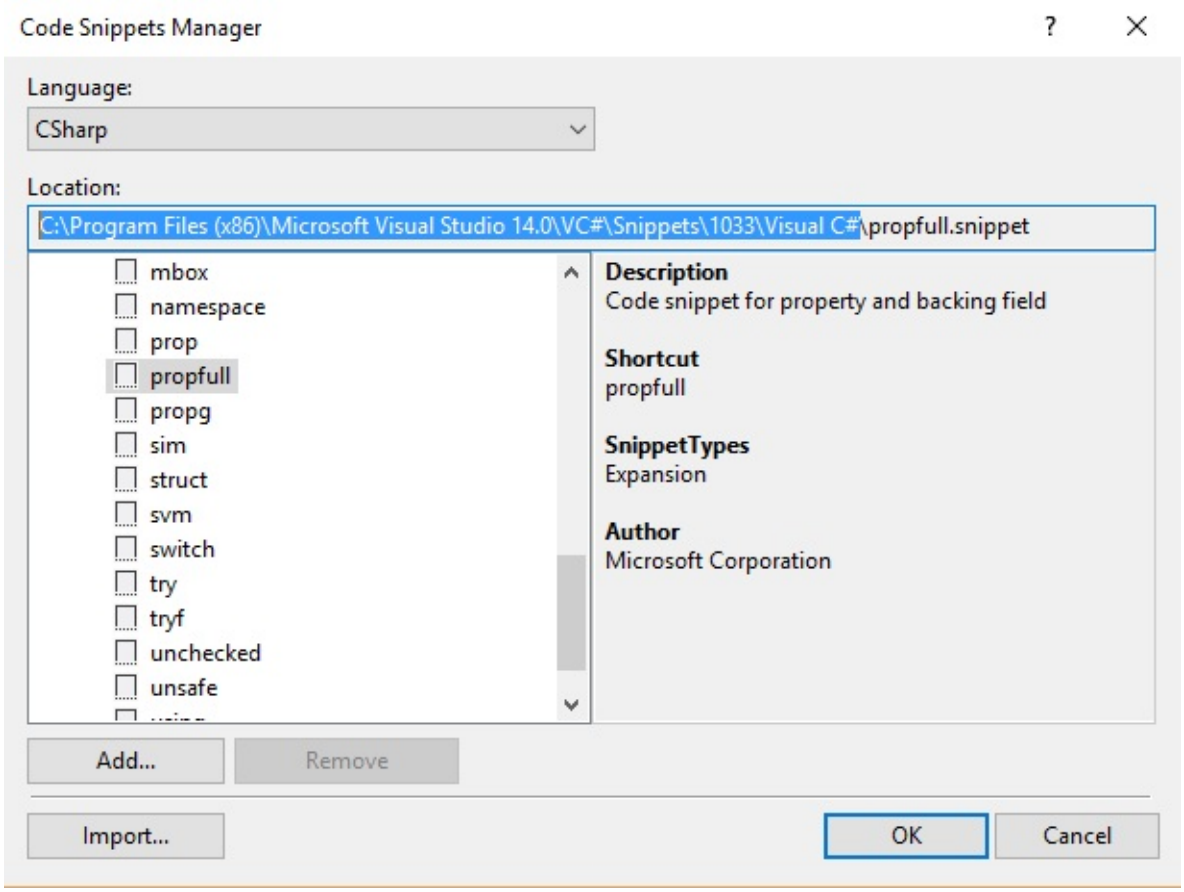


Pour être certain que l'IntelliSense fasse correctement son travail et vous évite de sélectionner la variable au lieu de la propriété, celle ci peut être réécrite de la façon suivante:

```
private int _myProperty;
public int MyProperty
{
    get
    {
        return this._myProperty;
    }
    set
    {
        this._myProperty = value;
    }
}
```

Pour vous aider à insérer automatiquement une nouvelle propriété en suivant la syntaxe proposée ci-dessus, vous pouvez substituer le code snippet *propfull* par votre propre version.

Pour cela, ouvrez la fenêtre "Code Snippets Manager", ouvrez le dossier Visual CSharp puis cliquez sur *propfull*. Le champs *Location* vous montre l'emplacement du fichier associé au code snippet *propfull*:



Allez dans le dossier indiqué:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC#\Snippets\1033\Visual C#
```

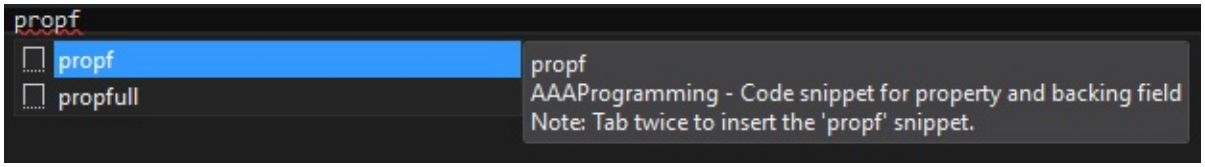
et copiez le fichier propfull.snippet dans le dossier :

```
Documents\Visual Studio 2015\Code Snippets\Visual C#\My Code Snippets.
```

Ouvrez le fichier ainsi copié dans Visual Studio et remplacez son contenu par:

```
<?xml version="1.0" encoding="utf-8" ?>
<CodeSnippets  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
    <CodeSnippet Format="1.0.0">
        <Header>
            <Title>propf</Title>
            <Shortcut>propf</Shortcut>
            <Description>AAAProgramming - Code snippet for property and backing field</
Description>
            <Author>My Company</Author>
            <SnippetTypes>
                <SnippetType>Expansion</SnippetType>
            </SnippetTypes>
        </Header>
        <Snippet>
            <Declarations>
                <Literal>
                    <ID>type</ID>
                    <ToolTip>Property type</ToolTip>
                    <Default>int</Default>
                </Literal>
                <Literal>
                    <ID>property</ID>
                    <ToolTip>Property name</ToolTip>
                    <Default>MyProperty</Default>
                </Literal>
                <Literal>
                    <ID>field</ID>
                    <ToolTip>The variable backing this property</ToolTip>
                    <Default>_myProperty</Default>
                </Literal>
            </Declarations>
            <Code Language="csharp"><![CDATA[
      private $type$ $field$;
    public $type$ $property$
    {
        get
    {
      return this.$field$;
    }
        set
    {
      this.$field$ = value;
    }
    }
    $end$]]>
            </Code>
        </Snippet>
    </CodeSnippet>
</CodeSnippets>
```

Enregistrez le fichier *propfull.snippet*. Retournez dans le code source de votre classe et tapez *propf*:



Si vous appuyez immédiatement deux fois sur la touche Tab, vous insérer dynamiquement la propriété:

```
private int _myProperty;
public int MyProperty
{
    get
    {
        return this._myProperty;
    }
    set
    {
        this._myProperty = value;
    }
}
```

En modifiant la section Code du code snippet *propf* de la façon suivante:

```
<Code Language="csharp"><![CDATA[
      #region $property$

  private $type$ $field$;
  /// <summary>
  ///
  /// </summary>
    public $type$ $property$
    {
        get
    {
      return this.$field$;
    }
        set
    {
      this.$field$ = value;
    }
    }

  #endregion
    $end$]]>
            </Code>
```

Vous obtiendrez le résultat suivant chaque fois que vous utiliserez le code snippet *propf*:

```
#region MyProperty

private int _myProperty;
/// <summary>
///
/// </summary>
public int MyProperty
{
    get
    {
        return this._myProperty;
    }
    set
    {
        this._myProperty = value;
    }
}

#endregion
```

En résumé, une variable locale qui est associée à une propriété doit répondre aux règles suivantes:

> Le nom d'une variable locale associée à une propriété est constitué d'un préfixe, le caractère _, suivi d'un nom conforme à la convention Camel Casing.
>
> La variable doit être déclarée au plus près de la définition de la propriété.
>
> La variable doit être utilisée exclusivement au sein du getter ou du setter de la propriété.
>
> > Cette règle est fondamentale: l'état d'un objet doit être consulté ou modifié exclusivement au travers de la propriété associée, y compris dans le code que vous écrivez au sein de la classe où est définie cette propriété. J'aime appeler cette règle : protect yourself from yourself.
> >
> > Déroger à cette règle, c'est prendre le risque de modifier un objet métier d'une façon inattendue, ou pire, de provoquer une modification erronée des données métier au moment de leur persistance par la couche d'accès aux données.

Références

Cette section contient la liste des références utilisées pour la rédaction de cette section.

Naming Guidelines
• https://msdn.microsoft.com/en-us/library/ms229002(v=vs.110).aspx • Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries de Krzysztof Cwalina et autres

Anders Hejlsberg
• https://en.wikipedia.org/wiki/Anders_Hejlsberg • Anders Hejlsberg - What influenced the development of C#? (Channel 9) • Behind the code (Channel 9)

A compléter.

Comment nommer une méthode ou une propriété qui renvoie un booléen

Une méthode ou une propriété qui renvoie un booléen doit être nommée en utilisant l'une des quatre options ci-dessous:

- le préfixe :
  - *Is*;
  - *Has*;
  - *Can*.
- Un verbe à la troisième personne du singulier comme :
  - *Exists*;
- Une séquence de mots qui représente une phrase en langage naturel dont on a supprimé les espaces entre les mots.
- *ToBooleanOrDefault()* dans le cas d'une méthode qui projette un objet vers un booléen.

Voici ci-après quelques exemples d'application.

Propriété automatique

```
public class Myclass
{
    ...
    public bool IsValid { get; set; }
    ...
}
```

Propriété avec membre privé associé

```
#region IsReadOnly

        private bool _isReadOnly = false;
        /// <summary>
        /// ...
        /// </summary>
        public bool IsReadOnly
        {
            get
            {
                return this._isReadOnly;
            }
            set
            {
                this._isReadOnly = value;
            }
        }

#endregion
```

Méthode d'instance développée sous la forme d'une méthode d'extension

```
public static class MyclassExtenions
{
    public static bool IsEqualTo(this Myclass input, Myclass value)
    {
        //code omitted for brevity
    }
}

public class Myclass
{
  //code omitted for brevity
}
```

Plutôt que de développer les méthodes publiques d'instance sous la forme:

```
public class Myclass
{
  public bool IsEqualTo(Myclass value)
  {
      //code omitted for brevity
  }
}
```

Il est souvent plus intéressant de coder toutes les méthodes publiques d'instance en utilisant la technique des méthodes d'extension.

En effet, les méthodes d'extension permettent d'étendre le code d'une classe sans avoir accès au code source de cette classe. Il est donc possible d'enrichir n'importe quelle classe du Framework .Net avec ses propres méthodes. Par conséquent il est également possible d'enrichir ses propres classes sans avoir à modifier leur code source.

La méthode *ToBooleanOrDefault()* décrite ci-dessous permet par exemple d'étendre la classe System.String en lui ajoutant la méthode *ToBooleanOrDefault()*, d'où le terme méthode d'extension. L'extension se fait en effet sous la forme d'une méthode et non d'une propriété.

L'avantage apporté par la méthode d'extension vient du fait que :

- il est parfaitement valide d'invoquer cette méthode sur un objet nul : le traitement du cas nul peut ainsi être complètement encapsulé à l'intérieur de la méthode d'extension;
- La classe qui est étendue contient principalement des propriétés qui décrivent l'état de l'objet; Ce type de classe peut ainsi être utilisée pour transférer un objet d'une couche à une autre de l'application (de la couche business à la couche de présentation, ou bien de la couche data à la couche business). Ce type d'objet est appelé un DTO object (Data Transfer Object);
- Les méthodes d'extension décrivent séparément toutes les actions qu'il est possible de réaliser sur un objet issu de cette classe;
- Une méthode d'extension est testable unitairement.

Projection développée sous la forme d'une méthode d'extension

```
public static class StringExtensions
{
    /// <summary>
    /// Try to convert input string into a boolean.
    /// </summary>
    /// <param name="input">String to be converted.</param>
    /// <returns>
    /// Returns true when input string has one of the value "1", "ok", "OK", "Ok", "Tr
ue";
    /// Returns false for all other cases (like null, string.empty,"0", "Ko", etc ...).

