Implémentation du modèle Singleton

Le `Singleton` est le modèle le plus souvent utilisé pour présenter le principe des modèles de conception. C'est l'un des plus faciles à mettre en œuvre, c'est pourquoi il est compréhensible et utilisable même pour des débutants.

Il permet de garantir l'unicité de l'instance d'une classe et son accessibilité à partir de n'importe endroit du programme. Les cas de figure où une instance unique présente un intérêt sont multiples. Par exemple, dans la programmation d'un jeu, le moteur de son, chargé de gérer la lecture des sons et des musiques, doit être unique. Vu son utilisation omniprésente, il doit être accessible de partout.

Ce modèle fait appel à plusieurs principes de la programmation orientée objet. L'idée principale du `Singleton` est que l'instance est englobée dans la classe. Elle sera donc un attribut de cette classe. Cet attribut devra être privé, ce qui empêchera son accès à tout élément extérieur à la classe. De plus, il devra être initialisé à `Nothing` pour que l'on puisse vérifier son existence. Pour pouvoir y accéder, on devra implémenter un accesseur. Cet accesseur devra être partagé. De cette manière, on pourra y accéder de partout, en invoquant la classe. De plus, le constructeur devra être privé. De cette manière, on empêchera une instanciation manuelle de l'objet. Or, si le constructeur est privé, comment créer l'instance de la classe ? Cela sera fait dans l'accesseur. Celui-ci devra vérifier si l'instance de la classe existe. Si elle existe, il la retournera. Dans le cas contraire, il l'instanciera et la retournera. Cet accesseur est la clé du modèle. En effet, c'est lui qui vérifie l'existence et l'unicité de l'instance, et la renvoie le cas échéant. Voici un schéma de ce modèle.

La première étape consiste à déclarer la classe avec son constructeur privé, pour empêcher l'instanciation :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub
End Class
```

Le constructeur peut contenir d'autres opérations à effectuer, par rapport à l'utilisation de la classe.

On crée ensuite l'instance de la classe. La particularité du `Singleton` est que l'instance est un attribut privé de cette classe. Celui-ci doit être

également partagé, car l'accesseur sera également partagé pour être accessible à partir de la classe, et non d'un objet. Il doit être initialisé à `Nothing`. De cette manière, on saura s'il existe ou pas. S'il vaut `Nothing`, cela signifie qu'il n'existe pas. Sinon il existe :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing
End Class
```

Il faut ensuite implémenter l'accesseur qui permettra de récupérer cette instance. Si l'instance n'existe pas, c'est-à-dire si l'attribut `instance` vaut `Nothing`, cet accesseur doit créer l'instance. Il le pourra car il a accès au constructeur, même si celui-ci est privé. Ensuite, il retournera l'instance.

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing

' L'accesseur crée l'instance si elle n'existe pas
' Puis il la retourne
Public Shared Function getInstance() As Singleton
  If (Me.instance Is Nothing)
    Me.instance = New Singleton()
  End If
  Return Me.instance
End Function

End Class
```

Le `Singleton` est maintenant totalement défini. Pour pouvoir y avoir accès et l'utiliser, il suffit d'invoquer la classe et de faire appel à sa méthode `getInstance` :

```
Singleton.getInstance()
```

Bien qu'il existe, le `Singleton` ne fait rien pour l'instant. On lui donne un nom, que l'on va modifier et réafficher, de manière à vérifier l'unicité de l'instance :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing

' L'accesseur crée l'instance si elle n'existe pas
' Puis il la retourne
Public Shared Function getInstance() As Singleton
  If (Me.instance Is Nothing)
    Me.instance = New Singleton()
  End If
  Return Me.instance
End Function

  Public nom As String = "Toto"
End Class
```

Le `Singleton` s'appelle `Toto` à l'origine. On modifie son nom que l'on réaffiche pour savoir si cela a affecté la bonne personne :

```
MessageBox.Show("nom du singleton : " & _
 Singleton.getInstance().nom)
Singleton.getInstance().nom = "Bibi"
MessageBox.Show("nom du singleton : " & _
 Singleton.getInstance().nom)
```

