Jacques Bourgeois – Formation Informatique Inc.

Visual Basic.NET
Interaction avec Excel

Notes de cours – Supplément Excel

Version 1.0

Avertissement

Ce document a été développé pour un cours privé et ne sera probablement utilisé qu'une seule fois. Nous n'avons pas eu le temps d'en faire une révision exhaustive comme c'est le cas pour les manuels que nous utilisons dans nos cours réguliers.

Nous avons fait tout notre possible pour que les informations qu'il contient soient correctes, claires et en bon français, mais étant donné le temps disponible et le cadre de son utilisation, nous ne pouvons nous porter garant de l'intégrité de son contenu.

Important Veuillez donc considérer ce document comme un guide de départ, et ne pas nécessairement prendre au pied de la lettre tout ce qui y est dit.

Présentation

Notre discussion porte sur Microsoft Excel, mais les concepts de bases sont les mêmes pour tous les logiciels Office. Jusqu'à la page 17, vous pouvez adapter la discussion presque directement à tous les autres logiciels de la suite en remplaçant par exemple le mot Excel par Word et classeur par document.

Ce document est une extension du cours Visual Basic .NET 2010 – Introduction donné par Jacques Bourgeois. Il a été développé pour les besoins spécifiques d'un cours privé et se veut une simple introduction aux concepts de base de l'utilisation de Visual Basic .NET avec Microsoft Excel.

Nous assumons que les utilisateurs de ce document ont déjà une connaissance de base en Visual Basic .NET et, idéalement, qu'ils sont familiers avec l'utilisation de VBA dans Excel. Étant données les limites de temps et de ressources pour le développement d'un cours privé, il s'agit d'un document sommaire qui ne couvre pas la matière en détail.

Public cible

Ce document a été conçu pour des ingénieurs et chercheurs qui ont besoin de développer des applications simples pour supporter leur travail. Il ne s'adresse donc pas, à priori, à des programmeurs professionnels. Ces derniers y trouveront quand même, au besoin, des informations de base sur les techniques d'accès à Excel à partir de Visual Basic .NET.

Exemples de code

Les exemples de code développés spécifiquement pour ce cours se retrouvent sur le DVD qui vous a été fourni au début du cours. Vous trouverez les instructions d'installation des exemples au début du manuel de cours standard.

Le sous-répertoire Exemples supplémentaires\VB_Excel contient les exemples de code pertinents pour cette portion du cours.

Versions des logiciels

Ce document ayant été créé pour un client précis, il est adapté à ses besoins et ne couvre par un éventail de versions des logiciels comme le font nos manuels standards.

Nous avons spécifiquement visé Visual Studio 2010 et Visual Basic 10, utilisés avec le framework 4.0 et Microsoft Excel 2003.

Nous avons développé nos exemples avec l'édition Premium de Visual Studio 2010, mais ils devraient fonctionner tels quels dans l'édition Professional. Il est fort probable qu'ils soient aussi compatibles avec l'édition Express.

Bien que les techniques de programmation soient similaires entre les différentes versions d'Excel et des outils .NET, il est fort probable que certains aspects abordés aient besoin d'être ajustés avec une combinaison différente des logiciels.

Les deux environnements (.NET vs Office) sont de natures différentes, et Excel utilise depuis la version 2007 un format de document complètement différent. La compatibilité entre les différentes versions des logiciels en est d'autant plus sensible.

Important

- Dans les exemples de code présentés dans ce manuel et sur le DVD d'exemples fourni avec le cours, nous assumons que le namespace *Microsoft.Office.Interop* a été activé dans les *Imports*, comme décrit à la page 10.
- L'application d'exemple utilise un document Excel dont la version originale est localisée par défaut dans C:\Cours VS2010_\Exemples supplémentaires\VB_Excel\InventaireBois.xls. L'application n'utilise pas directement ce fichier, elle pointe plutôt à des copies situées dans les répertoires de compilation. L'original peut donc être récupéré si jamais les copies utilisées deviennent corrompues à cause de bogues durant le développement. Nous fonctionnons ainsi pour faciliter l'installation des exemples de cours dans différents environnements. **Ce n'est pas une approche à utiliser dans une « vraie » application**, parce que le répertoire Program Files habituellement utilisé pour installer les applications est en lecture seule pour les utilisateurs qui ne sont pas administrateur de leur station. Dans la vraie vie, vous avez un meilleur contrôle de l'installation et des chemins d'accès et pourrez donc installer les fichiers Excel là où ils sont facilement accessibles par l'application et l'utilisateur

.NET avec COM ➔ Framework + Interop

Comme nous l'avons vu dans la portion VB.NET du cours, le framework doit être préalablement installé sur les stations des utilisateurs qui voudront utiliser vos applications .NET. Excel doit aussi être installé, étant donné que les applications .NET ne font rien d'autre qu'automatiser les opérations effectuées par Excel.

Microsoft Excel, dans sa forme actuelle, a été développé au début des années 90 et est basé sur une technologie appelée COM (Component Object Model) ou ActiveX. Cette technologie spécifie des normes qui permettent à des applications des communiquer les unes avec les autres. Le Visual Basic « classique »[1], utilisé pour programmer respectivement des macros (VBA) et applications (VB6) avec Excel, utilise cette technologie.

Avec les avancées technologiques, l'arrivée de l'Internet, les besoins accrus de sécurité et d'interactivité entre différents environnements, le modèle COM est devenu trop limité. Microsoft a donc développé au début des années 2000 un nouveau modèle appelé .NET. Comme son nom l'indique, Visual Basic .NET utilise ce modèle.

COM et .NET sont deux environnements très différents, qui ne fonctionnent pas sous les mêmes principes. La mémoire de l'ordinateur n'y est pas gérée de la même façon, et certains des types de variables qu'ils utilisent sont différents, entre autres les dates qui ne sont pas enregistrées dans le même format.

Fondamentalement, COM et .NET sont incompatibles et ne peuvent pas se parler. Vous aurez donc besoin d'un « traducteur » pour permettre à une application .NET de pouvoir « parler » à Excel. Cet intermédiaire est communément appelé un interop. Dans le cas des logiciels Office, on a développé un interop spécial appelé un *Primary Interop Assemblies* (PIA).

Important Sans ce PIA Visual Basic .NET et Excel ne peuvent pas communiquer ensemble. Il est donc primordial qu'il soit installé sur les stations de développement.

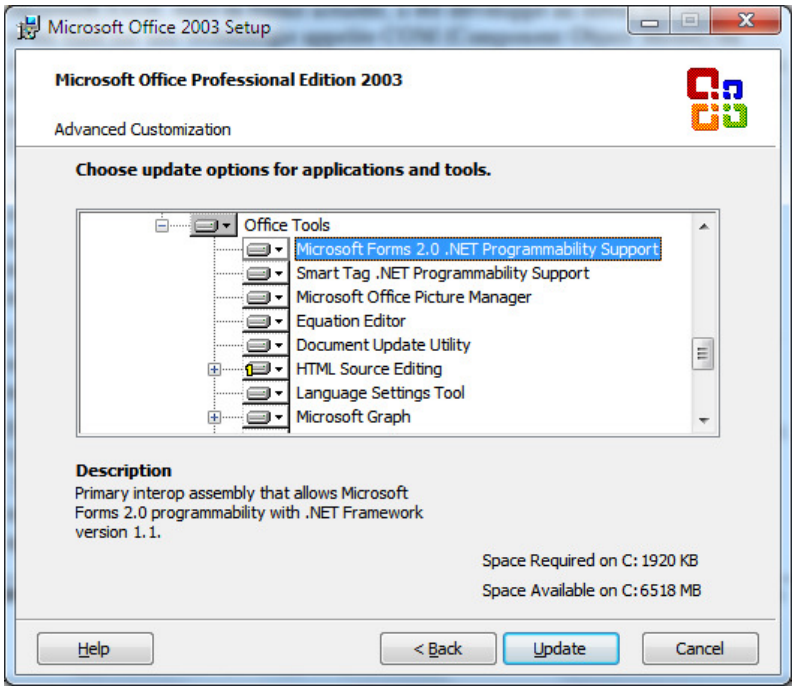
Pour savoir s'il est installé sur votre station, vous pouvez jeter un coup d'œil au répertoire C:\Windows\assembly. Les fichiers dll composant le PIA sont clairement identifiés avec le libellé Microsoft.Office.Interop.