    /// </returns>
    public static bool ToBooleanOrDefault(this string input)
    {
        //code omitted for brevity
    }
}
```

Séquence de mots qui représente une phrase

Pour déterminer le nom d'une méthode sous la forme d'une séquence de mots qui représente une phrase, commencez par créer un projet de test.

Ce projet de test va vous permettre d'utiliser en situation réelle le nom que vous allez choisir. Il va vous permettre de vérifier que le nom choisi est suffisamment intuitif et que le code induit est suffisamment expressif.

Le projet de test que vous avez crée contient une méthode de test par défaut :

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

L'écriture du code dans une méthode de test se fait en trois phases nommées *Arrange*, *Act*, *Assert*.

Mettez en évidence ces trois phases de la façon suivante à l'intérieure de la méthode de test:

```
[TestMethod]
public void TestMethod1()
{
    //Arrange

    //Act

    //Assert
}
```

La phase *Arrange* est la phase dans laquelle vous allez préparer tous les objets nécessaires pour exécuter votre future méthode.

La phase *Act* est la phase pendant laquelle vous allez exécuter votre méthode et en récupérer le résultat.

La phase *Assert* est la phase pendant laquelle vous allez comparer le résultat obtenu dans la phase *Act* avec le résultat attendu; en cas de différence vous signalez l'échec du test.

En appliquant ces principes pour créer un test de la méthode d'extension *ToBooleanOrDefault()* décrite ci-dessus, vous devriez obtenir un code de test semblable au code ci-dessous:

```
[TestMethod]
public void TestMethod1()
{
    //Arrange
    var input = "1";

    //Act
    var result = input.ToBooleanOrDefault();

    //Assert
    var expected = true;
    if (result != expected)
    {
        Assert.Fail();
    }
}
```

Pour créer le premier test unitaire de votre future méthode, il faut mettre en œuvre les étapes ci-dessous dans l'ordre indiqué:

1. Décrire brièvement ce que doit faire cette méthode sous la forme d'une seule phrase la

plus synthétique possible; cette phrase doit être si possible en anglais;
2. Transformer cette phrase en une ligne de code (typiquement la ligne de code que vous seriez amené à écrire dans la partie *ACT* du test unitaire) en supprimant les mots inutiles et en concaténant les mots restants pour former un nom conforme à la convention Pascal Casing;
3. Définir le nom obtenu sous la forme d'une méthode d'extension à l'extérieur du projet de test (c'est à dire dans le projet de votre application);
4. Commenter la méthode sous la forme de commentaires XML;
5. Écrire les spécifications de cette nouvelle méthode sous la forme d'exemples en commençant par les cas les plus simples;
6. Prendre le premier exemple de la spécification pour écrire la partie *Arrange* du premier test unitaire;
7. Mettre en place le code d'exécution de la nouvelle méthode dans la partie *Act* de la méthode de test en vérifiant que :
    - l'IntelliSense ramène bien le commentaire XML
    - Le commentaire affiché est cohérent avec le nom choisi
    - L'IntelliSense permet de découvrir et de manipuler rapidement cette nouvelle méthode
8. Valider le résultat dans la partie *Assert* de la méthode de test;
9. Vérifier que le test échoue;
10. Montrer l'intégralité de ce premier test unitaire aux autres membres de l' équipe pour vérifier qu'ils comprennent le code;
11. Vérifier que le nom choisi fait l'unanimité au sein de l'équipe.

Je vous propose d'appliquer ces 11 étapes pour développer une méthode qui consiste à déterminer si une chaîne de caractères est présente dans un tableau de chaînes de caractères.

Étape 1 : Décrire brièvement ce que doit faire la méthode

Vous pouvez construire cette phrase en la structurant de la manière suivante:

> L'objectif est de ... (verbe à l'infinitif) ... (sur ce quoi porte l'action définie par le verbe précédent) est dans une situation donnée (description de la situation) ou possède un attribut spécifique (description de l'attribut);

Dans le cas d'une projection d'un objet vers un booléen, la phrase est structurée de la manière suivante:

> L'objectif est de `...` (verbe à l'infinitif) `...` (sur ce quoi porte l'action définie par le verbe précédent) en `...` (résultat de la projection) ou bien de récupérer `...` (valeur par défaut) en cas d'impossibilité.

Par exemple:

- L'objectif est de `déterminer` `que le mot saisi par l'utilisateur est présent dans un tableau de valeurs prédéfinies`.

Autre exemple:

- L'objectif est de `convertir` `le texte présent dans une cellule d'un tableau` en un `booléen` ou bien de récupérer la valeur `false` en cas d'échec de la conversion.

Écrivez ensuite la phrase en anglais. Les deux exemples ci-dessus pourraient ainsi être traduits de la manière suivante:

- Checks if input string is present in a given array of values.
- Convert input string into a boolean or get false value when conversion fails.

Étape 2 : Transformer cette description en une ligne de code

Cette transformation peut se faire en plusieurs étapes.

Partez d'abord de l'étape précédente comme par exemple:

> Checks if input string is present in a given array of values.

Supprimez ensuite les espaces inutiles, concaténez les mots entre eux en respectant la convention PasCal Casing pour le nom de la méthode et la convention Camel Casing pour le nom des paramètres, séparez le sujet et le verbe par un point:

> Checks if inputString.IsPresentIn(givenArrayOfValues)

Supprimez tous les mots qui vous semblent inutiles pour obtenir une écriture la plus ramassée possible:

> Checks if input.IsIn(values)

Vérifiez qu'en remplaçant le début de la phrase par *var result* =, vous pouvez former une ligne de code valide pour la partie *ACT* de votre premier test unitaire:

```
//ACT
var result = input.IsIn(values);
```

Vérifiez ensuite qu'en situation réelle d'usage de la méthode, vous êtes capable de reformuler à l'identique la description de départ:

```
if (input.IsIn(values) )
{
    //Ce code est exécuté si le texte saisi par l'utilisateur est présent dans une lis
te de valeurs prédéfinies.
}
```

Vous allez certainement constater un écart entre la description de départ et la description reconstruite d'après la simple lecture du code.

Dans l'exemple ci-dessus la phrase reconstruite laisse penser que le paramètre *values* représente une liste de valeurs (`List<string>`) et non pas un tableau de valeurs (`string[]`).

Constater un écart signifie que l'usage de cette future méthode va poser un problème tôt ou tard soit pour vous même soit pour les autres développeurs de votre équipe, soit pour les personnes qui seront en charge de la maintenance future de l'application.

Même si cet écart vous paraît insignifiant ou négligeable, il est nécessaire de le réduire au maximum, car il introduit un risque majeur en terme de coûts quand il s'agira de faire évoluer l'application conformément aux attentes du marché ou du métier.

Pour réduire cet écart, demandez aux membres de l'équipe d'écrire ce qu'ils comprennent de votre code :

```
if (input.IsIn(values) )
{
    //Ce code est exécuté si <merci de compléter>
}
```

De cette demande d'avis va naître un échange qui vous permettra de trouver le nom de la méthode et en définir un usage qui fera l'unanimité.

Étape 3 : Définir le nom obtenu sous la forme d'une méthode d'extension

Pour créer une méthode d'extension, vous pouvez ajouter le code suivant soit dans le même fichier de classe soit dans un projet séparé si vous souhaitez réutiliser celle-ci dans différents projets:

```
public static class StringExtensions
{
    public static bool IsIn(this string input, string[] values)
    {
        //code omitted for brevity
    }
}
```

Notez que le nom du premier paramètre est nommé *input*, et que le nom du deuxième paramètre est *values* quand celui-ci désigne un ensemble d'objets, c'est à dire soit une liste `List<T>` soit un tableau `T[]` d'objet du même type, soit un nombre indéterminé d'ensembles d'objets `params Object[]`.

Ainsi la méthode d'extension ci-dessus pourrait aussi supporter les signatures suivantes:

```
public static class StringExtensions
{
    public static bool IsIn(this string input, IList<string> values)
    {
        //code omitted for brevity
    }
}
```

```
public static class StringExtensions
{
    public static bool IsIn(this string input, params Object[] values)
    {
        //code omitted for brevity
    }
}
```

Dans la méthode d'extension ci-dessus, le mot clé *params* permet de passer un nombre quelconque de listes ou de tableaux.

De manière générale je vous recommande de respecter la convention de nommage ci-dessous pour définir la signature d'une méthode d'extension:

> Le nom donné au premier paramètre d'une méthode d'extension est `input`
>
> > Si cette méthode d'extension nécessite d'accéder à un ensemble d'objets du même type:
> >
> > > le nom de ce paramètre est `values`
> >
> > Si cette méthode d'extension nécessite d'accéder à un ensemble d'objets de type différents, vous devez créer une classe qui regroupe tous ces objets sous la forme de propriétés, puis passer tous ces objets sous la forme d'une instance de cette classe:
> >
> > > le nom de ce paramètre est alors `context` quand les objets correspondants définissent un contexte d'exécution, ou bien `args` quand au moins un des objets est utilisé pour transférer une information entre l'appelant et l'appelé,
>
> et le nom de la méthode d'extension est postfixé par l'un des termes suivants:
>
> > In
> >
> > From
> >
> > Of
> >
> > With

Étape 4 : Commenter la méthode sous la forme de commentaires XML

Une fois que vous avez défini la signature de la méthode d'extension comme dans l'exemple ci-dessous:

```
public static class StringExtensions
{
    public static bool IsIn(this string input, string[] values)
    {
        //code omitted for brevity
    }
}
```

Insérez trois fois le caractère / au dessus de la déclaration de la méthode.

Ce raccourci permet de générer automatiquement les commentaires dits XML de la méthode d'extension:

```
public static class StringExtensions
{
    /// <summary>
    ///
    /// </summary>
    /// <param name="input"></param>
    /// <param name="values"></param>
    /// <returns></returns>
    public static bool IsIn(this string input, string[] values)
    {
        //code omitted for brevity
    }
}
```

Dans la balise `<summary> ...</summary` insérez la description de la méthode que vous avez réalisée à la fin de l'étape 1:

```
public static class StringExtensions
{
    /// <summary>
    /// Checks if input string is present in a given array of values.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="values"></param>
    /// <returns></returns>
    public static bool IsIn(this string input, string[] values)
    {
        //code omitted for brevity
    }
}
```

Renseignez le plus précisément possible la ou les balises `<param> ...</param>` :

```
public static class StringExtensions
{
    /// <summary>
    /// Checks if input string is present in a given array of values.
    /// </summary>
    /// <param name="input">Input string.</param>
    /// <param name="values">Array of strings</param>
    /// <returns></returns>
    public static bool IsIn(this string input, string[] values)
    {
        //code omitted for brevity
    }
}
```

Renseignez ensuite la balise `<returns>...</returns>` en vous aidant de ce que vous avez produit à l'étape 2:

```
if (input.IsIn(values) )
{
    //Ce code est exécuté si <merci de compléter>
}
```

Cela pourrait donner par exemple:

```
public static class StringExtensions
{
    /// <summary>
    /// Checks if input string is present in a given array of values.
    /// </summary>
    /// <param name="input">Input string.</param>
    /// <param name="values">Array of strings</param>
    /// <returns>Returns true if input string is present in the predefined array of va
lues.</returns>
    public static bool IsIn(this string input, string[] values)
    {
        //code omitted for brevity
    }
}
```

En résumé, tout le contenu des commentaires XML provient directement du travail effectué à l'étape 1 (Décrire brièvement ce que doit faire la méthode) et à l'étape 2 (Transformer cette description en une ligne de code) décrites ci-dessus.

Notez que la balise `<returns>...</returns>` décrit uniquement le cas où la méthode renvoie vrai.

Le contenu de la balise `<returns>...</returns>` doit suivre la règle suivante:

> Dans un commentaire XML, la balise `<returns>...</returns>` commence toujours par `Returns true if ...`

Autrement dit la balise `<returns>...</returns>` doit toujours décrire le cas positif. Si vous décrivez directement le cas négatif car cela vous semble plus direct ou plus naturel, cela signifie que le nom choisi nécessite de penser négativement ce qu'il faut à tout prix éviter: je vous expliquerai pourquoi dans le chapitre suivant.

Étape 5 : Écrire les spécifications

Cette étape est à mes yeux l'étape pivot, celle qui va permettre de dérouler naturellement les tests unitaires dans une approche TDD (Test Driven Development) s'il s'agit d'une action technique ou les tests fonctionnels dans une approche BDD (Behavior Driven Development) s'il s'agit d'une action métier.

Ecrivez les spécifications sous forme d'exemples.

Commencez d'abord par décrire les exemples les plus simples.

Un exemple simple est un exemple qui permet un traitement immédiat nécessitant très peu de lignes de code pour être implémenté. Vous constaterez que les exemples les plus simples sont souvent les moins probables.

Ecrivez ensuite des exemples de plus en plus complexe.

Finissez ensuite avec des exemples qui décrivent des exigences en terme de performance, c'est à dire en terme de vistesse d'exécution, en terme d'allocation mémoire ou en terme de pourcentage d'occupation de la CPU. J'appelle ces exigences des spécifications orientées performance.

> En résumé, l'écriture des spécifications se fait en trois étapes:
>
> - Commencer par les exemples les plus simples;
> - Continuer avec des exemples de moins en moins simples;
> - Finissez avec des exemples orientés performance.

Voici comment pourraient êtres écrites les spécifications de la méthode d'extension `IsIn()` :

Exemples simples

> Quand le tableau de valeurs est null, la méthode renvoie toujours faux.
>
> Quand le tableau de valeurs est vide, la méthode renvoie toujours faux.
>
> Quand la chaîne de caractères en entrée est nulle et que le tableau de valeurs contient un élement null, la méthode renvoie vrai.
>
> Quand la chaîne de caractères en entrée est vide et que le tableau de valeurs contient un élement vide, la méthode renvoie vrai.
>
> Quand la chaîne de caractères en entrée est présente à l'identique dans le tableau, la méthode renvoie vrai.

Exemples moins simples

> Quand la chaîne de caractères est présente dans le tableau non pas à l'identique mais avec une différence uniquement sur la casse, la méthode renvoi vrai.
>
> Quand la chaîne de caractères est présente dans le tableau non pas à l'identique mais avec des espaces en plus après la chaîne, la méthode renvoi vrai.

Exemples orientés performance

> Quand la chaîne de caractères n'est pas présente dans un tableau de valeurs contenant 100 élements, le temps d'éxécution de la méthode ne doit pas dépasser 0,5 millisecondes.

Étape 6 : Prendre le premier exemple de la spécification pour écrire la partie *Arrange* du premier test unitaire

Revenez sur votre projet de test. Mettez en place le premier test unitaire sous la forme:

```
[TestMethod]
public void TestMethod1()
{
    //Arrange

    //Act

    //Assert
}
```

Partez du premier exemple de la spécification:

> Quand le tableau de valeurs est null, la méthode renvoie toujours faux.

Eventuellement traduisez en anglais l'exemple:

> Should return false when the array of values is null

Pour donnez un nom suffisament évocateur à la méthode de test (et oui, même les méthodes de test doivent avoir un nom suffisament évocateur et donc nommer correctement une méthode de test est aussi important que nommer correctement la méthode ou la propriété qui est visée par la méthode de test), il suffit de concaténer tous les mots de la phrase:

```
[TestMethod]
public void QuandLeTableauDeValeursEstNullLaMéthodeRenvoieToujoursFaux()
{
    //Arrange

    //Act

    //Assert
}
```

ou bien en anglais:

```
[TestMethod]
public void ShouldReturnFalseWhenTheArrayOfValuesIsNull()
{
    //Arrange

    //Act

    //Assert
}
```

Il reste à définir dans un premier temps la partie *Arrange* du test:

```
[TestMethod]
public void ShouldReturnFalseWhenTheArrayOfValuesIsNull()
{
    //Arrange
    string input = "test";
    string[] values = null;

    //Act

    //Assert
}
```

Dans la partie *Arrange* vous préparez tous les objets nécessaires pour exécuter la méthode. Pour la méthode d'extension `IsIn()`, il suffit de définir la chaîne de caractères en entrée ainsi que le tableau de valeurs.

L'exemple ci-dessus est un exemple simple. Dans le cas d'un test unitaire fonctionnel, la partie *Arrange* peut contenir tout type de code permettant de créer des objets métiers dans un état spécifique. Cependant le nombre de lignes de code dans la partie *Arrange* ne doit jamais excéder cinq lignes.

Étape 7 : Mettre en place le code d'exécution de la nouvelle méthode dans la partie *Act* de la méthode de test

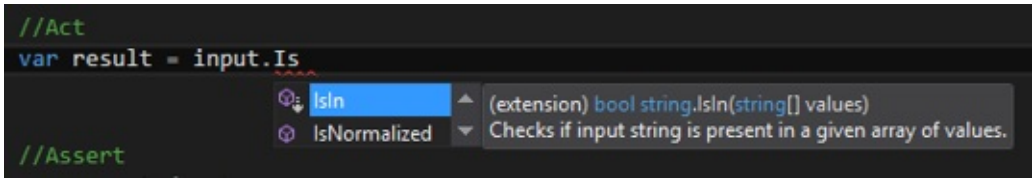
La phase *Act* est la phase pendant laquelle vous allez exécuter votre méthode et en récupérer le résultat:

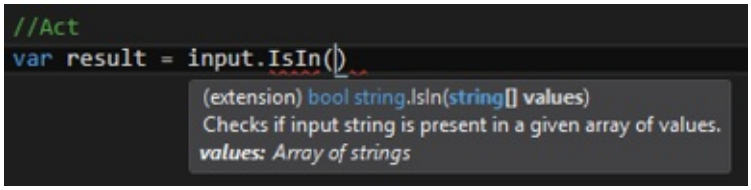
```
[TestMethod]
public void ShouldReturnFalseWhenTheArrayOfValuesIsNull()
{
    //Arrange
    string input = "test";
    string[] values = null;

    //Act
    var result = input.IsIn(values);

    //Assert
}
```

Cette phase est l'une des plus importantes car vous allez vérifier que :

- l'IntelliSense ramène bien le commentaire XML;
- Le commentaire affiché est cohérent avec le nom choisi;
- L'IntelliSense permet de découvrir et de manipuler rapidement cette nouvelle méthode.

Vous devriez obtenir quelque chose de similaire aux copies d'écran ci-dessous:





Étape 8 : Valider le résultat dans la partie *Assert* de la méthode de test

La phase *Assert* est la phase pendant laquelle vous allez comparer le résultat obtenu dans la phase *Act* avec le résultat attendu; en cas de différence vous signalez l'échec du test:

```
[TestMethod]
public void ShouldReturnFalseWhenTheArrayOfValuesIsNull()
{
    //Arrange
    string input = "test";
    string[] values = null;

    //Act
    var result = input.IsIn(values);

    //Assert
    var expected = true;
    if (result != expected)
    {
        Assert.Fail();
    }
}
```

L'objet `Assert` expose bien d'autres méthodes que la méthode `Fail()` utilisée ci-dessus, comme par exemple:

```
Assert.AreSame(result, expected);
Assert.Equals(result, expected);
Assert.IsFalse(result);
```

N'employez aucune de ces méthodes car elles peuvent introduire ce qu'on appelle un false positive, c'est à dire que le teste unitaire passe au vert alors qu'il devrait rester au rouge.

Définissez explicitement le résultat attendu:

```
var expected = true;
```

Signalez l'échec du test sous la forme la plus basique qui soit:

```
Assert.Fail();
```

Signalez l'échec du test uniquement quand le résultat est différent de ce qui est attendu en testant le plus simplement et le plus directement la non égalité des deux objets:

```
if (result != expected)
{
    Assert.Fail();
}
```

Étape 9 : Vérifier que le test échoue

Une fois que vous avez mis en place les trois phases du test, vous pouvez exécuter le test. Dans une approche TDD cette première exécution doit aller à l'échec.

Pour faire échouer la première exécution du test, la première et unique ligne de code de la méthode à tester doit être:

```
throw new NotImplementedException();
```

Avant la première exécution du test assurez vous donc que la méthode sous test est codée de la manière suivante:

```
public static bool IsIn(this string input, string[] values)
{
    throw new NotImplementedException();
}
```

Étape 10 : Montrer l'intégralité de ce premier test unitaire aux autres membres de l'équipe pour vérifier qu'ils comprennent le code

A compléter

Étape 11 : Vérifier que le nom choisi fait l'unanimité au sein de l'équipe

A compléter

Toujours penser positif

Si un des facteurs clés pour délivrer rapidement une application est de coder en ayant à l'esprit de raconter une histoire et le chapitre précédent vous a donné les éléments de base pour créer des noms suffisamment expressifs pour rendre la lecture du code source similaire à la lecture d'une histoire, l'autre élément clé du succès est la pensée positive.

En terme de programmation, la pensée négative se résume principalement sous la forme du pattern ci-dessous :