La première boîte de dialogue affiche le nom d'origine, "*Toto*". Ensuite, on modifie celui-ci. D'après le principe du `Singleton`, il n'y a qu'une instance. Par conséquent, celui qui tout à l'heure s'appelait Toto doit d'appeler Bibi maintenant. On écrit donc la même instruction. Effectivement la seconde boîte de dialogue confirme que le `Singleton` s'appelle maintenant Bibi.

Un modèle de conception est une enveloppe pour des classes correspondant à des problématiques. Vous devrez les compléter pour les adapter au contexte, en y ajoutant les attributs et les méthodes nécessaires. Sinon, vous pouvez utiliser les autres mécanismes de la programmation orientée objet. Par exemple, si vous faites de `Singleton` une classe mère dont d'autres classes peuvent dériver, ces

autres classes n'auront également qu'une seule instance. À vous de voir comment ces modèles s'adaptent le mieux.

Quelques modèles de conception courants

Le nombre des modèles de conception est considérable. Chacun d'entre eux résout une problématique donnée. Dans cette partie, nous survolerons quelques-uns des modèles de conception les plus courants. Nous n'entrerons pas dans les détails et ne les implémenterons pas non plus, car ce n'est pas le sujet de cet ouvrage. Cependant, c'est un aspect important de la programmation, car ce sont des techniques valables, éprouvées, efficaces et fiables.

DisposeFinalize

Vous devriez pour l'instant laisser le ramasse-miettes s'occuper de la libération de la mémoire, car vous risqueriez d'être moins efficace que lui en tentant de forcer vous-même la libération. Effectivement, libérer la mémoire proprement n'est pas chose aisée. L'implémentation du modèle `DisposeFinalize` permet de bien le faire.

Il est composé de plusieurs éléments :

- Il faut tout d'abord un attribut booléen pour déterminer si la destruction a été faite.
- Puis, on doit avoir une fonction `Dispose`. Celle-ci fait appel à une fonction surchargée prenant en paramètre un booléen. Ce booléen détermine si c'est vous qui avez forcé la destruction de l'objet, ou si c'est le ramasse-miettes. La fonction `Dispose` sans paramètre appelle la fonction `Dispose` avec le paramètre `true`, puis la fonction `SuppressFinalize` du ramasse-miettes.
- Ensuite, il faut redéfinir la fonction `Finalize`. Elle appellera la fonction `Dispose` avec le paramètre `false` pour prévenir que c'est le ramasse-miettes qui a lancé la destruction. Si la classe en cours est une classe fille, il faut libérer la classe mère.
- Dans la classe `Dispose` surchargée, selon le paramètre, le traitement de la libération des différents attributs devra être faite.

Si vous implémentez ce modèle, vous pourrez alors détruire vous-même vos objets en faisant appel à la méthode `Dispose`. Elle se chargera de libérer la mémoire dédié à l'objet et aux autres objets qui dépendent de lui.

Iterator

Lorsque les projets sont relativement conséquents, les structures de données complexes, il peut être fastidieux de les parcourir dans leur globalité. En effet, on ne peut pas toujours se contenter d'utiliser une simple boucle `for`, en partant du premier au dernier élément, comme dans les tableaux.

Le modèle `Iterator` permet justement de parcourir des classes structurées complexes, comme si elles n'étaient que de simples listes. De plus, utiliser ce modèle permet de parcourir la structure, mais sans exposer son fonctionnement intérieur. Pour cela, on a besoin de trois éléments :

- Il faut une interface définissant un itérateur. Cette interface contient au moins une méthode permettant d'accéder à l'élément courant, ainsi qu'une méthode permettant d'accéder à l'élément suivant. On peut compléter cette interface en ajoutant la possibilité de supprimer l'élément en cours, mais l'indispensable est de pouvoir accéder à l'élément en cours et au suivant.
- Il faut une classe `MonIterator` adaptée à la structure qui implémente l'interface `Iterator`. Cette classe saura parcourir la structure.
- Dans la classe correspondant à une structure, il faut une méthode qui crée et renvoie un itérateur capable de parcourir cette structure.