S'il n'est pas là, le PIA pour Office 2003 peut être téléchargé du site Web de Microsoft à http://www.microsoft.com/downloads/en/details.aspx?familyid=3c9a983a-ac14-4125-8ba0-d36d67e0f4ad&displaylang=en.[2]

[1] Par opposition à Visual Basic .NET.

[2] Vous trouverez une copie de ce lien dans le document Lien.txt du dossier d'exemples.

Il peut aussi être installé à partir de l'installation standard d'Office en activant l'option .NET dans la portion Office Tools de l'installation :



Programmer Excel : plusieurs approches

Il existe plusieurs façons de programmer pour Excel.

Une application VBA

La façon classique de programmer dans Office, avec des macros. Si vous suivez ce cours, c'est que vous désirez vous familiariser avec une autre approche. Nous ne couvrirons donc pas cette approche, mais comme vous le verrez plus loin, elle reste utile si vous avez besoin de performance.

Un projet Excel

Un projet Excel est un dll .NET qui est appelé par la feuille Excel au besoin. C'est le pendant .NET des add-ins VBA dans un document Excel.

Contrairement à une application VBA conventionnelle, où une copie du code est répétée dans chaque document, un add-in ou un dll .NET est partagé par plusieurs classeurs.

Le gros avantage de cette approche est qu'elle simplifie considérablement les mises à jour. Mettre à jour le fichier contenant le code met automatiquement à jour tous les classeurs Excel qui l'utilisent.

Le principal désavantage est que le code étant distribué séparément du document Excel, le classeur perd sa fonctionnalité s'il ne trouve pas le fichier contenant le code.

En ce qui concerne .NET, il y a aussi un désavantage majeur : la compatibilité entre les versions des différents logiciels. Un dll .NET conçu pour Excel 2003 doit être recompilé et parfois modifié pour fonctionner avec Excel 2007, ce qui n'est habituellement pas le cas pour VBA.

Encore pire, dans votre cas particulier. Visual Studio 2010 ne permet pas de développer des dll compatibles avec Office 2003. Si vous tentez de créer un projet Office par *File►New...Project...Visual Basic...Office*, vous allez constater que seuls les projets pour Office 2007 et Office 2010 sont offerts. Vous devez avoir une version antérieure de Visual Studio pour pouvoir développer des projets pour Office 2003.

Comme ce n'est pas votre cas, nous ne couvrirons pas ce type de projet dans le cours.

Une application .NET (.exe) contenant tout le code

Vous pouvez écrire un programme (application) en .NET qui manipule des classeurs et des feuilles Excel. Cette application ne fait qu'utiliser Excel avec du code .NET à la place du VBA traditionnel.

Un des avantages de cette approche est qu'elle permet d'utiliser n'importe quel langage de programmation disponible sur .NET; il y en a une quarantaine. Mais parce qu'Excel a été au départ conçu pour être programmé à partir de Visual Basic, la plupart des programmeurs vont utiliser Visual Basic .NET plutôt qu'un autre langage.

L'intérêt de programmer en VB.NET plutôt qu'en VBA est que VB.NET est un langage beaucoup plus puissant. Il a par ailleurs été développé au début des années 2000 plutôt qu'au début des années 1990, et est donc plus adapté aux besoins modernes des entreprises.

L'environnement .NET permet entre autres, pour certaines applications, le développement d'interfaces différentes et beaucoup plus sophistiquées que celles qui sont utilisées dans Excel, facilitant beaucoup entre autres la validation de la saisie des utilisateurs, quelque chose de possible mais fastidieux en VBA.

Comme à peu près tout en informatique cependant, quand on gagne à quelque part, il y a aussi des désavantages.

Si vous envoyez un classeur développé de façon conventionnelle à quelqu'un par courriel par exemple, le code VBA est enregistré dans le classeur. Les fonctionnalités que vous y avez définies sont disponibles pour quiconque a accès au document Excel. Dans le cas d'un document manipulé par une application VB.NET, le récipiendaire du courriel a accès aux données, mais pas à la fonctionnalité à moins qu'il n'installe chez lui le logiciel développé en VB.NET.

Un autre désavantage, qui peut-être important dans les classeurs où on traite une grande quantité de données par le code, c'est la performance qui est réduite de façon considérable. Le code VBA roule à l'interne dans Excel et a un lien direct avec l'information contenue dans le classeur. Il utilise par ailleurs les mêmes formats pour la plupart des types de données.

Par contre, le code .NET doit communiquer avec Excel qu'il voit comme une application externe. En plus, .NET et le vieux monde COM dont fait partie Excel n'utilisent pas les mêmes conventions de communication et n'ont pas les mêmes formats de données[3]. Une application .NET doit donc passer par un intermédiaire qui servira de traducteur entre les modes COM et .NET, la notion d'interop dont nous avons parlé à la page 5.

Le fait d'avoir à passer par un intermédiaire et de devoir convertir certains types de données pénalise beaucoup les applications .NET / Excel au niveau de la performance. D'où l'approche qui suit, qui est souvent la plus intéressante quand on a une grande quantité de données à traiter en « batch ».

Une application .NET (.exe) qui active du code VBA dans un document

Ici, .NET sert d'interface. Il permet d'afficher les données dans un environnement qui peut sous bien des aspects être plus intéressant que celui d'Excel tout en facilitant certaines opérations comme la validation les données à la saisie. Il est très facile, dans Excel, de taper par erreur un 32 décembre ou d'ajouter 900 ans à une date parce que notre doigt a glissé sur la touche à côté du zéro : 2 mars 2911 au lieu de 2 mars 2011. En .NET, comme le programmeur peut intervenir plus facilement sur une zone de saisie, on peut s'assurer qu'une date est une date valide et qu'elle se situe dans une plage de valeurs adéquate pour le besoin.

L'application .NET peut aussi faire un certain travail de traitement.

Mais si on s'aperçoit en cours de développement qu'une fonctionnalité qui intervient sur un très grand nombre de données est trop lente, on va tout simplement déménager le code .NET dans une macro VBA, et appeler cette macro à partir de notre code .NET, comme nous apprendrons à le faire à la page 23.

Connexion entre VB.NET et Excel

Définir une référence

Nous avons déjà étudié, dans le cours de base, comment créer un nouveau projet et définir des références (voir l'entrée *Références* dans l'index à la fin du manuel d'introduction).

Pour pouvoir utiliser Excel à partir de .NET, vous devrez faire une référence à Microsoft Excel dans l'onglet COM de la fenêtre d'ajout de références

---

[3] Par exemple, dans Excel, les dates sont enregistrées comme un nombre avec décimales qui représente le nombre de jours et d'heures depuis le 31 décembre 1899 à minuit. En .NET, les dates sont enregistrées comme un nombre entier représentant le nombre de ticks (1 tick = 1000 nanosecondes) depuis le 1 janvier 0001. Par exemple, le 2 mars 2011 à midi est enregistré comme étant 40604.5 en VBA, mais 634346640000000000 en VB.NET.

(*Project►<Project> Properties…References...Com...Microsoft Excel 11.0 Object Library*). Si plusieurs versions d'Excel sont présentées, utilisez la plus haute[4].

À partir de ce point, Visual Studio sait quels fichiers dll utiliser pour se connecter à Excel, et est capable d'afficher correctement les noms des classes, propriétés et méthodes dans les listes IntelliSense qui vous aident durant le développement.

Inclure le PIA dans l'application

Nous avons parlé à la page 5 du besoin de passer par un intermédiaire entre le monde .NET et le monde COM dont fait partie Excel.

Dans Visual Studio 2010, par défaut, les portions pertinentes du PIA seront inclues dans votre application. Cela évite d'avoir à installer le PIA chez les utilisateurs. Pour vous en assurer, allez dans l'onglet *References* des propriétés de projet où l'activation de la référence a inclus les 3 dll nécessaires pour pouvoir travailler avec Excel :

- Microsoft Excel 11.0 Object Library [5]
- Microsoft Office 1.0 Object Library
- Microsoft Visual Basic for Applications Extensibilité 5.3

En cliquant individuellement sur ces trois entrées, la propriété *Embed Interop Types* devrait être à True.

Namespace Microsoft.Office.Interop.Excel

L'activation de la référence dans votre environnement de développement a aussi amené un nouveau namespace *Microsoft.Office.Interop.Excel*, que vous devrez utiliser chaque fois que vous voudrez « parler » à Excel à partir de votre code VB.NET.