```
if ( une certaine condition n'est pas remplie )
{
  // la plupart du temps beaucoup de lignes de code
}
else {
  // la plupart du temps très peu de lignes de code
}
```

Ce pattern est tellement commun que la plupart des développeurs que j'ai pu rencontrer pensent qu'il s'agit d'un standard de programmation.

La pensée négative pose cependant deux problèmes:

> Le premier problème est que l'usage de ce pattern est en contradiction avec une approche TDD (Test Driven Development) et/ou BDD (Behavior Driven Development) et de manière générale est en contradiction avec la façon de coder un test unitaire.

En effet pour tester le code suivant:

```
// A est une expression booléenne
if ( ! A )
{
  // la plupart du temps beaucoup de lignes de code
}
```

Il va falloir mettre en place autant de tests unitaires que de cas où l'expression A est fausse. Il est vraisemblable que ces cas forment un ensemble ouvert, c'est à dire un ensemble non exhaustif des cas possibles.

J'aime bien comparer le pattern ci-dessus à un tireur à l'arc qui doit atteindre le centre d'une cible située à 100 mètres. L'expression `A` dans le pattern ci-dessus représente le centre de la cible , `! A` représente toutes les positions possibles de la flèche quand elle n'atteint pas le centre, soit une infinité de positions.

Le risque de tomber sur un cas imprévu quand l'application sera déployée en production est ainsi extrêmement élevé.

Dans un contexte TDD ou BDD, la spécification associée à un test est exprimée grossièrement de la manière suivante :

> Quand je suis dans la situation `A`, je m'attends à avoir le résultat `R`.

C'est le principe de la spécification par l'exemple.

L'écriture de la méthode ou de la propriété à tester d'après cette spécification se ferait ainsi en deux temps. Dans un premier temps la méthode ou la propriété contient une seule ligne de code:

```
throw new NotImplementedException();
```

Cette ligne de code garanti que le test unitaire va à l'échec lors de sa première exécution.

Dans un deuxième temps le contenu de la méthode ou de la propriété devient:

```
if ( A )
{
  // la plupart du temps peu de lignes de code
  return R
}

throw new NotImplementedException();
```

Vous pouvez ainsi constater que le code obtenu est la version positive du pattern évoqué en début de chapitre.

La pensée négative est donc en contradiction avec une approche TDD et/ou BDD, et de manière générale est en contradiction avec la façon de coder un test unitaire.

> Le deuxième problème posé par la pensée négative est qu'elle réduit considérablement la productivité du développeur ainsi que la lisibilité et la compréhension du code ce qui affectera les délais de correction ou de mise à disposition de nouvelles fonctionnalités.

Je vous invite à analyser la méthode ci-dessous et à déterminer quel devrait être le nom réel de cette méthode:

```
public static bool F(this string path)
{
    if (path != null)
    {
        int num = path.Length;
        while (--num >= 0)
        {
            char c = path[num];
            if (c == '.')
            {
            return num != path.Length - 1;
            }
        }
    }
    return false;
}
```

Notez combien de temps il vous a fallu pour déterminer le nom exact de cette méthode. Demandez aux développeurs de votre équipe d'analyser ce code et de trouver le meilleur nom pour cette méthode. Notez le temps qu'ils ont mis pour accomplir cette tâche.

Déterminez ainsi le temps moyen mis pour analyser une seule ligne de code (en divisant la durée d'analyse par le nombre effectif de lignes de code - soit 7 dans l'exemple ci-dessus).

Voici un autre exemple de code qui est une variante de la pensée négative basée sur l'usage de l'opérateur ternaire `?` :

```
public void ExportToCsv(bool isDetailedExport, bool isAnnualReport)
{
    var connection = new SqlConnection("...");
    var command = connection.CreateCommand();
    command.CommandType = System.Data.CommandType.StoredProcedure;
    command.CommandText = !isDetailedExport ? (!isAnnualReport ? "ps4" : "ps2"):(isAnn
ualReport?"ps3":"ps1");
    // code omitted for brevity
}
```

Pour ce type de code la démarche est la suivante: déterminez quelles sont les conditions pour que la procédure stockée appelée soit `ps1`, mais surtout déterminez combien de temps il vous faut pour répondre de façon certaine à la question.

Demandez aux développeurs de votre équipe d'analyser ce code et de trouver la réponse à la question. Notez le temps qu'ils ont mis pour accomplir cette tâche.

Déterminez ainsi le temps moyen mis pour analyser une seule ligne de code (en divisant la durée d'analyse par le nombre effectif de lignes de code - soit une seule ligne de code dans l'exemple ci-dessus).

Déterminez ensuite le nombre de lignes de code, dans l'application que vous développez, qui suivent le pattern de la pensée négative. Imaginons par exemple que vous avez déterminé que le temps moyen pour analyser une ligne de code (à partir des deux exemples ci-dessus) est de 4 secondes et qu'il y a 1000 lignes de code dans votre application qui suivent le pattern de la pensée négative: chaque correction ou chaque évolution de l'application prendra vraisemblablement au moins une heure de plus (pour chaque développeur affecté à cette correction ou évolution) qu'initialement prévu au planning.

La pensée négative a donc un coût qu'il est possible de quantifier en utilisant la méthodologie exposée à partir des exemples ci-dessus.

La pensée négative, en fonction de la taille de l'application, en fonction du pourcentage de lignes de code écrites en pensée négative, en fonction du nombre de développeurs dans l'équipe, peut entraîner un surcoût pouvant aller de 10% à 100% du budget initialement prévu.

L'objectif de ce chapitre est de vous montrer les techniques de base qui vont vous permettre de toujours penser positif.

Ne jamais utiliser l’opérateur de négation !

L'opérateur de négation utilisé dans le langage .Net mais aussi dans le langage C et JavaScript est un point d'exclamation qui peut être utilisé de trois façons différentes:

```
// A est une expression booléenne
if ( ! A )
{
  //la plupart du temps beaucoup de lignes de code
}
```

```
// A est un objet
if ( A != null )
{
  //la plupart du temps beaucoup de lignes de code
}
```

```
// A et B sont des objets du même type
if ( A != B )
{
  //la plupart du temps beaucoup de lignes de code
}
```

L'usage de la négation à l'aide d'un point d'exclamation provoque un ralentissement du cerveau pour deux raisons:

> L'usage de la négation rentre en conflit avec d'autres usages dans la vie de tous les jours.

En effet le point d'exclamation peut indiquer un danger comme le montre l'illustration ci-dessous:

Le point d'exclamation peut indiquer un impératif quand il est utilisé en fin de phrase comme par exemple :

> merci de respecter les délais!!!

Dans un cas comme dans l'autre, le cerveau cesse toutes ses activités pour analyser la situation actuelle et déterminer quelles seraient les conséquences de ne pas tenir compte du danger ou de l'ordre.

> L'usage de la négation à l'aide d'un signe placé devant une affirmation est sans équivalent dans le langage naturel.

Par conséquent pour analyser le code:

```
if ( ! A )
{
  //la plupart du temps beaucoup de lignes de code
}
```

il faut réaliser les opérations suivantes:

- Ignorer dans un premier temps la présence du signe `!` ;
- Transformer l'expression A en une phrase (plus l'expression A est compliquée - présence multiple de `&&` et de `||` - plus cette transformation est longue);
- Tourner cette phrase sous une forme négative;
- Déterminer dans quelles circonstances cette forme négative est vrai;
- Refaire au moins une fois toutes ces opérations pour vérifier qu'on ne s'est pas trompé.

L'usage de l’opérateur de négation `!` entraîne donc un arrêt brutal de la lecture du code environnant mais aussi un ralentissement significatif au moment de l'écriture du code.

Pour toutes ces raisons, il faut toujours éviter d'utiliser l'opérateur de négation `!`

Je vais vous montrer qu'il est parfaitement possible de se passer totalement de cet opérateur.

Partons dans un premier temps du pattern de la pensée négative exposé en introduction de ce chapitre:

```
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
}
else {
  C //la plupart du temps très peu de lignes de code
}
```

Dans l'exemple ci-dessus `A` représente une expression booléenne, `B` le bloc de code à l'intérieur du `If` , et `C` le bloc de code à l'intérieur du `else` . Supposons dans un premier temps qu'il n'y a pas de code ni avant le `if` ni après le `else` c'est à dire que le code ci-dessus est entièrement encapsulé dans une méthode ou une propriété.

Il est possible de réécrire ce code de la manière suivante:

```
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
}
if ( A )
{
  C //la plupart du temps très peu de lignes de code
}
```

Notez la disparation du `else` et notez qu'il y a maintenant deux if qui sont indépendants l'un de l'autre: il est donc possible d'inverser les deux `if` .

```
if ( A )
{
  C //la plupart du temps très peu de lignes de code
}
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
}
```

Il est possible d'ajouter un `return` (ou un `return R` si la méthode renvoie un résultat ou s'il s'agit d'une propriété) à la fin du bloc de code `C` sans que cela change quoi que ce soit à l'exécution:

```
if ( A )
{
  C //la plupart du temps très peu de lignes de code
  return;
}
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
}
```

Il est maintenant possible de se débarrasser de l'opérateur de négation:

```
if ( A )
{
  C //la plupart du temps très peu de lignes de code
  return;
}

B //la plupart du temps beaucoup de lignes de code
```

Pour transformer en version positive un code négatif, il suffit donc d'appliquer les deux règles suivantes:

> La dernière instruction d'un bloc de code `If` est toujours `return` ou `continue`
>
> Un `If` n'a jamais de `else`

Si ces deux règles sont nécessaires elles ne sont pas suffisantes.

En effet, il reste à étudier le cas suivant:

```
A1
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
}
else {
  C //la plupart du temps très peu de lignes de code
}
A2
```

Dans l'exemple ci-dessus `A` représente une expression booléenne, `B` le bloc de code à l'intérieur du `If`, `C` le bloc de code à l'intérieur du `else`. `A1` représente toutes les lignes de code situées avant le `if` et `A2` toutes les lignes de code situées après le `else`.

Le cas ci-dessus est équivalent à l'écriture suivante:

```
A1
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
  A2
}
else {
  C //la plupart du temps très peu de lignes de code
  A2
}
```

Il est maintenant possible d'appliquer la transformation que vous avez apprise:

```
A1
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
  A2
}
if ( A )
{
  C //la plupart du temps très peu de lignes de code
  A2
}
```

puis:

```
A1
if ( A )
{
  C //la plupart du temps très peu de lignes de code
  A2
}
if ( ! A )
{
  B //la plupart du temps beaucoup de lignes de code
  A2
}
```

puis:

```
A1
if ( A )
{
  C //la plupart du temps très peu de lignes de code
  A2
  return;
}
B //la plupart du temps beaucoup de lignes de code
A2
```

Vous pouvez remarquer que les lignes représentées par `A2` sont dupliquées à l'issue de la transformation positive.

Ainsi l'insertion d'un bloc `if...else` à l'intérieur d'un bloc de code est une façon déguisée de dupliquer tout le code situé après le `else`.

Le codage en pensée positive a donc l'avantage de mettre en évidence la duplication de code et permet donc de découvrir très rapidement quelle partie de code doit être factorisée sous la forme d'une méthode.

Cette fois-ci pour terminer cette transformation en pensée positive il a fallu ajouter la règle suivante:

> Toute ligne de code (ou ensemble de lignes) dupliquée doit être factorisée.

Le code ci-dessous est parfaitement conforme avec la pensée positive. Par contre il pose un problème dans la mesure où le deuxième if imbriqué ne se termine pas par un `return` :