Vous pourrez alors parcourir cette structure sans vous soucier de son fonctionnement, étant donné que l'itérateur est fait pour cela, et adapté à ladite structure.

État

Les comportements d'un objet dépendent souvent d'un certain état, à un moment donné. Un magasin par exemple est susceptible d'accueillir des clients s'il est dans un état "*ouvert*". S'il est dans un état "*fermé*", il n'y a pas de clients dedans et les lumières sont éteintes.

La notion d'état détermine donc un certain comportement de l'objet à un moment donné. Le rôle de ce modèle est de symboliser cette notion d'état, sans toucher à l'objet lui-même. De cette manière, on ne rend pas les états dépendant de l'objet. Si par exemple on doit ajouter ou supprimer des états, il ne sera pas nécessaire de remodifier totalement l'objet lui-même.

La méthode communément utilisée pour ce problème est de définir les états comme des constantes (`ETAT_OUVERT`, `ETAT_FERME`), d'avoir un attribut symbolisant l'état en cours, et selon la valeur de cet attribut, de faire les opérations ad hoc. Si le nombre d'états est important, le code peut devenir confus.

Le principe de ce modèle est de définir une interface correspondant à un état. Ensuite, chaque état sera une classe implémentant cette interface.

- Tout d'abord, il faut créer l'interface. Cette interface contiendra l'ensemble des méthodes en rapport avec l'objet, par exemple pour le magasin `servirClient`, `rangerMateriel`, etc., c'est-à-dire l'ensemble des méthodes qui ont une dépendance par rapport à un état donné.
- Ensuite, il faut créer les classes des différents états et leur faire implémenter l'interface. Par exemple, pour l'état `magasinFerme`, la classe retournera une exception, alors que pour `magasinOuvert`, elle effectuera l'opération comme il faut.
- Il faut enfin décrire le mécanisme de changement d'état, permettant de passer de l'un à l'autre et de définir un état de départ.

De cette manière, vous fournissez à l'objet un comportement qui dépend d'un état donné. Or, ce traitement est séparé de l'objet lui-même, ce qui permet de gagner en compréhension, en souplesse et en fiabilité.

L'étude des modèles de conception est bénéfique pour la compréhension des mécanismes de la programmation orientée objet. Nous en avons présenté ici trois qui sont relativement simples. D'autres sont plus complexes et il en existe une multitude, chacun adapté à un problème donné. N'hésitez pas à en utiliser, vos programmes seront de meilleure qualité.

9.5. Quelques bonnes habitudes à prendre

Un bon programme repose sur les habitudes de son programmeur. Il est rare d'avoir un programme de qualité si celui qui l'a conçu l'a fait de manière désinvolte, sans faire attention.

Nous exposerons ici quelques bonnes habitudes à prendre, que vous avez mises en œuvre pour la plupart. Vous comprendrez tout leur intérêt et vous les aurez sous la main.

Pour une meilleure compréhension, bien indenter

L'indentation consiste à commencer des lignes du même bloc au même niveau. On utilise des tabulations pour les décaler. Cela permet une meilleure compréhension dans le code source. La règle est : on entre dans un bloc, on indente, on sort d'un bloc, on revient au niveau de départ. Un bloc peut être la déclaration d'une fonction, une structure de contrôle, etc.

```
Public Sub testIndent(ByVal ind As Boolean)
  If ind
MessageBox.Show("Le paramètre est vrai")
  Else
    MessageBox.Show("Le paramètre est faux")
  End If
End Sub
```

Vous avez ici trois niveaux d'indentation :

- La déclaration de la fonction, alignée à gauche.
- Le contenu de la fonction, qui est décalé d'une tabulation après la déclaration de celle-ci. À ce niveau seront alignées toutes les instructions de la fonction, dont le `If` qui ouvre un nouveau bloc.
- L'intérieur des blocs du `If` et du `Else`, contenant les instructions d'affichage.