Il peut être fastidieux d'avoir à taper ce namespace chaque fois qu'on veut initialiser un objet :

```
Dim application As Microsoft.Office.Interop.Excel.Application
Dim classeur As Microsoft.Office.Interop.Excel.Workbook
Dim feuille As Microsoft.Office.Interop.Excel.Worksheet
Dim cellule As Microsoft.Office.Interop.Excel.Range
```

Pour éviter cette redondance fastidieuse, vous pouvez activer un *Imports* sur le namespace, dans le bas de la fenêtre de références, ce qui permettrait par exemple de pouvoir le code précédent à :

```
Dim application As Application
Dim classeur As Workbook
Dim feuille As Worksheet
Dim cellule As Range
```

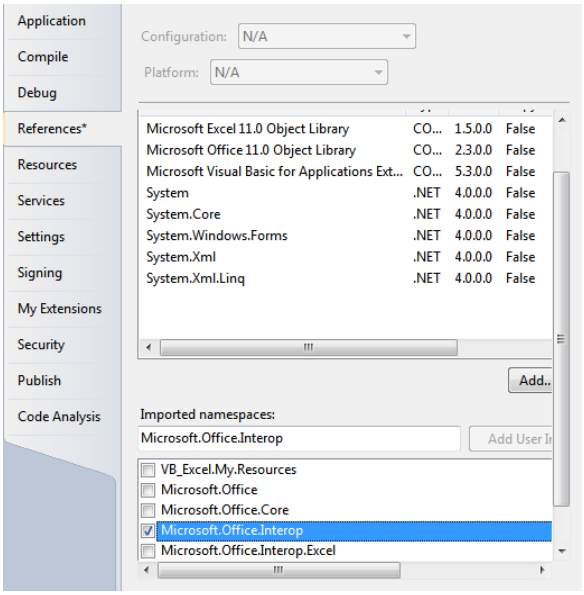
---

[4] La version 5.0 est entre autres souvent offerte en plus de la plus récente. Elle sert à pouvoir communiquer avec les classeurs créés dans les anciennes versions d'Excel (Excel 5 ou moins) et n'est pas nécessaire pour les classeurs créés depuis Excel 95.

[5] Les numéros de version seront différents avec des versions d'Office autres que 2003.

Nous vous suggérons cependant fortement de limiter le *Imports* à la portion *Microsoft.Office.Interop* du namespace, ce qui forcera alors à ajouter le libellé Excel dans vos déclarations :

```
Dim application As Excel.Application
Dim classeur As Excel.Workbook
Dim feuille As Excel.Worksheet
Dim cellule As Excel.Range
```



Cette façon de faire a deux avantages. Dans un premier temps, il est plus facile de faire la distinction entre ce qui vient d'Excel et ce qui vient de .NET. Ensuite, ça vous permettra d'éviter des conflits si jamais vous voulez aussi programmer avec d'autres logiciels de la suite Office, qui utilisent souvent les mêmes noms de classe pour des concepts semblables.

Ainsi, ce code ne peut pas compiler…

```
Dim appExcel As Application
Dim appWord As Application
Dim appAccess As Application
Dim appPowerPoint As Application
```

… tandis que le code suivant est acceptable :

```
Dim appExcel As Excel.Application
Dim appWord As Word.Application
Dim appAccess As Access.Application
Dim appPowerPoint As PowerPoint.Application
```

VB.NET vs VBA, un aperçu rapide

Quand vous travaillez dans Excel en VBA, plusieurs choses se font automatiquement sans que vous ayez à vous en occuper :

- Excel roule déjà.
- Le document est déjà chargé dans Excel.
- Par défaut, VBA assume que le code est exécuté dans le classeur et la feuille qui sont affichés à l'écran pour l'utilisateur.

Tout ça peut sembler superflu, mais en VB.NET, rien de tout ça ne vient tout seul. Il faut lancer Excel et y charger un document, puis ensuite, à chaque opération, spécifier à quelle classeur/feuille l'opération s'applique. Excel n'étant pas visible par défaut quand on l'active avec du code .NET, le système ne peut assumer que la page affichée est la page par défaut.

Le code suivant en VBA …

```
Cells(5,5).Value = 5
```

… devient à priori ceci en VB.NET :

```
Dim app As Excel.Application

app = New Excel.Application      'Lance une copie invisible d'Excel
app.Workbooks.Open("C:\Classeurs\Mon classeur.xls")
app.Workbooks(1).Worksheets(1).Cells(5, 5).Value = 5
```

On comprend peut-être plus facilement si on sait que le code VBA est presque toujours du code abrégé qui fait appel à certaines valeurs par défaut. Ainsi, les deux lignes suivantes font la même chose en VBA :

```
Application.ActiveWorkbook.ActiveSheet.Cells(5,5).Value = 5

Cells(5,5).Value=5
```

VBA permet donc des raccourcis qui ne sont pas là en .NET, où l'application, le classeur actif et la feuille active ne sont jamais définis par défaut et doivent toujours être implicitement spécifiés.

Voici ce qui se passe dans l'exemple de code .NET ci-dessus :

- *app* représente une copie[6] d'Excel, qui est lancée automatiquement quand nous appelons `New Excel.Application`.
- L'objet *Application* contient une collection classeurs, un groupe d'objets *Workbook*, appelés naturellement dans le code avec le nom anglais, soit *Workbooks*. Notez que le singulier représente un classeur unique, tandis que le nom au pluriel est la collection représentant tous les classeurs actuellement ouverts dans l'application. Même s'il n'y en a qu'un seul, la collection *Workbooks* est le mécanisme par lequel vous accédez au classeur.
- Pour ouvrir un classeur, vous appelez la méthode *Open* de la collection, ce qui ouvre le fichier spécifié dans Excel et l'ajoute à la collection.
- *Workbooks(1)* permet d'accéder au premier classeur ouvert.
- Un principe similaire s'applique à la relation entre le classeur et les feuilles qu'il contient. Un *Workbook* contient une collection de feuilles, un groupe d'objets individuels *Worksheet* regroupés dans une collection *Worksheets* (le pluriel encore pour la collection). *Workbooks(1).Worksheets(1)* permet donc d'accéder à la première feuille du premier classeur ouvert.

[6] La documentation et beaucoup de programmeurs diraient une **instance** d'Excel.

To Set or not to Set

L'assignation à des variables objet en VBA nécessite l'utilisation d'un Set. Le Set n'est plus utilisé en .NET. Si vous mettez un Set devant une assignation, l'environnement de développement va automatiquement l'enlever.

L'application

Comme indiqué plus tôt, vous avez besoin d'un objet *Application* pour pouvoir lancer Excel et accéder à ses fonctionnalités. Une application .NET / Excel va donc toujours avoir du code similaire à :

```
Dim app As Excel.Application
app = New Excel.Application
```

Beaucoup de programmeurs vont combiner la déclaration de la variable et l'appel du constructeur en une seule ligne :

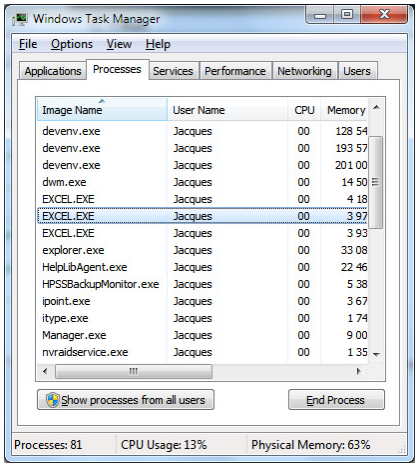
```
Dim app As New Excel.Application
```

Deux choses distinguent Excel lancé par le code d'Excel lancé par un utilisateur :

- Excel est lancé par le *New*, mais reste invisible.
- Excel ne crée pas automatiquement un classeur vide.

C'est bien commode, parce qu'une application VB.NET utilisant Excel peut ouvrir ou créer, modifier et enregistrer un document Excel sans que l'utilisateur en soit conscient. Si Excel n'est pas visible, vous augmentez aussi considérablement la performance, parce qu'il a besoin de moins de mémoire pour fonctionner et ne perd pas son temps à rafraîchir l'écran à chaque petit changement.

Vous ne voyez donc pas Excel à l'écran et il n'est pas affiché dans la liste d'applications du Gestionnaire de tâches de Windows (*Task Manager*) activable par CTRL-ALT-SUPPR. Vous pouvez cependant déterminer qu'une copie invisible roule en jetant un coup d'œil aux processus… avec possiblement des petits problèmes comme ici à droite.



À chaque fois que notre application appelle New, une nouvelle copie d'Excel est lancée.