```
if ( A )
{
  if ( B )
  {
    C
  }
  D
  return;
}
```

Le code ci-dessous est équivalent au code suivant:

```
if ( A )
{
  if ( B )
  {
    C
    D
    return;
  }
  if ( ! B )
  {
    D
    return;
  }
  return;
}
```

Vous pouvez remarquer qu'un `if` imbriqué est une façon déguisé de penser négatif et est aussi une façon déguisée de dupliquer du code (en effet toutes les lignes représentée par `D` sont dupliquées). Cependant il est possible de transformer le code ci-dessous en une version semi-positive:

```
if ( A && B )
{
  C
  D
  return;
}
if ( A && !B )
{
  D
  return;
}
```

Pour effectuer cette transformation semi-positive il a fallu ajouter la règle suivante :

> Un `if` ne contient jamais de `if` imbriqué.

L'application de cette règle induit cependant une complexification des expressions : d'un `if (A) {}` et `if (B) {}`, on est passé à un `if (A && B) {}` et `if (A && !B) {}`.

En réalité, l'application des règles ci-dessus permet de mettre rapidement en évidence l'ensemble des contextes possibles lors de l'exécution du code. Le code ci-dessus pourrait être décrit par le pseudo-code suivant:

```
if ( contexte 1)
{
  C
  D
  return;
}

if (contexte 2 )
{
  D
  return;
}
```

La façon de coder ce contexte peut être multiple. Dans les sections qui vont suivre, je vais vous donner les clés pour réaliser au mieux cette opération.

La première de ces clés est de toujours coder une expression booléenne sous la forme d'un appel à une méthode d'extension sur l'objet qui vous semble le plus approprié : c'est l'objet de la prochaine section.

Comment coder une expression booléenne

Votre code doit raconter une histoire; par conséquent toute expression booléenne telle que :

```
if ( A )
{
  //code omitted for brevity
}
```

doit être écrite sous la forme d'une phrase positive.

Par défaut, et dans la mesure du possible, vous devez toujours formuler de manière positive l'expression booléenne que vous êtes sur le point de coder.

Pour réaliser cela, substituer l'expression `A` par une méthode d'extension définie sur le type d'objet qui vous semble le plus approprié.

Cette méthode d'extension doit être nommée en utilisant l'une des trois options ci-dessous:

- avoir le préfixe :
  - *Is*;
  - *Has*;
  - *Can*.
- Être Un verbe à la troisième personne du singulier comme :
  - *Exists*;
- Etre Une séquence de mots qui représente une phrase en langage naturel dont on a supprimé les espaces entre les mots.

Pour mettre en œuvre la méthodologie de création de nom d'une méthode ou d'une propriété qui renvoie un booléen reportez vous à la section correspondante : Comment nommer une méthode ou une propriété qui renvoie un booléen.

Le code snippet ci-dessus peut donc être réécrit de la manière suivante:

```
var myObject = ... //code omitted for brevity
if ( myObject.IsXXX() )
{
  //code omitted for brevity
}
```

ou bien

```
var myObject = ... //code omitted for brevity
if ( myObject.HasXXX() )
{
  //code omitted for brevity
}
```

ou bien

```
var myObject = ... //code omitted for brevity
if ( myObject.CanXXX() )
{
  //code omitted for brevity
}
```

Notez qu'il faut toujours commencer par coder l'usage de la méthode d'extension avant même de coder son implémentation. Si vous utilisez la méthodologie TDD vous trouverez cette approche normale, dans le cas contraire cette approche devrait vous inciter à mettre en œuvre cette méthodologie.

Une fois que vous avez défini le nom et l'usage de la méthode d'extension, il reste à l'implémenter.

La première ligne de code de cette méthode d'extension doit être :

```
throw new NotImplementedException();
```

Ceci est la garantie que la première exécution du premier test unitaire associée à cette méthode va à l'échec.

Si vous êtes dans une approche TDD et si vous avez défini les spécifications associées à cette méthode d'extension et bien commencez par mettre en place le code qui fait passer la première spécification.

Pour savoir commencer définir les spécifications d'une méthode booléenne, reportez vous à la section correspondante : Comment nommer une méthode ou une propriété qui renvoie un booléen.

Si vous n'avez pas défini de spécifications et que vous souhaitez déjà coder l'expression booléenne sous la forme d'une méthode d'extension, appuyez vous sur les règles suivantes pour structurer le contenu de votre code:

> Quand vous développez une méthode, vous devez en sortir le plus vite possible. Autrement dit si vous pouvez déterminer un cas de figure qui permet de faire immédiatement un `return` écrivez d'abord ce cas.
>
> La dernière instruction d'une méthode booléenne est toujours: `return false;`
>
> le bloc de code qui précède la dernière ligne de code d'une méthode booléenne est toujours de la forme:
>
> ```
> if ( A )
> {
>   //code omitted for brevity
>   return true;
> }
> ```
>
> L'expression `A` ci-dessus peut être exprimée en pensée positive ou en pensée négative.

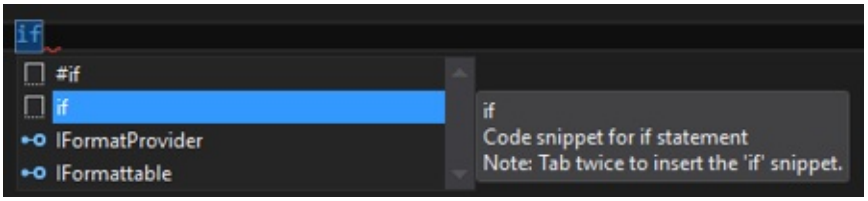
Une méthode booléenne doit donc toujours être structurée de la manière suivante:

```
public static bool IsIn(this string input, string[] values)
{
    //code omitted for brevity
    if ( A )
    {
      //code omitted for brevity
      return true;
    }
    return false;
}
```

ou bien

```
public static bool IsIn(this string input, string[] values)
{
    //code omitted for brevity
    if ( ! A )
    {
      //code omitted for brevity
      return true;
    }
    return false;
}
```

Vous verrez dans la section suivante comment coder une expression négative sans utiliser l'opérateur de négation `!`.

> Quand vous codez un `if` procédez de la manière suivante: Tapez le mot clé `if` puis attendez que l'IntelliSense vous montre l'existence du code snippet associé:
>
> 
>
>
> Appuyez ensuite deux fois sur la touche `TAB`. Visual Studio substitue le mot clé `if` par le code:

```
if (true)
{

}
```

Notez que Visual Studio vous incite par défaut à penser et à coder positif.

Il ne vous reste plus qu'à remplacer `true` par une expression booléenne codée en utilisant la technique décrite dans cette section.

Si la méthode que vous développez contient une boucle `for` ou `foreach`, vous pouvez structurer votre boucle en vous appuyant sur la règle suivante:

> Quand vous développez une boucle `for` ou `foreach`, vous devez en sortir le plus vite possible. Autrement dit si vous pouvez déterminer un cas de figure qui permet de passer immédiatement à l'itération suivante ,écrivez d'abord ce cas.
>
> Si vous pouvez déterminer un cas de figure qui permet de sortir de la boucle, écrivez d'abord ce cas.

Autrement dit, une boucle `for` ou `foreach` doit toujours être codée de la manière suivante:

```
foreach (var item in input)
{
    if (A)
    {
        //code omitted for brevity
        continue;
    }
    if (B)
    {
        //code omitted for brevity
        return result;
    }
    //code omitted for brevity
}
```

A compléter

Comment coder négativement une expression booléenne sans utiliser l'opérateur de négation

Il existe des cas où le fait de penser négatif permet de faire immédiatement un `return` dans l'implémentation d'une méthode ou d'une propriété.

Votre code doit raconter une histoire; par conséquent toute expression booléenne telle que :

```
if ( ! A )
{
  //code omitted for brevity
}
```

doit être écrite sous la forme d'une phrase négative.

Pour réaliser cela, substituer l'expression `! A` par une méthode d'extension définie sur le type d'objet qui vous semble le plus approprié.

Cette méthode d'extension doit être nommée en utilisant l'une des trois options ci-dessous:

- avoir le préfixe :
  - *IsNot*;
  - *HasNo*;
  - *Cannot*.
- Être Un verbe à la troisième personne du singulier comme :
  - *DoesNotExist*;
- Être Une séquence de mots qui représente une phrase négative en langage naturel dont on a supprimé les espaces entre les mots.

Pour mettre en œuvre la méthodologie de création de nom d'une méthode ou d'une propriété qui renvoie un booléen reportez vous à la section correspondante : Comment nommer une méthode ou une propriété qui renvoie un booléen.

Le code snippet ci-dessus peut donc être réécrit de la manière suivante:

```
var myObject = ... //code omitted for brevity
if ( myObject.IsNotXXX() )
{
  //code omitted for brevity
}
```

ou bien

```
var myObject = ... //code omitted for brevity
if ( myObject.HasNoXXX() )
{
  //code omitted for brevity
}
```

ou bien

```
var myObject = ... //code omitted for brevity
if ( myObject.CannotXXX() )
{
  //code omitted for brevity
}
```

ou bien

```
var myObject = ... //code omitted for brevity
if ( myObject.DoesNotExist() )
{
  //code omitted for brevity
}
```

La mise en œuvre de cette méthode d'extension se fait en trois étapes:

1. Implémentation de la version positive de la méthode;
2. Mise en œuvre des tests unitaires sur la version positive uniquement;
3. Définition de la version négative sous la forme montrée dans les exemples ci-dessous:

```
public static bool IsIn(this string input, string[] values)
{
    //code omitted for brevity
}

public static bool IsNotIn(this string input, string[] values)
{
    return input.IsIn(values) == false;
}
```

```
public static bool HasXXX(this MyClass input)
{
  //code omitted for brevity
}

public static bool HasNoXXX(this MyClass input)
{
    return input.HasXXX() == false;
}
```

Remarquez que la version négative est toujours codée en s'appuyant sur la forme positive :