Pendant le développement, notre application a « planté » à quelques reprises. La copie invisible d'Excel n'a pas été fermée. Comme elle n'est pas visible à l'écran et ne s'affiche pas à l'écran, on ne s'en rend pas compte.

Non seulement cela peut-il causer des problèmes de mémoire dans l'environnement de Windows, mais il est fort possible qu'une de ces copies aie verrouillé le document sur lequel vous travaillez. Vous relancez votre application dans Visual Studio, et elle n'est plus capable d'ouvrir le document, parce qu'il est verrouillé.

Important Prenez l'habitude d'aller voir régulièrement dans la liste des processus pendant le développement d'une application Excel, et de fermer manuellement tous les processus superflus découlant de copies d'Excel lancées par votre application sans avoir été fermées correctement.

Ceci implique que vous devez prendre soin de ces choses dans votre programme VB.NET.

Typiquement, voici ce que vous devriez faire.

```
Dim app As New Excel.Application

app.Visible = True  'Optionnel
'On travaille
app.Quit()
```

Naturellement, ce code peut être éparpillé dans plusieurs méthodes, dépendant de l'application. Dans un formulaire par exemple, il est courant d'appeler le *New* dans l'événement *Load*, et de faire le *Quit* dans l'événement *FormClosing*.

Application visible

Vous n'êtes pas obligé de rendre l'application *Visible*, mais ça permet à l'utilisateur de voir le résultat sans avoir à lui-même avoir à lancer Excel et charger le document une fois les opérations terminées.

Important Quand Excel est *Visible*, la commande *Quit* dans le code ne ferme par l'application. L'utilisateur devra fermer Excel lui-même. Il faut quand même l'utiliser, sinon la copie d'Excel risque de continuer à rouler en tant que processus après que l'utilisateur l'aura fermée.

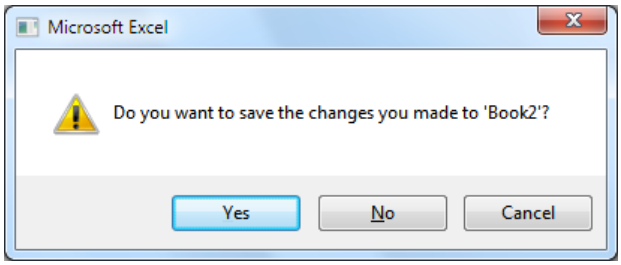
Rendre Excel visible peut aussi causer des problèmes. Le code qui manipule Excel est beaucoup plus lent quand Excel est visible. Si vous faites un grand nombre d'opérations, ça va se sentir.

En plus, il y a des chances que l'utilisateur essaie de travailler dans une feuille visible en même temps que vous travaillez dessus avec votre code, ce qui peut causer des conflits qui causeront des problèmes dans les données, quand ils ne feront pas tout simplement sauter votre application.

Dans certains cas, on va lancer Excel invisible, faire le travail nécessaire par le code, et rendre l'objet *Application* visible uniquement quand on a terminé pour que l'utilisateur puisse voir les résultats.

Application invisible

Si vous laissez l'application invisible, parce que le rôle de votre application est simplement de créer un document Excel sans intervention de l'utilisateur, assurez-vous d'enregistrer le fichier avant de terminer. Rien n'est plus surprenant pour l'utilisateur que de recevoir un écran comme celui-ci quand, pour lui, Excel ne roule pas :



Pour éviter ce message impromptu, assurez-vous de bien fermer tous les documents ouverts dans une application qui n'affiche pas Excel :

```
App.Workbooks(1).Save()
App.Workbooks(2).Save()
App.Workbooks(3).SaveAs("C:\Classeurs\Toto.xls")
app.Quit()
```

Le *Save* est utile quand votre code a ouvert un document existant et l'a modifié. *SaveAs* avec un nom de fichier est essentiel si vous avez créé le document de toutes pièces dans votre application .NET ou si vous voulez conserver une copie de l'original. Si vous faites un simple *Save* ou un *SaveAs* sans nom de fichier, l'utilisateur se fera demander par Excel de fournir un nom de fichier, même si Excel roule uniquement en tant que processus et est invisible.

Important Idéalement, vous devriez en plus très bien gérer les erreurs dans votre application pour vous assurer de fermer correctement la copie d'Excel en cas de pépin.

⊙ Notre application d'exemple montre une façon de s'organiser pour s'assurer qu'Excel sera fermé correctement en cas de problèmes. Le *Try…Catch* de la procédure *ModuleExcel.Main* s'occupe de trapper tout erreur inattendue et qui n'est pas traitée localement dans le code.

Le problème des copies multiples et non visibles d'Excel roulant en arrière-plan se produit particulièrement quand vous programmez, alors que l'application va souvent « planter » avant que la commande *Quit* ne soit appelée. Il est donc recommandé de travailler avec le gestionnaire de tâches ouvert pendant le développement, et d'y jeter un coup d'œil régulièrement pour fermer manuellement les copies superflues.

Si le problème des processus multiples survient après le développement, alors que l'application est en cours d'utilisation réelle, vérifiez tout d'abord votre code pour vous assurez que chaque *New Application* soit fermé par un *Quit*. Et si l'application a tendance à générer des erreurs, assurez-vous aussi de traiter les exceptions à l'intérieur d'un *Try…Catch* et d'appeler le *Quit* dans le traitement de l'exception.

- La méthode *Affiche* du formulaire *FormExcel* démontre comment traiter localement les erreurs.

  Si après tout ça, vous constatez que dans certaines circonstances, malgré un *Quit* bien placé, la copie d'Excel reste toujours « accrochée » dans les processus, faites une recherche sur l'Internet pour la commande *ReleaseComObject*. Dans certaines circonstances, vous devez utiliser cette commande sur toutes les variables objet utilisées par votre application pour complètement relâcher la copie d'Excel. Malheureusement, cette commande peut être fastidieuse et complexe à utiliser.

Le classeur (Workbook)

Ouvrir un classeur

Nous avons déjà utilisé ce code dans un exemple précédent. Une fois l'objet *Application* initialisé, vous ouvrez un classeur déjà existant en appelant la méthode *Open* de la collection de *Workbooks* :

```
Dim app As Excel.Application

app = New Excel.Application
app.Workbooks.Open("C:\Classeurs\Mon classeur.xls")
app.Workbooks(1).Worksheets(1).Cells(5, 5).Value = 5
```

Créer un nouveau classeur vide

Encore une fois, vous devez tout d'abord besoin de lancer Excel pour avoir accès à sa collection de *Workbooks*. Il suffit ensuite d'appeler la méthode *Add* du classeur pour créer un nouveau classeur.

```
Dim app As Excel.Application

app = New Excel.Application
app.Workbooks.Add()
app.Workbooks(1).Worksheets(1).Cells(5, 5).Value = 5
```

Si le nouveau classeur est basé sur un modèle (*template*), passez tout simplement les coordonnées du modèle à la méthode *Add* :

```
app.Workbooks.Add("C:\Modèles\Mon modèle.xlt")
```

Référencer le classeur

Vous pouvez travailler avec le classeur de plusieurs façons.

En l'appelant par son indice comme nous l'avons fait jusqu'ici :

```
app.Workbooks(1)
```

C'est la méthode la plus simple et la plus efficace dans une application qui ne travaille que sur un seul classeur. Mais si une application utilise plusieurs classeurs différents, le code devient plus difficile à lire si on a :

```
app.Workbooks(1)
app.Workbooks(2)
app.Workbooks(3)
```

C'est encore pire si les classeurs ne sont pas toujours ouverts dans la même séquence, ce qui peut arriver dans certaines applications où l'utilisateur a le contrôle du flot de travail. Il peut aussi devenir difficile de « suivre » un classeur. Si vous fermez le classeur 2 par exemple, le 3 devient 2. Si vous insérez un classeur devant le 3, le 3 devient 4.

Dans ce cas, il peut devenir plus intéressant de spécifier le classeur par son nom, soit le nom du fichier, sans le chemin d'accès :

```
App.Workbooks("Toto.xls")
```

Ceci n'est cependant pas possible si vous ouvrez plusieurs fichiers du même nom provenant de répertoires différents.

Ça peut aussi ne pas fonctionner correctement dans des environnements ou les utilisateurs n'ont pas tous une version d'Excel dans la même langue, ce dont nous discutons plus loin. Dans un tel cas, un tout nouveau classeur créé dans la version anglaise s'appellera *Woorkbook1* tandis qu'un autre, créé avec le même code s'appellera *Classeur1*.

Pour éviter tous les problèmes énumérés ci-dessus, un grand nombre de programmeurs vont tout simplement assigner chaque classeur à une variable. Ainsi, le code devient indépendant de l'ordre dans lequel les classeurs sont ouverts, ainsi que de leur nom :

```
Dim app As New Excel.Application
Dim classeurTintin As Excel.Workbook
Dim classeurMilou As Excel.Workbook
Dim classeurTournesol As Excel.Workbook

app.Workbooks.Add()
classeurTournesol = app.Workbooks(1)
classeurTintin = app.Workbooks.Add()
classeurMilou = app.Workbooks("Milou.txt")

classeurTournesol.Worksheets(1).Cells(5, 5).value = 5
classeurTintin.Worksheets(1).Cells(5, 5).value = 5
classeurMilou.Worksheets(1).Cells(5, 5).value = 5
```

ClasseurTournesol référence le premier classeur, un nouveau classeur créé dans la ligne qui précède l'assignation.

ClasseurTintin référence un tout nouveau classeur créé sur la même ligne que l'application.

ClasseurMilou référence un classeur déjà existant qui est ouvert sur la même ligne que l'assignation.

La feuille (Worksheet)

Nous avons déjà utilisé la collection de *Worksheets* du classeur dans nos exemples précédents. Ajoutons ici quelques petits détails.

Référencer une feuille

Quand vous créez un *Workbook*, ce dernier possède déjà un certain nombre de *Worksheets* :

```
Dim app As New Excel.Application
Dim classeur As Excel.Workbook

Classeur = app.Workbooks.Add()
'ou classeur = app.Workbooks.Open("Toto.xls")

classeur.Worksheets(1).Cells(5, 5).value = 5
classeur.Worksheets(2).Cells(5, 5).value = 5
classeur.Worksheets(3).Cells(5, 5).value = 5
…
```

Il n'est cependant souvent pas possible de connaître d'avance le nombre de feuilles contenues dans un classeur[7], de sorte que la dernière commande de l'exemple précédent pourrait générer une erreur si jamais il n'y avait que 2 feuilles dans le classeur. Vous devrez donc parfois devoir déterminer le nombre de classeurs présents en utilisant la propriété *Count* de la collection de *Worksheets* :

```
If classeur.Worksheets.Count > 2 Then
   classeur.Worksheets(3).Cells(5, 5).value = 5
End If
```

Tous comme pour les *Workbooks*, référencer une *Worksheet* par son indice peut parfois causer des problèmes. Si on insère, enlève ou déplace des feuilles dans le classeur, la *Worksheet(2)* peut devenir la *Worksheet(1)*. Le code écrit pour la *Worksheet(1)* ne référence plus alors la bonne feuille. Un autre problème est que le code devient très difficile à lire. Il faut se souvenir constamment ce que représente *Worksheet(2)*, et on peut facilement faire des erreurs en programmant.

Il y a donc un gros avantage à nommer les feuilles. Cela peut être fait manuellement dans le classeur ou le modèle utilisé au départ pour créer des documents, ou bien dans le code :

```
Dim app As New Excel.Application
Dim classeur As Excel.Workbook

classeur = app.Workbooks.Add()
classeur.Worksheets(1).name = "Revenus"
classeur.Worksheets(2).name = "Dépenses"
```

[7] Ne vous fiez pas au fait que par défaut, il y a 3 feuilles (ou 5?) dans un nouveau classeur créé sur votre station de travail. Le nombre de feuilles dépend de la configuration d'Excel pour chaque utilisateur.

Important Évitez de référencer les feuilles par leur nom par défaut. Dépendant de la version d'Excel utilisée pour créer un nouveau classeur, le nom peut être aussi bien « Feuil1 » que « Sheet1 ».

Une fois que vous avez implicitement donné un nom à une feuille, les deux lignes suivantes donnent exactement le même résultat, mais la deuxième est plus facile à comprendre dans le code :

```
Dim x As Integer
x = CInt[8](classeur.Worksheets(2).Cells(5, 5).Value)
x = CInt(classeur.Worksheets("Dépenses").Cells(5, 5).Value)
```

Et si jamais on manipule les feuilles que la feuille 2 devient 3, *Worksheets("Dépenses")* continuera à référencer la bonne feuille. Excel ne vous laissera pas donner le même nom à deux feuilles.

Peut-être encore mieux[9], vous pouvez simplement fabriquer une variable pour référencer la feuille :

```
Dim app As New Excel.Application
Dim classeur As Excel.Workbook
Dim revenus As Excel.Worksheet
Dim dépenses As Excel.Worksheet

classeur = app.Workbooks.Add()
revenus = classeur.Worksheets(1)
dépenses = classeur.Worksheets(2)

Dim x As Integer
Dim y As Integer
x = CInt[8](dépenses.Cells(5, 5).Value)
y = CInt(revenus.Cells(5, 5).Value)
```

Ajouter ou enlever une feuille

Vous pouvez ajouter une feuille à la collection :

```
classeur.Worksheets.Add()
```

Vous pouvez aussi insérer une feuille entre deux autres. Par exemple, le code suivant insère une feuille avant la 2e, soit entre la 1ère et la 2e :

```
classeur.Worksheets.Add(classeur.Worksheets(2))
```

[8] Value retourne un *Object*, qui doit être converti en *Integer* pour qu'on puisse l'assigner à *x* quand *Option Strict* est réglé à *On* dans les propriétés de projet.

[9] Si Excel est visible, un utilisateur pourrait changer le nom de la feuille pendant que l'application est connectée à cette dernière.

Pour éliminer un classeur, appelez simplement sa méthode *Delete*. Dépendant de la méthode utilisée pour référencer la feuille, les trois lignes suivantes donnent le même résultat:

```
classeur.Worksheets(2).Delete()
classeur.Worksheets("dépenses").Delete()
dépenses.Delete()
```

Notez cependant que si vous détruisez une feuille en utilisant une variable qui la référence, la variable devient inutilisable tant que vous ne l'aurez pas associée à une autre feuille.

La plage (Range) : cellule, ligne, colonne, etc.

Savoir travailler avec l'application, les classeurs et les feuilles est essentiel, mais qu'est-ce qu'on veut manipuler habituellement dans une feuille Excel? Ce sont les cellules.

Bizarrement, il n'y a pas d'objet *Cell* dans Excel. Microsoft a pris une autre approche qui permet d'uniformiser la façon de travailler en créant un objet *Range* (on dirait une « plage » en français), qui est un groupe de cellules. Un *Range* peut donc contenir plusieurs cellules, mais il peut aussi en contenir une seule.

```
Dim app As New Excel.Application
Dim feuille As Excel.Worksheet

app.Workbooks.Add()
feuille = app.Workbooks(1).Worksheets(1)

'Les expressions suivantes référencent toutes des objets Range

feuille.Cells(5, 5)
feuille.Columns(2)
feuille.Rows(2)
feuille.Range("E4")
feuille.Range("B7:F7").Select
feuille.Range("B4,C8,D6,E4")
app.Selection
```

Cells, *Columns* et *Rows* représentent évidemment une cellule, une colonne ou une ligne.

Range peut référencer une seule cellule, devenant une alternative à *Cells*, mais peut aussi définir une plage ou un groupe de cellules complètement disparates, permettant de faire une opération sur plusieurs cellules simultanément.

La dernière, la *Selection*, qui s'applique à l'application plutôt qu'à la feuille, est la ou les cellules actuellement sélectionnées. Cette sélection peut avoir été faite manuellement par l'utilisateur, ou par une commande *Select* dans le code.

Si vous consultez l'aide en ligne pour les propriétés *Cells*, *Columns*, *Rows* et *Selection*, vous verrez que chacune « *Gets a Range object* ». Des noms de propriétés différentes, mais en dessous, toujours le même objet. L'intérêt d'une telle uniformité est qu'étant donné que ces éléments d'Excel sont tous représentés par une classe unique, vous les

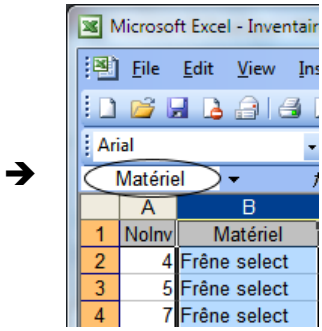
manipulez de la même façon. Ainsi, mettre en gras une cellule, une colonne ou une ligne se fait toujours avez la même propriété de l'objet *Range* :

```
feuille.Cells(5, 5).Font.Bold = True
feuille.Columns(2).Font.Bold = True
feuille.Rows(2).Font.Bold = True
feuille.Range("E4").Font.Bold = True
feuille.Range("B7:F7").Select.Font.Bold = True
feuille.Range("B4,C8,D6,E4").Font.Bold = True
app.Selection.Font.Bold = True
```

Tout comme pour les objets *Workbook* et *Worksheet*, référencer un *Range* par les types de références utilisés dans notre exemple (5,5 ou "E4") n'est pas très recommandable, parce que si la feuille est modifiée en insérant ou éliminant des lignes ou des colonnes, le code ne référencera plus les bonnes cellules.