```
return <expresion positive> == false;
```

Remarquez aussi que la version négative est codée sans utiliser l'opérateur de négation `!`.

En résumé il est possible de ne jamais coder explicitement la forme négative d'une expression.

Mais surtout gardez à l'esprit la règle suivante:

> Ne jamais tester unitairement la version négative d'une méthode

Ne jamais commenter à l'intérieur d'un bloc de code

Très souvent il m'arrive de vouloir commenter le bloc de code que je suis en train de mettre en place. Très souvent le commentaire est là pour permettre à soi même ou aux autres développeur de l'équipe de bien comprendre ce qui se passe.

Documenter unitairement une ou plusieurs lignes de code à l'intérieur d'une méthode ou d'une propriété permet de masquer la complexité de l'écriture du code ou de l'algorithmique.

Ce type de commentaire est en réalité une autre façon de penser négatif non pas vis à vis de son propre code mais vis à vis des autres développeurs qui seront amenés à le modifier dans l'avenir.

Le message implicite est le suivant : comme il y a de forte chance que vous ne compreniez pas le code que j'ai écris je vais vous faire gagner du temps en vous expliquant ce qu'il fait.

Le commentaire masque la complexité du code associé mais surtout masque très souvent le fait que ce code implémente une règle métier ou une algorithmique spécifique empêchant ainsi d'externaliser cette règle ou cet algorithme dans une autre partie du code ou dans une autre couche, empêchant ainsi toute forme de maintenance, de factorisation ou d'évolution du code associé.

Autrement dit, un commentaire est le révélateur d'un morceau de code qui n'est pas à sa place et qui sera difficilement maintenable.

L'usage des commentaires à l'intérieur d'un bloc de code pose aussi les problèmes suivants:

- Ces commentaires sont perdus à la compilation;
- Ces commentaires sont souvent trop courts pour ne pas gêner la lecture du code environnant;
- Ces commentaires échappent à tout système de génération automatique de la documentation du logiciel tel que SandCastle.

  Si vous êtes amené à documenter du code à l'intérieur d'une méthode ou d'une propriété, considérez le fait de remplacer ce code par une méthode d'extension ayant un nom suffisamment évocateur et de déplacer votre commentaire dans la documentation XML associée à la méthode d'extension.

  En procédant de la sorte, vous rendez votre code plus lisible, plus maintenable mais surtout vos commentaires seront conservés à la compilation et seront affichés dynamiquement par l'IntelliSense quand un développeur utilisera votre méthode

d'extension.

Comment remplacer un IF...ELSE par une projection

Beaucoup de constructions `if...else` sont en réalité des projections et peuvent être substituées en développant une méthode d'extension nommée `ToXXX()` sur l'objet concerné par cette projection.

Voici un premier exemple tiré d'une couche d'accès au données:

```
var input = "test";
if (input != null)
{
    command.Parameters.Add(new SqlParameter("@UserLogin", input));
}
else
{
    command.Parameters.Add(new SqlParameter("@UserLogin", System.DBNull.Value));
}
```

L'exemple ci-dessus consiste en réalité à projeter un objet de type `string` soit vers lui-même soit vers la constante `System.DBNull.Value`.

Si vous êtes amené à développer une couche d'accès aux données en vous appuyant sur ADO.NET, vous serez amené à répéter très souvent ce pattern.

L'exemple ci-dessus peut être factorisé de la façon suivante:

```
var input = "test";
command.Parameters.Add(new SqlParameter("@UserLogin", input.ToStringOrDBNull() ));
```

`ToStringOrDBNull()` est une méthode d'extension qui pourrait être définie de la manière suivante:

```
public static object ToStringOrDBNull(this string input)
{
    if (input == null)
    {
        return System.DBNull.Value;
    }
    return input;
}
```

Voici un autre exemple similaire au précédent:

```
int? input = 10;
if (input != null && input.HasValue)
{
    command.Parameters.Add(new SqlParameter("@Quantity", input.Value));
}
else
{
    command.Parameters.Add(new SqlParameter("@Quantity", System.DBNull.Value));
}
```

cet exemple peut être refactorisé de la manière suivante:

```
int? input = 10;
command.Parameters.Add(new SqlParameter("@Quantity", input.ToValueOrDBNull() ));
```

`ToValueOrDBNull()` est une méthode d'extension qui pourrait être définie de la manière suivante:

```
public static object ToValueOrDBNull<T>(this T? input) where T : struct
{
    if (input == null)
    {
        return System.DBNull.Value;
    }

    if (input.HasValue)
    {
        return input.Value;
    }

    return System.DBNull.Value;
}
```

Cette méthode d'extension peut ainsi être réutilisée pour être appliquée sur n'importe quelle structure : `int` , `long` , `decimal` , `float` , etc...

Comment remplacer l'opérateur ternaire ? par une méthode d'extension

L'opérateur ternaire `?` permet de coder un `if...else` sur une seule ligne.

```
var result = (A) ? expression1 : expression2;
```

La ligne ci-dessus est équivalente au code ci-dessous:

```
if (A)
{
    result = expression1;
}
else
{
    result = expression2;
}
```

L'opérateur ternaire `?` est donc une manière élégante d'écrire un `if...else`.

Cette élégance conduit inévitablement au syndrome connu qui consiste à tenter de coder un maximum de choses en une seule ligne comme le montre l'exemple ci-dessous:

```
string code = ... //product code;
string libelle = ... //product description;
string lib = (libelle != "" && !string.IsNullOrEmpty(libelle)) ? string.Concat(code, "
 - ", libelle) : code;
```

L'opérateur ternaire `?` est une manière déguisée de coder en pensée négative mais est aussi une des causes principales du fameux "pourquoi faire simple quand on peut faire compliquer".

Dans la section précédente je vous ai montré qu'il est très souvent possible de remplacer un `if...else` par une projection.

Cela est moins vrai concernant l'opérateur ternaire `?`.

En effet, cet opérateur est souvent utilisé pour exprimer une exigence métier, une exigence technique (comme par exemple une règle d'affichage - qui est en réalité l'expression d'une exigence métier au niveau de l'interface graphique).

Pour factoriser ce type de code ou pour vous aider à ne plus utiliser cet opérateur, appuyez vous sur les règles suivantes:

> Substituez l'usage de l'opérateur ternaire `?` par une méthode d'extension.
>
> Sur une ligne de code ne faites qu'une seule chose à la fois.

Je vais vous montrer comment substituer l'opérateur `?` dans l'exemple ci-dessus.

Il faut dans un premier temps décrire sous la forme d'une phrase positive la ligne de code contenant l'opérateur ternaire. Cette phrase doit être la plus simple possible:

```
Quand le produit a une description
Le code de ce produit est concaténé avec un séparateur et la description.
```

Puis transformer cette phrase en du pseudo code, en supprimant tous les mots inutiles et en concaténant ceux qui restent. La phrase ci-dessus pourrait être alors transcrite de la manière suivante (pseudo-code):

```
string label = productCode;
if (Product.HasDescription)
{
  label = productCode.ConcatWithSeparatorAndValue(" - ", productDescription);
}
```

La méthode `ConcatWithSeparatorAndValue()` peut être développée sous la forme d'une méthode d'extension :

```
public static string ConcatWithSeparatorAndValue(this string input, string separator,
string value)
{
    if (input == null)
    {
        return null;
    }

    if (value == null)
    {
        return input;
    }

    if (value == string.Empty)
    {
        return input;
    }

    if (separator == null)
    {
        return input.ConcatWithSeparatorAndValue(string.Empty, value);
    }

    var result = string.Format(@"{0}{1}{2}",input,separator,value);
    return result;

}
```

Grâce à cette méthode d'extension, le code initial :

```
string code = ... //product code;
string libelle = ... //product description;
string lib = (libelle != "" && !string.IsNullOrEmpty(libelle)) ? string.Concat(code, "
 - ", libelle) : code;
```

devient:

```
string code = ... //product code;
string libelle = ... //product description;
string lib = code.ConcatWithSeparatorAndValue(" - ",libelle);
```

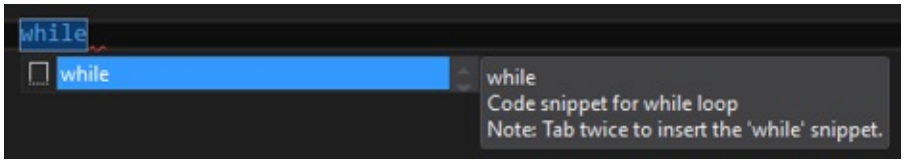
La méthodologie exposée ci-dessus est itérative. Cette première itération a permis d'externaliser une exigence métier et a permis de rendre le code plus lisible.

A compléter

Comment coder une boucle While

Pour coder une boucle `while` , vous pouvez vous appuez sur le code snippet du même nom.

Tapez le mot clé `while` puis attendez que l'IntelliSense vous montre l'existence du code snippet associé:



Appuyez ensuite deux fois sur la touche `TAB` . Visual Studio substitue le mot clé `while` par le code:

```
while (true)
{

}
```

Notez que Visual Studio vous incite par défaut à penser et à coder positif une boucle `while` .

L'astuce consiste ici à ne jamais modifier l'expression booléenne en entrée de la boucle `while` , c'est à dire qu'une boucle `While` une fois codée doit toujours être de la forme:

```
while (true)
{
  //code omitted for brevity
}
```

Cette approche vous mets dans la position de faire la preuve que votre code permettra de sortir de la boucle à l'exécution.

La mise en œuvre d'une boucle `while` entraîne un risque non négligeable d'une boucle infinie, la conséquence d'une boucle infinie étant dans le meilleure des cas un stack overflow et dans le pire des cas un redémarrage de la machine.

Il est donc vital d’apporter la preuve qu'à l’exécution le code sortira toujours de la boucle.

Une boucle while doit être codée en s'inspirant du modèle ci-dessous:

```
int loopCounter = 0;
int loopCounterMaxValue = 1000;
while (true)
{
    loopCounter += 1;
    if (loopCounter > loopCounterMaxValue)
    {
        //TODO : log
        break;
    }

    if (A)
    {
        break;
    }

    //code ommited for brevity

}
```

Dans l'exemple ci-dessus `A` représente l'expression qui détermine la sortie de la boucle, `loopCounterMaxValue` a une valeur arbitraire que vous devrez adapter à votre cas.

L'opérateur new

Pour créer un objet il suffit de créer une instance de la classe voulue en utilisant l'opérateur `new` comme dans l'exemple ci-dessous:

```
var myObject = new MyClass();
```

L'usage de cet opérateur paraît anodin tant il est la base de la programmation orientée objet.

Cependant chaque fois que vous faîtes un `new`, vous rendez votre code moins maintenable, moins évolutif et moins testable.

Chaque fois que vous vous apprêtez à écrire un `new` dans votre code, posez vous la question : comment pourrais-je faire pour ne pas faire de `new` à cet endroit?

L'objet de ce chapitre est de vous montrer comment faire pour utiliser le moins possible, l'opérateur `new`.

Comment remplacer l'opérateur new : propriété statique, méthode statique et chaînage de méthode

Imaginez une facture fournisseur définie de la manière suivante:

```
public class Invoice
{
    public Currency Currency { get; set; }

    //code omitted for brevity
}
```

La classe `Invoice` contient une propriété `Currency` dont le type est défini ci-dessous:

```
public class Currency
{
    public string IsoCode { get; set; }
    public string Description { get; set; }

    //code omitted for brevity
}
```

Si vous êtes dans une approche TDD, chaque test unitaire que vous allez écrire pour tester la classe `Invoice` sera de la forme:

```
//Arrange
var invoice = new Invoice();
invoice.Currency = new Currency() { IsoCode = "EUR", Description = "Euro" };

//Act
...

//Assert
...
```

ou bien de la forme:

```
//Arrange
var invoice = new Invoice();
invoice.Currency = new Currency() { IsoCode = "GBP", Description = "British Pound" };

//Act
...