Il est donc préférable de donner des noms aux cellules et plages qui sont référencées dans le code, un mécanisme appelé un *named range (*une *plage nommée)*. Il existe différentes façons de créer une plage nommée, incluant le code, mais on va habituellement faire ce travail dans la feuille ou le modèle utilisé pour emmagasiner les données.

Pour ce faire, il suffit de sélectionner la ou les cellule(s) dans Excel et taper un nom dans la petite zone de saisie en haut à gauche de la feuille. Dans l'illustration ci-dessous, nous avons tout d'abord sélectionné la colonne B, puis nous lui donnons le nom *Matériel*.



Important Si vous tapez simplement le libellé et cliquez ensuite à l'intérieur de la feuille, l'entrée n'est pas enregistrée. Vous devez activer *Enter (Entrée)* dans la zone de saisie après avoir tapé le nom.

Cette colonne peut maintenant être modifiée des 3 façons suivantes :

```
feuille.Columns(2).Font.Bold = True
feuille.Columns("B").Font.Bold = True
feuille.Range("B:B").Font.Bold = True
feuille.Range("Matériel").Font.Bold = True
```

D'après vous, laquelle de ces 4 syntaxes est la plus intéressante quand on relit le code? Laquelle va continuer à fonctionner correctement si jamais on insère une colonne entre la première et la deuxième? Si vous avez répondu `Range("Matériel")`, vous avez 100% à l'examen.

⊙ Si vous jetez un coup d'œil à la feuille *Inventaire* du fichier C:\Cours VS2010_\Exemples supplémentaires\VB_Excel\InventaireBois.xls utilisé pour nos

exemples, vous verrez que nous avons ainsi donné un nom à toutes les colonnes. Il suffit de sélectionner une colonne pour voir apparaître son nom dans la petite case de saisie en haut à gauche de la feuille.

On va aussi souvent donner un nom à une seule cellule. Par exemple, nous pourrions avoir une cellule qui contient le taux de la TPS. Même si on déplace des lignes ou des colonnes dans la feuille, la ligne suivante continuera toujours à donner le bon résultat :

```
taxeFed = valeur * feuille.Range("TauxTPS")
```

Truc Pour voir la liste des plages nommées dans un classeur et éventuellement sélectionner la plage pour voir ce qu'elle contient, activez *Edit► Go To (Édition► Aller à)* ou CTRL-G dans Excel.

Offset : références relatives

Une propriété intéressante de l'objet Range est *Offset*, qui permet de faire une référence relative à une cellule donnée. Par exemple *feuille.Cells(4,4).Offset(2,3)* référence la cellule qui est 2 lignes plus bas et 3 colonnes à droite de la cellule 4,4 soit la 6,7.

Des valeurs négatives comme *Cells(-2,-3)* référencent alors une cellule à gauche et plus haute.

Des valeurs 0 permettent des référencer une autre cellule sur la même ligne *Cells(0,3)* ou sur la même colonne *Cells(2,0)*.

Naviguer dans une ligne ou une colonne

Naviguer dans une ligne ou une colonne est une opération assez courante. Pour ce faire, utilisez la commande *For Each*, étudié dans la portion *Contrôle de l'application* du manuel de cours d'introduction. Ainsi, le code suivant double la valeur de toutes les cellules sur la 3e ligne d'une feuille :

```
For Each cellule As Excel.Range In feuille.Rows(3).Cells
  If cellule.Value = Nothing Then Exit For
  cellule.Value = cellule.Value * 2
Next
```

Notez que nous devons avoir un mécanisme pour détecter la dernière cellule, sinon la boucle va inscrire une valeur sur toutes les cellules de la ligne, de la colonne A à la colonne ZZ. Dans notre exemple, nous assumons que toutes les cellules de la ligne contiennent une valeur, alors nous détectons tout simplement l'arrivée dans la première cellule vide pour sortir de la boucle avec un *Exit For*. Si certaines des cellules de la ligne pouvaient être vides, nous créerions plutôt une colonne marqueur, une colonne contenant une valeur fixe comme par exemple un « X ». Ce marqueur deviendrait alors notre indicateur de limite :

```
For Each cellule As Excel.Range In feuille.Rows(3).Cells
  If cellule.Value = "X" Then Exit For
  cellule.Value = cellule.Value * 2
Next
```

Valeur ou formule

Quand vous travaillez avec la propriété *Value* d'une cellule, vous modifiez sa valeur. Si vous voulez travailler avec une formule, vous utiliserez plutôt sa propriété *Formula* :

```
feuille.Cells(5, 4).Value = 5
feuille.Cells(5, 5).Value = 10
feuille.Cells(5, 6).Formula = "=SUM(E4:E5)"
```

Tout comme on le fait quand on crée manuellement une feuille, on doit ajouter un signe égal devant l'expression pour qu'elle soit considérée comme une formule.

Problèmes de performance?

Comme nous l'avons vu plus haut, les communications entre une application .NET et une application COM comme Excel ne se font pas directement, elles doivent passer par un interop (page 5), mais aussi par OLE[10], le gestionnaire de communications dans le monde COM :

NET → Interop → OLE → Excel → OLE → Interop → .NET

Chaque communication doit donc transiter, aller et retour, au travers de 5 couches intermédiaires, qui servent entre autres à convertir les formats de données qui ne sont pas tous les mêmes entre .NET et COM.

Encore pire, si vous avez une commande composée de plusieurs objets, ce trajet peut être effectué plusieurs fois sur une même commande :

```
App.Workbooks(1).Worksheets(1).Cells(5,5).Font.Italic = True
App.Workbooks(1).Worksheets(1).Cells(5,5).Font.Bold = True
App.Workbooks(1).Worksheets(1).Cells(5,5).Font.Name = "Consolas"
App.Workbooks(1).Worksheets(1).Cells(5,5).Font.Size = 8
```

Dépendant de la façon dont le compilateur voit cette ligne, il est possible que le voyage aller-retour se fasse jusqu'à 6 fois sur chacune des lignes, 24 fois au total pour formater la cellule, 144 communications entre les intermédiaires.

Si vous effectuez un grand nombre d'opérations, particulièrement dans des boucles, la performance peut en prendre un coup.

Il y a 4 trucs pour vous aider à régler des problèmes de performance.

- Cherchez pour voir s'il n'existerait pas une méthode permettant de passer plusieurs paramètres en une seule ligne de code. Il n'y en a malheureusement pas pour le formatage d'un *Range*. Mais si vous jetez un coup d'œil à la méthode *SaveAs* de l'objet *Workbook*, vous verrez qu'elle possède une foule de paramètres qui correspondent à des propriétés du *Workbook*. Appeler *SaveAs* en passant ces

[10] *Object Linking and Embedding*, souvent appelé ActiveX, est le gestionnaire d'échanges entre les applications dans le monde COM.

paramètres est plus efficace que de régler les propriétés avant d'appeler *Save* ou *SaveAs*.

- Quand vous utilisez le même objet sur plusieurs lignes, utilisez une variable pour le référencer. Si on revient aux lignes ci-dessus, elles vont probablement[11] s'effectuer plus rapidement si vous les écrivez ainsi :

```
Dim fnt As Excel.Font
fnt = App.Workbooks(1).Worksheets(1).Cells(5,5).Font

fnt.Italic = True
fnt.Bold = True
fnt.Name = "Consolas"
fnt.Size = 8
```

- À la place, vous pourriez utiliser une structure *With* [11]:

```
With App.Workbooks(1).Worksheets(1).Cells(5,5).Font
   .Italic = True
   .Bold = True
   .Name = "Consolas"
   .Size = 8
End With
```

- Appelez une macro VBA qui fait le travail dans le document Excel.

Appeler une macro VBA

⊙ Le fichier Excel utilisé pour nos exemples contient un module *VBNet_Exel* dans lequel nous avons créé une macro appelée *FormatCellules*, qui reproduit en VBA le code .NET utilisé plus haut :

```
Public Sub FormatCellules(ParamArray cellules())

   Dim cellule As Variant
   Dim x As Integer

   For Each cellule In cellules
      With cellule.Font
         .Italic = True
         .Bold = True
         .Name = "Consolas"
         .Size = 8
      End With
   Next

End Sub
```

L'utilisation d'un *ParamArray*[12] est optionnelle, mais permet de passer simultanément plusieurs cellules, augmentant encore un peu la performance.

---

[11] Le compilateur tente d'optimiser votre code, et peut parfois faire un travail équivalent sous la couverture au moment de compiler le .exe.

[12] Array contenant un nombre variable de paramètres, discuté brièvement dans le manuel d'introduction.

Pour appeler une macro, vous invoquez la méthode *Run* de l'objet application :

```
app.Run("FormatCellules", feuille.Cells(5,5))
```

Ou ainsi pour passer plusieurs cellules :

```
app.Run("FormatCellules", feuille.Cells(3, 4), feuille.Cells(4, 5), …)
```

Interface simple

⊙ Notre application d'exemple contient un formulaire *FormExcel* dans lequel nous avons mis des commentaires à profusion, et dans lequel nous avons travaillé régulièrement durant le cours. Nous n'insisterons donc pas ici, le code vous donne les détails importants. Mais voici quelques pointeurs.

- L'application est lancée dans la procédure *Main* du module *ModuleExcel*. Cette procédure lance le formulaire *FormExcel* qui constitue l'interface principale de l'application. La fermeture de ce formulaire déclenche la fermeture de l'application en « relâchant » la commande *Application.Run* qui reste en suspend tant que le formulaire est ouvert. *Main* termine alors son travail, arrête Excel s'il roule encore, et se termine. Comme on avait lancé l'application dans *Main*, l'application s'arrête.
- Comme la commande *Application.Run* reste en fonction durant toute la durée de vie de l'application, et comme elle est entourée d'un *Try…Catch*, nous sommes assurés[13] de toujours fermer Excel en cas de problème non prévu. C'est la principale raison pour laquelle nous lançons l'application dans une procédure *Main* plutôt que de partir directement dans le formulaire comme nous pourrions le faire dans l'option *Startup Object* de l'onglet *Application* des propriétés de projet.
- Nous avons réglé la propriété *FormBorderStyle* du formulaire à *FixedDialog*. Par défaut, cette propriété est à *Sizable*, ce qui n'est pas pertinent pour notre utilisation.
- L'événement *Load* de ce formulaire ouvre un document Excel appelé InventaireBois.xls localisé dans le répertoire de l'application, et établit une référence à une feuille appelée « Inventaire », que nous référençons dans toute l'application par l'intermédiaire de la variable publique *inventaire*.

Important	Nous avons localisé le fichier dans le répertoire de l'application pour faciliter le transport des exemples de cours d'une salle de formation à l'autre. Dans la réalité, mettre un document dans le répertoire de l'application n'est généralement pas une bonne idée, puisque par défaut, dans .NET, le répertoire Program Files est en lecture seule pour les utilisateurs qui ne sont pas des administrateurs.

[13] Certains types d'erreurs peuvent empêcher le système de remonter jusque là, mais c'est relativement rare.

- Un élément majeur de ce formulaire est la variable *ligne*, définie dans la section *Déclarations* du formulaire, et qui est utilisée un peu partout pour déterminer la ligne actuellement affichée à l'écran. Cette variable est modifiée chaque fois que l'utilisateur clique sur un des boutons de navigation.
- La méthode *Affiche* s'occupe de lire une ligne dans la feuille et de l'afficher dans l'interface. Cette méthode est appelée au lancement du formulaire (événement *Load*) pour afficher la première ligne, ainsi que chaque fois que l'utilisateur se déplace d'une ligne à l'autre.
- À l'inverse, la méthode *Enregistre* s'occupe de récupérer les données de l'interface. Elle est appelée avant tout déplacement d'une ligne à l'autre, de même qu'à la fermeture du formulaire, dans son événement *FormClosing*.
- Les événements regroupés dans la région *Navigation* servent à se déplacer d'un item à l'autre, donc d'une ligne à l'autre dans la feuille Excel.
- Dans les événements regroupés sous *Autres*, *TextBox_Leave* s'assure que tous les *TextBox* où c'est pertinent[14] contiennent bien une valeur numérique. *ToucheEntrée_KeyUp* permet pour sa part d'avancer d'une zone de saisie à l'autre par la touche ENTRÉE[15].

Utilisation des grilles

Qui dit Excel dit grille de saisie.

Qui dit grille de saisie à un programmeur expérimenté voit la commissure des lèvres de ce dernier s'affaisser très bas. Travailler avec une grille est toujours un travail complexe, surtout avec la *DataGridView*, la grille relativement sommaire fournie avec .NET.

Excel a été conçu pour un utilisateur, il est convivial. La *DataGridView* a été pensée pour un programmeur. En soi, elle ne fait pas grand-chose d'autre qu'afficher et saisir des données. C'est ça que vous voulez faire, direz-vous. Mais vous allez la trouver très limitée, vous qui êtes habitués de travailler dans Excel.

La simplicité de la DataGridView donne au programmeur la latitude de pouvoir la faire fonctionner comme il le veut bien. Mais cela demande un travail considérable. Vous voulez donner la possibilité à l'utilisateur de mettre une colonne en gras? Vous devez ajouter le bouton (ou autre mécanisme) et le code pour le faire. Vous voulez laisser l'utilisateur déterminer la largeur des colonnes et que la grille ouvre dans le même état à l'utilisation suivante de l'application? Vous devez écrire le code pour enregistrer la largeur des colonnes à quelque part (habituellement un fichier) et réutiliser ces valeurs pour recréer la grille au lancement de l'application.

---

[14] Voir la clause *Handles* à la fin de la déclaration de la procédure événementielle.

[15] Dans Windows, le défaut est d'utiliser la touche Tab. Si on veut que l'utilisateur puisse avancer avec ENTRÉE, il faut intervenir par le code.

La *DataGridView* a été à priori pensée pour faciliter le travail avec des bases de données et des collections .NET. Or, Excel n'est pas une base de données[16]. Et bien qu'il travaille beaucoup avec des collections (collection de classeurs, de feuilles, de lignes, de colonnes, de cellules et quelques autres dont nous n'avons pas parlé dans ce cours), ces collections ne sont pas structurées comme le sont des collections .NET et ne peuvent pas s'intégrer automatiquement dans une grille .NET comme le feraient des collections natives à l'environnement.

Vous avez déjà une grille, très performante et très conviviale, que vous connaissez déjà : Excel lui-même.

Avant de vous engager dans un travail qui peut devenir fastidieux, posez-vous toujours la question suivante : qu'est-ce que je gagne à tenter de recréer les fonctions d'Excel dans une grille .NET?

Les programmeurs qui veulent utiliser des grilles dans leurs applications finissent presque toujours par délaisser la *DataGridView* et acheter une grille .NET d'une tierce-partie. Certaines de ces grilles sont d'ailleurs conçues pour travailler avec Excel, de sorte qu'emplir la grille n'est l'affaire que d'une ligne de code ou deux au lieu de tout ce que nous devons faire dans notre application d'exemple qui est pourtant très sommaire et incomplète.

Si vous faites une recherche pour « Grid.NET » chez ComponentSource[17], il y a 188 entrées, ce qui démontre à quel point les programmeurs cherchent à remplacer la grille .NET par quelque chose d'autre. Si vous ajoutez le mot Excel dans la recherche, ça diminue à seulement 55 entrées, mais ça indique encore une fois que bien des gens ont les mêmes besoins que vous pour afficher des données Excel dans .NET, et qu'ils sont prêts à dépenser de bonnes sommes pour ne pas avoir à utiliser la *DataGridView* du framework.

Personnellement, chez nos clients, l'entente a toujours été très claire. Si on a besoin de travailler dans une grille, on utilise Excel avec du code VBA pour aider à gérer et valider ce qui se passe. On utilise VB.NET quand on a du traitement à faire sur des feuilles existantes, quand on veut créer des feuilles par le code, ou quand on veut créer des interfaces alternatives comme nous le faisons dans le formulaire *FormExcel* de nos exemples.

En tant que formateur, nous tentons par ailleurs de nous limiter le plus possible à ce qui est fourni avec .NET. Si nous nous mettions à utiliser des grilles commerciales, nous serions incapables de répondre aux questions de nos étudiants dans nos cours sur Visual Studio, le framework et Visual Basic.