//Assert
...
```

Ce code de test pose plusieurs problèmes:

- Le code de la partie `Arrange` forme un pattern qui est répété dans chaque test;
- Le code de la partie `Arrange` manque d'expressivité, c'est à dire qu'on ne comprend pas très bien pour quel usage sont crées ces objets;
- L'usage du mot clé `new` entraîne un couplage fort entre la classe de test et les classes `Invoice` et `Currency`. En particulier vous pouvez vous rendre compte que si les noms des propriétés ou si le contenu des propriétés de la classe `Currency` changent, cela va entraîner un refactoring des classes clientes et notamment des classes de test;
- Le test unitaire fait deux choses simultanément :
  - il prend la responsabilité de créer correctement tous les objets nécessaires dans la partie `Arrange` ;
  - et il prend la responsabilité de mener à bien le test dans la partie `Act` et `Assert` .

  S'il s'agit d'instancier des objets non métiers , cela pose en général aucun problème, mais s'il s'agit d'instancier des objets métiers il est vraisemblable que cela doive se faire en suivant des règles métiers prédéfinies mais susceptibles de changer à tout moment.

  L'usage de l'opérateur `new` entraîne donc les effets suivants:

  - Duplication de code;
  - Manque d'expressivité du code;
  - Fragilisation du code parce qu'il prend des responsabilités (celles de créer correctement des objets qui sont le plus souvent des objets métiers) qui ne sont pas de son ressort.

Pour vous aider à utiliser le moins souvent possible l'opérateur `new` , vous pouvez vous appuyer sur les règles suivantes:

> Toute duplication d'un pattern d'écriture de code doit être factorisé.
>
> Ne prenez jamais la responsabilité de construire par vous même un objet métier.
>
> Ne faites qu'une seule chose à la fois dans votre code mais faites le bien.

Je vais vous montrer comment utiliser ces principes pour factoriser le code de test ci-dessus.

La classe `Currency` peut être modifiée de la façon suivante:

```
public class Currency
{
    public string IsoCode { get; set; }
    public string Description { get; set; }

    public static Currency Euro
    {
        get
        {
            var result = new Currency()
            {
                IsoCode = "Euro",
                Description = "Euro"
            };
            return result;
        }
    }

    public static Currency BritishPound
    {
        get
        {
            var result = new Currency()
            {
                IsoCode = "GBP",
                Description = "British Pound"
            };
            return result;
        }
    }
    //code omitted for brevity
}
```

Grâce à cette première factorisation le premier test unitaire peut maintenir s'écrire:

```
//Arrange
var invoice = new Invoice();
invoice.Currency = Currency.Euro;

//Act
...

//Assert
...
```

Et le deuxième test unitaire peut s'écrire:

```
//Arrange
var invoice = new Invoice();
invoice.Currency = Currency.BritishPound;

//Act
...

//Assert
...
```

Vous voyez que cette première factorisation a permis :

- de passer de 2 `new` à un seul `new` pour chaque test unitaire;
- de déléguer la responsabilité de remplir correctement les propriétés `IsoCode` et `Description` à la classe même où sont définies ces propriétés.

Dans une deuxième étape de factorisation la classe `Invoice` peut être modifiée de la façon suivante:

```
public class Invoice
{
    public Currency Currency { get; set; }

    public static Invoice EmptyInvoiceInEuro
    {
        get
        {
            var result = new Invoice()
            {
                Currency = Currency.Euro
            };
            return result;
        }
    }

    public static Invoice EmptyInvoiceInBritishPound
    {
        get
        {
            var result = new Invoice()
            {
                Currency = Currency.BritishPound
            };
            return result;
        }
    }

    //code omitted for brevity
}
```

Grâce à cette deuxième factorisation le code du premier test peut maintenir s'écrire:

```
//Arrange
var invoice = Invoice.EmptyInvoiceInEuro;

//Act
...

//Assert
...
```

Grâce à cette deuxième factorisation le code du deuxième test peut maintenir s'écrire:

```
//Arrange
var invoice = Invoice.EmptyInvoiceInBritishPound;

//Act
...

//Assert
...
```

Cette deuxième étape de factorisation a permis:

- d'éliminer l'usage de l'opérateur `new` : l'action de créer correctement un objet est maintenant totalement déléguée à la classe où est défini cet objet.

Cependant cette deuxième factorisation a introduit un pattern de codage dans la classe `Invoice` entre les propriétés `EmptyInvoiceInEuro` et `EmptyInvoiceInBritishPound`. Ce pattern pourrait se nommer `EmptyInvoiceInCurrencyX` où X est le code ISO code de la devise.

Si l'application qui gère les factures ne gère que deux devises, cela ne pose aucun problème. Par contre si l'application est réellement multi-devises, l'implémentation des différentes devises, au sein de la classe `Invoice` se fera par copier/coller d'une propriété existante.

Il est donc nécessaire de réaliser une troisième étape de factorisation pour éliminer ce pattern. Pour cela il existe une méthode de codage qui est très courante en JavaScript et qui s'appelle le chaînage de méthode (Fluent API en anglais).

En appliquant cette méthode dite de chaînage, la classe `Invoice` peut être modifiée de la façon suivante:

```
public class Invoice
{
    public Currency Currency { get; set; }

    public static Invoice Empty
    {
        get
        {
            var result = new Invoice();
            //application des règles métiers
            //pour l'initialisation d'une nouvelle facture
            return result;
        }
    }

    //code omitted for brevity
}

public static class InvoiceExtensions
{
    public static Invoice SetCurrencyTo(this Invoice input, Currency  value)
    {
        if (input == null)
        {
            return input;
        }

        var oldValue = input.Currency;
        input.Currency = value;

        //application des règles métiers quand la devise change

        return input;
    }
}
```

Grâce à cette troisième factorisation le code du premier test peut maintenir s'écrire:

```
//Arrange
var invoice = Invoice.Empty
                    .SetCurrencyTo(Currency.Euro);

//Act
...

//Assert
...
```

Et le code du deuxième test peut maintenir s'écrire:

```
//Arrange
var invoice = Invoice.Empty
                     .SetCurrencyTo(Currency.BritishPound);

//Act
...

//Assert
...
```

Grâce à cette technique de chaînage de méthode, le code de la classe `Invoice` a été allégé et toutes les méthodes de chaînage sont externalisées sous la forme de méthodes d'extension.

Notez également qu'une méthode de chaînage renvoie toujours le même objet (et doit toujours renvoyer le même objet) si bien que le chaînage des appels se passe correctement y compris quand l'objet de départ est nul.

Cette technique de chaînage des méthodes a aussi renforcé l'expressivité du code. En effet, si on extrait de son contexte la ligne de code suivante:

```
var invoice = Invoice.Empty
                     .SetCurrencyTo(Currency.BritishPound);
```

On comprend parfaitement que le code réalise en séquence les étapes suivantes:

- création d'une nouvelle facture;
- affection de la devise GBP à cette facture.

La classe `Currency` contient maintenant un pattern d'écriture de code, situé à l'intérieur de chacune des propriétés statiques `Euro` et `BritishPound`. Ce pattern est le suivant:

```
var result = new Currency()
{
    IsoCode = "X",
    Description = "Y"
};
return result;
```

Ce pattern consiste à créer un objet à partir de deux autres objets : le code ISO de la devise et sa description.

Ce pattern d'écriture sera dupliqué à chaque définition d'une nouvelle devise. Il faut donc factoriser cette répétition qui n'est pas la répétition d'un ensemble de lignes de code mais qui est la répétition d'un pattern d'écriture de code.

Une première solution possible est de définir un constructeur paramétré dans la classe `Currency`. Dans ce cas, la classe `Currency` pourrait être factorisée de la manière suivante:

```
public class Currency
{
    public string IsoCode { get; set; }
    public string Description { get; set; }

    public Currency(string isocCode, string description)
    {
        this.IsoCode = IsoCode;
        this.Description = description;
    }

    public static Currency Euro
    {
        get
        {
            var result = new Currency("Euro", "Euro");
            return result;
        }
    }

    public static Currency BritishPound
    {
        get
        {
            var result = new Currency("GBP", "British Pound");
            return result;
        }
    }

}
```

Dans le code ci-dessus, il reste toujours un pattern d'écriture de code qui est le suivant:

```
var result = new Currency("XXX", "YYY");
return result;
```

Si la méthode d'instanciation d'un objet de type `Currency` doit être modifiée pour répondre à une évolution de l'application, il faudra alors réécrire toutes les méthodes qui font appel au constructeur paramétré.

L'utilisation de l'opérateur `new` doit être centralisée au sein d'une seule et unique méthode dans la classe correspondante. Cette méthode peut être codée de la manière suivante:

```
public class Currency
{
    public string IsoCode { get; set; }
    public string Description { get; set; }

    public static Currency FromIsoCode(string isoCode, string withDescription = null)
    {
        var result = new Currency();
        result.IsoCode = isoCode;
        result.Description = isoCode;

        if (withDescription == null)
        {
            return result;
        }

        result.Description = withDescription;
        return result;

    }

    public static Currency Euro
    {
        get
        {
            var result = Currency.FromIsoCode("Euro");
            return result;
        }
    }

    public static Currency BritishPound
    {
        get
        {
            var result = Currency.FromIsoCode("GBP", withDescription: "British Pound")
;
            return result;
        }
    }

}
```

Dans la classe `Currency` , l'usage de l'opérateur `new` est maintenant centralisé dans une méthode statique dont le nom commence par le terme `From` .

Ce terme indique la création d'un objet de type `Currency` à partir d'un ou de plusieurs autres objets.

La technique consistant à déclarer dans la méthode `FromXXX` un paramètre optionnel dont le nom commence par le terme `with` ou bien le terme `and` est empruntée à la méthodologie de nommage des méthodes et paramètres préconisée par Apple. Cette méthodologie rend le code encore plus expressif.

Si un test unitaire souhaite créer une devise "bidon" pour vérifier son impact dans d'autres parties de l'application, il pourra s'écrire de la façon suivante:

```
//Arrange
var deviseBidon = Currency.FromIsoCode("IGA", withDescription: "Inter Galactique");

//Act
...