Nous n'avons donc pas d'expérience significative avec les grilles de fournisseurs indépendants, et ne pouvons donc pas vous en suggérer une plutôt qu'une autre. Nous pouvons quand même vous donner quelques orientations sur les grilles les plus utilisées.

---

[16] Bien qu'un trop grand nombre d'utilisateur s'en servent à cet usage.

[17] ComponentSource est le Amazon des programmeurs, à www.componentsource.com.

Les plus populaires sont la TrueDBGrid et la FlexGrid vendues par ComponentOne[18], une compagnie qui a des relations de longue date avec Microsoft, puisque c'est une version de TrueDBGrid qui était fournie avec Visual Basic dans les années 90.

Un grand nombre de programmeurs préfèrent les contrôles (une grille est un contrôle) de la compagnie Infragistic[19], qui sont visuellement très modernes. Ils ont entre autres un contrôle appelé Infragistics.Excel qui simule presque complètement l'interface d'Excel.

Attendez-vous cependant à peut-être payer plus cher pour ces contrôles que vous ne l'avez fait pour Visual Studio, et à devoir passer aux travers de manuels de 600 pages et plus pour pouvoir les maîtriser.

Notre exemple

⊙ Ceci étant dit, nous vous avons quand même préparé un petit exemple d'utilisation de la *DataGridView*, que vous trouverez dans le formulaire *FormGrille* de notre projet *VB_Excel*. Vous pouvez l'activer dans l'application à partir du bouton [*Grille…*] du formulaire principal.

Important Ne vous fiez pas à cet exemple pour porter un jugement. Le code présenté pour démontrer les concepts de base est relativement simple. Quand on commence à vouloir travailler sur une grille relativement conviviale et qui évite le plus possible les erreurs de saisie, on va très souvent se retrouver avec plusieurs centaines de lignes de code pour chaque grille[20].

Comme pour *FormExcel*, nous avons pris le soin de bien documenter le code, alors nous n'insisterons pas beaucoup ici. Nous prendrons cependant la peine de mentionner une petite technique que nous utilisons pour éviter certains problèmes courants quand on manipule Excel à partir d'une autre application.

Tout d'abord, il y avait avantage à ce que l'utilisateur puisse agrandir ce formulaire en fonction des besoins. Vous remarquerez que quand on change les dimensions du formulaire, le bouton et de la grille ne réagissent pas de la même façon. C'est déterminé par la propriété *Anchor* des deux contrôles. Nous avons aussi ajusté la propriété *MinimumSize* du formulaire pour éviter un affichage qui n'aurait aucun de sens.

Au moment d'activer ce formulaire, notre document Excel est déjà ouvert dans *FormExcel*. Si vous lancez un deuxième formulaire comme nous le faisons dans

---

[18] www.componentone.com/SuperProducts/StudioWinForms

[19] www.infragistics.com/dotnet/netadvantage/winforms.aspx

[20] En fait, la feuille Excel de notre exemple vient de l'exportation d'une table dans une base de données, que nous manipulons par une DataGridView dans une application que nous sommes en train de développer. En sachant que nous n'avons pas encore finalisé tout le travail, que la grille .NET est optimisée pour des bases de données (pas pour Excel), que nous avons déjà une version de la DataGridView modifiée et optimisée pour nos besoins et notre façon de travailler, et que nous utilisons une librairie d'outils développée au cours des années pour simplifier une partie des validations, nous en sommes à 910 lignes de codes uniquement pour rendre la grille conviviale. Nous prévoyons en avoir autant pour la portion de validation qui n'est pas encore écrite.

btnGrille_Click, et que dans ce nouveau formulaire vous vous connectez à la feuille Excel par le même mécanisme que vous l'aviez fait dans le premier, vous allez vous retrouver avec deux copies d'Excel en mémoire[21], chacune travaillant sur le même document. Le premier des deux formulaires qui ferme enregistre ses changements. Le deuxième écrase les changements enregistrés par l'autre.

Et ça, c'est si vous êtes chanceux. Dépendant de différentes conditions qui dépendent entre autres des options d'Excel là où le programme est installée, vous risquez de plutôt voir l'application se terminer abruptement à cause d'une exception.

Pour corriger ce problème, nous passons donc au deuxième formulaire (*FormGrille*) une référence à la feuille que nous avons déjà en mémoire dans le premier (*FormExcel*), par l'intermédiaire d'une variable publique définie dans *FormGrille* :

```
Dim frmGrille As New FormGrille
frmGrille.Inventaire = inventaire
```

Dans votre travail de programmation, et c'est probablement arrivé durant le cours, vous allez par ailleurs rapidement constater qu'on crée souvent des bogues quand on travaille dans un document ouvert simultanément à partir de deux interfaces, par exemple, à partir de votre application et dans Excel lui-même quand on le rend visible pendant que l'application est en fonction. Un changement effectué dans l'une des deux interfaces ne se reflète pas toujours immédiatement dans l'autre, et cet autre peut éventuellement écraser le changement fait dans le premier.

C'est aussi le cas si les deux interfaces sont dans la même application.

Si l'utilisateur fait dans *FormGrille* un changement sur l'enregistrement actuellement affiché dans *FormExcel*, le changement fait dans *FormGrille* sera écrasé quand *FormExcel* va appeler sa méthode *Enregistre*. Il faut donc éviter autant que possible que l'utilisateur puisse travailler simultanément dans deux écrans qui manipulent les mêmes données (la feuille *Inventaire* dans notre cas).

Pour ce faire, certains programmeurs vont « jouer » avec la propriété *Visible* des formulaires pour s'assurer qu'un seul est visible à la fois. Dans notre expérience, ça a tendance à faire paniquer les utilisateurs qui ont l'impression d'avoir perdu leur travail quand ils voient disparaître un écran de saisie.

Une meilleure alternative est de travailler avec des formulaires spéciaux, des dialogues, qu'on appelle parfois des formulaires modaux. C'est le mécanisme utilisé dans la plupart des applications pour les fenêtres *About... (À propos de...)*. Ces fenêtres figent les autres quand on les affiche. Il faut les fermer avant de pouvoir retourner dans l'application.

Pour ce faire, plutôt que d'afficher notre formulaire de grille comme on le fait couramment…

```
frmGrille.Show()
```

… nous l'affichons plutôt avec la méthode ShowDialog :

```
frmGrille.ShowDialog()
```

---

[21] Une nouvelle copie d'Excel est lancée chaque fois que vous faites *New Excel.Application*.

L'utilisateur ne pourra alors plus travailler dans *FormExcel* tant que le formulaire contenant la grille sera affiché à l'écran.

Quelques petits pointeurs aux endroits intéressants dans le code de *FormGrille* :

- La grille ne contient aucune colonne par défaut. Les colonnes doivent être incorporées à la grille avant de pouvoir y entrer des données. Ça peut être fait en activant la propriété *Columns* dans la fenêtre de propriétés, quand la grille est sélectionnée dans l'environnement de développement. Pour avoir plus de contrôle nous préférons personnellement faire le travail dans le code, comme nous le faisons dans la méthode *PréparerGrille*.
- Emplir la grille est relativement simple, il suffit de boucler dans les lignes d'Excel et pour chacune, ajouter une ligne dans la *DataGridView*. À l'intérieur de chaque ligne, on copie ensuite les valeurs qu'on trouve dans Excel vers les colonnes équivalentes de la grille .NET, en étant conscient que les lignes et colonnes d'Excel sont indexées à partir de 1 et qu'on commence à la deuxième ligne[22], tandis que celles de la grille .NET le sont à partir de 0. Ce code est dans la méthode *EmplirGrille*.
- À l'inverse, le transfert des données de la grille .NET au document Excel pourrait être fait de la même façon, en lançant une routine qui boucle dans les cellules de la grille .NET et copie les valeurs qu'elle y rencontre dans la feuille Excel. Comme cela peut prendre quelques secondes, c'est le genre d'opération qu'on ferait normalement à la fermeture du formulaire. En fonctionnant ainsi cependant, si l'application « plante », l'utilisateur perd tout son travail. Nous préférons donc faire ce transfert chaque fois qu'une nouvelle valeur est entrée dans la grille, ce qui est le rôle de l'événement *dgvInventaire_CellEndEdit*, qui est appelé quand l'utilisateur sort d'une cellule après y avoir fait un changement.
- C'est aussi dans ce même événement *dgvInventaire_CellEndEdit* que nous faisons les calculs correspondant aux colonnes calculées dans Excel.

[22] La première ligne de notre feuille Excel est simplement une en-tête.