//Assert
...
```

Toutes les factorisations effectuées ci-dessus ont permis de centraliser la création des instances en un seul endroit pour chacune des classes `Currency` et `Invoice`. Les clients de ces classes et notamment l'ensemble des tests unitaires associés n'ont plus besoin de prendre la responsabilité de créer par eux-même des instances valides du point de vue métier.

Vous pouvez constater que l'usage de l'opérateur `new` a également disparu dans toutes les classes clientes.

Vous pouvez aussi constater que la factorisation progressive du mot clé `new` a rendu le code plus expressif en utilisant les techniques suivantes:

- Propriété statique typiquement nommée `Empty` ;
- Chaînage de méthode (la base des APIs dites Fluent - technique principalement empruntée à JavaScript);
- Méthode statique typiquement nommée `FromXXX` ;
- Paramètres optionnels dont le nom est typiquement préfixé par `with` ou `and` (technique empruntée à Apple).

Références

Cette section contient la liste des références utilisées pour la rédaction de cette section.

Misko Every - The testability explorer blog
• How to Think About the “new” Operator with Respect to Unit Testing • Top 10 things which make your code hard to test

Apple
• Coding Guidelines for Cocoa

Google
• Guide: Writing Testable Code
To keep our code at Google in the best possible shape we provided our software engineers with these constant reminders. Now, we are happy to share them with the world.

La Loi de Déméter

Lorsqu'un objet contient une hiérarchie d'autres objets accessibles sous la forme de propriétés ou de méthodes, il arrive souvent que l'information dont on a besoin soit portée par un objet imbriqué dans cette hiérarchie comme le montre l'exemple ci-dessous:

```
var myObject = new MyClass();
var result = myObject.PropertyA.PropertyB.GiveMeTheObjectINeed();
```

Dans la classe `MyClass`, la propriété `PropertyA` est définie de la manière suivante:

```
public class MyClass
{
    public A PropertyA { get; set; }
    // code omitted for brevity
}
```

Dans la classe `A`, la propriété `PropertyB` est définie de la manière suivante:

```
public class A
{
    public B PropertyB { get; set; }
    // code omitted for brevity
}
```

Dans la classe `B`, La méthode `GiveMeTheObjectINeed` est définie de la manière suivante:

```
public class B
{
    public C GiveMeTheObjectINeed()
    {
        return new C();
    }
    // code omitted for brevity
}
public class C
{
  // code omitted for brevity
}
```

Dans l'exemple ci-dessus, la hiérarchie d'objets est définie par l'ensemble des classes suivantes:

- MyClass
  - A
    - B

Obtenir un objet par une succession d'appels en utilisant la notation `.`, comme dans l'exemple montré initialement:

```
var myObject = new MyClass();
var result = myObject.PropertyA.PropertyB.GiveMeTheObjectINeed();
```

est une technique courante chez la plupart des développeurs : pourquoi écrire plusieurs lignes de code quand on peut tout faire en une seule ligne? Pour certains développeurs le but ultime serait de pouvoir écrire l'application entière en une seule ligne de code.

Le syndrome typique de cette approche (faire un maximum de choses en une seule ligne de code) se manifeste souvent au niveau de l'instruction `return` d'une méthode comme dans l'exemple suivant:

```
public C MyMethod()
{
    var myObject = new MyClass();
    return myObject.PropertyA.PropertyB.GiveMeTheObjectINeed();
}
```

Ce type de code pose plusieurs problèmes:

- Vous introduisez un couplage fort entre votre classe et toutes les classes qui sont utilisées dans les appels successifs. Par exemple la méthode ci-dessus `MyMethod` introduit outre un couplage avec la classe MyClass, un couplage avec les classes `A` et `B`;
- Vous supposez que tous les objets obtenus ainsi au fil des appels sont valides et non nuls;
- Vous pouvez faire, sans le savoir, de la fuite de mémoire ou de la fuite de handle si les objets intermédiaires ouvrent des ressources sur le système d'exploitation, ou bien instancient en interne un objet COM;
- Quand un problème survient à l'exécution et que vous mettez un point d'arrêt sur la ligne pour l'analyser, il est souvent difficile de démêler la séquence des appels, obligeant ainsi a ré-écrire temporairement le code comme ci-dessous:

```
public C MyMethod()
{
  var myObject = new MyClass();
  var val1 = myObject.PropertyA;
  var val2 = val1.PropertyB;
  var result = val2.GiveMeTheObjectINeed();
  return result;
}
```

Pour éviter tous ces problèmes, vous devez appliquer la loi de Déméter (The Law of Demeter). L'objet de ce chapitre est de vous montrez comment appliquer cette loi.

Comment appliquer la loi de Déméter

Ce que dit la loi de Déméter est simple : soyez toujours courtois avec vos interlocuteurs. Si vous avez besoin d'une information ou d'une action qui ne peut être fournie que par une connaissance directe ou indirecte de votre interlocuteur, demandez à ce dernier de jouer les intermédiaires; ne court-circuitez jamais votre interlocuteur en communiquant directement avec ses relations.

Autrement dit, la loi de Déméter consiste à ne parler qu'avec ses voisins directs.

Chaque fois que vous écrivez du code qui consiste à chaîner des appels sur des objets différents via leurs propriétés ou leur méthodes en formant une séquence similaire à l'exemple ci-dessous:

```
var myObject = new MyClass();
var result = myObject.PropertyA.PropertyB.GiveMeTheObjectINeed();
```

cherchez à appliquer la loi de Déméter.

En choisissant de ne pas appliquer la loi de Déméter, vous diminuez la maintenabilité de votre application car toute modification d'un objet intermédiaire aura un impact sur votre code. De plus vous introduisez un couplage entre votre application et les classes sous-jacentes appelées au travers du chaînage des appels. Il est fort probable que ce couplage n'est jamais été prévu au départ de la conception de ces classes.

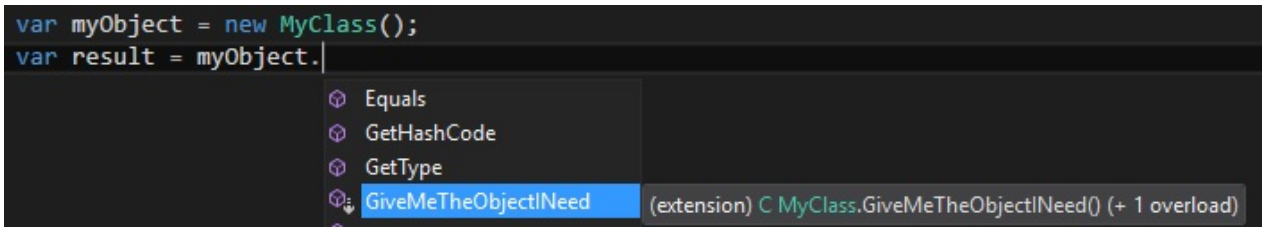
Pour appliquer la loi de Déméter, vous devez "remonter" le dernier appel directement sur le premier objet utilisé dans la chaîne des appels. Ainsi l’exemple de code ci-dessus doit être écrit de la manière suivante:

```
var myObject = new MyClass();
var result = myObject.GiveMeTheObjectINeed();
```

Bien évidemment cette méthode ou cette propriété n'est pas disponible car sinon vous l'auriez utilisée. En utilisant le principe des méthodes d'extension, vous pouvez créer cette méthode qui vous manque de la façon suivante:

```
public static class MyClassExtensions
{
    public static C GiveMeTheObjectINeed(this MyClass input)
    {
        if (input == null)
        {
            return null;
        }

        var firstObject = input.PropertyA;
        if (firstObject == null)
        {
            return null;
        }

        var secondObject = firstObject.PropertyB;
        if (secondObject == null)
        {
            return null;
        }

        var result = secondObject.GiveMeTheObjectINeed();
        return result;
    }
}
```

L'IntelliSense vous indique désormais que la méthode dont vous avez besoin est maintenant disponible directement sur votre "voisin":



La méthode d'extension mise en place permet de gérer les états possibles des objets intermédiaires et notamment le cas où ils sont nuls. Vous avez ainsi augmenté la maintenabilité de l'application.

Le code mis en place dans la méthode d'extension permet aussi de mettre en évidence si les objets intermédiaires sont susceptibles d'ouvrir des ressources sur le système d'exploitation. En effet si vous détectez la présence d'une méthode `Dispose()` ou `Close()` sur ces objets, vous devez impérativement appeler l'une ou l'autre avant de sortir de la méthode.

Synthèse des règles spécifiques au AAA Programming

Ce chapitre regroupe l'ensemble des règles spécifiques au AAA Programming.

Je vous invite à les mettre en pratique. Commencez par la règle qui vous semble la plus facile à appliquer, ou bien la plus appropriée dans votre contexte. Dans tous les cas de figure vous allez vous rendre compte qu'appliquer ces règles va automatiquement enclencher une série de questions et d'échanges avec les autres développeurs. De ces questions et de ces échanges sortira une meilleure architecture, une meilleure implémentation, une meilleure cohésion d'équipe.

Règle n° 1
La convention de nom Camel Casing doit être utilisée pour nommer les éléments suivants: • Variable locale; • Paramètre de méthode.
Voir les chapitres Camel Casing et Quand utiliser Camel Casing

Règle n° 2
La convention de nom Pascal Casing doit être utilisée pour nommer les éléments suivants: • Classe; • Énumération; • Valeur d'une énumération; • Événement; • Membre public static et read-only; • Méthode; • Propriété; • Namespace; • Interface.
Voir les chapitres Pascal Casing et Quand utiliser Pascal Casing

Règle n° 3
Quand une variable locale est associée à une propriété, considérez le fait de préfixer son nom par le caractère _.
Voir le chapitre Cas particulier des variables qui sont associées à une propriété

Règle n° 4

Toute variable doit être déclarée au plus près de son utilisation.

Règle n° 5

Toute variable associée à une propriété doit être utilisée exclusivement au sein du getter ou du setter de cette propriété.

Règle n° 6

Ne jamais utiliser l'opérateur de négation `!`

= pensez toujours positif.

Règle n° 7

Un `if` ne contient jamais de `if` imbriqué.

Règle n° 8

Toute ligne de code (ou ensemble de lignes) dupliquée doit être factorisée.

Règle n° 9

La dernière instruction d'un bloc de code `If` est toujours `return` ou `continue`.

Règle n° 10

Un `If` n'a jamais de `else`.

Règle n° 11

Quand vous commencez à coder un `If`, utilisez toujours le code snippet associé en tapant `if` puis `TAB` deux fois.

Remarque : cette astuce vous rappelle qu'il faut toujours penser positif.

Règle n° 12

Quand vous documentez une méthode booléenne, la balise `<returns>...</returns>` dans le commentaire XML, commence toujours par `Returns true if ...` ou `Returns true when ...`

Règle n° 13

Quand vous développez une méthode, vous devez en sortir le plus vite possible. Autrement dit si vous pouvez déterminer un cas de figure qui permet de faire immédiatement un `return` écrivez d'abord ce cas.

Règle n° 14

La dernière instruction d'une méthode booléenne est toujours: `return false;`

Règle n° 15

Le bloc de code qui précède la dernière ligne de code d'une méthode booléenne est toujours de la forme:

```
if ( A )
{
   //code omitted for brevity
   return true;
}
```

Règle n° 16

Si vous êtes amené à documenter du code à l'intérieur d'une méthode ou d'une propriété, considérez le fait de remplacer ce code par une méthode d'extension ayant un nom suffisamment évocateur et de déplacer votre commentaire dans la documentation XML associée à la méthode d'extension.

Règle n° 17

Quand vous développez une boucle `for` ou `foreach`, vous devez en sortir le plus vite possible. Autrement dit si vous pouvez déterminer un cas de figure qui permet de passer immédiatement à l'itération suivante ,écrivez d'abord ce cas. Si vous pouvez déterminer un cas de figure qui permet de sortir de la boucle, écrivez d'abord ce cas.

Règle n° 18

Une boucle while est toujours de la forme:

```
while ( true )
{
   //code omitted for brevity
}
```

Règle n° 19

Quand vous écrivez un `new` dans votre code, posez vous la question : comment pourrais-je faire pour ne pas faire de `new` à cet endroit?

Règle n° 20
Tout duplication d'un pattern d'écriture de code doit être factorisé.

A compléter