L'indentation permet un gain réel de lisibilité. Voici un code sans indentation :

```
Public Sub testIndent(ByVal ind As Boolean)
If ind
MessageBox.Show("Le paramètre est vrai")
Else
MessageBox.Show("Le paramètre est faux")
End If
End Sub
```

Ce bout de programme reste compréhensible car il est petit. Mais il est beaucoup moins lisible ainsi. C'est pourquoi nous vous recommandons de bien indenter vos programmes.

Être clair et expliquer

Soyez explicite. De cette manière, en lisant le code source ultérieurement, vous comprendrez son sens et son intérêt. Prenons le `Singleton` examiné précédemment :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing

' L'accesseur crée l'instance si elle n'existe pas
' Puis il la retourne
Public Shared Function getInstance() As Singleton
  If (Me.instance Is Nothin)
    Me.instance = New Singleton()
  End If
  Return Me.instance
End Function

  Public nom As String = "Toto"
End Class
```

Cette classe est claire, explicite et commentée. Grâce aux noms des différents éléments, on comprend à quoi ils correspondent. Vous pouvez vous reporter aux commentaires pour chercher des compléments d'information.

Voici la même classe, sans commentaire, avec des noms qui ne sont pas explicites du tout :

```
Public Class Ljopusd
  Private Sub New()
  End Sub

Private Shared utjvqsd As Ljopusd = Nothing

Public Shared Function oieurIqsg() As Ljopusd
  If (Me.utjvqsd Is Nothing)
    Me. utjvqsd = New Ljopusd()
  End If
  Return Me.utjvqsd
  End Function

  Public pisudgsglqjef As String = "Toto"
End Class
```

On a du mal à s'imaginer que cela est un `Singleton`. Pourtant, c'est exactement la même classe que la précédente, avec un comportement identique. D'où l'intérêt d'être explicite et de mettre des commentaires.

Tester les éléments séparément d'abord

Quand vous testez l'application, faites-le élément par élément. On ne sait jamais, il est possible qu'un défaut quelque part soit compensé par un autre défaut ailleurs. Prenons un cas évident :

```
Public Function addition1(ByVal nbr1 As Integer, _
 ByVal nbr2 As Integer)
  Return nbr1 + nbr2 + 1
End Function

Public Function addition2 (ByVal nbr1 As Integer, _
 ByVal nbr2 As Integer)
  Return nbr1 + nbr2 - 1
End Function
```

Ces deux fonctions effectuent une addition. Elles sont toutes les deux fausses au niveau du sens. Testez-les ainsi :

```
Dim res As Integer
res = addition1(3, 4) + addition2(6, 9)
MessageBox.Show("res = " & res)
```

Figure 9.43 :
Résultat correct avec des fonctions fausses

Vous obtiendrez effectivement le bon résultat car les deux erreurs s'annulent. Pourtant les deux fonctions sont fausses. C'est pourquoi, il faut faire les tests des éléments séparément :

```
MessageBox.Show("somme = " & addition1(3, 4))
MessageBox.Show("somme = " & addition2(6, 9))
```

Figure 9.44 :
Erreur sur addition1

Figure 9.45 :
Erreur sur addition2

De cette manière, vous vous rendrez compte que les deux fonctions sont fausses, et en plus vous pourrez comprendre l'erreur.

Forcer un comportement pour les cas d'erreur

Le pire dans les programmes est l'imprévu. En effet, en tirant parti des cas de figure non traités, les pirates provoquent des bogues, prennent le contrôle sur certaines choses.

Lisez à ce sujet le chapitre ***Rendre un programme robuste.***

```
Dim age As Integer
age = -1
If (age < 18) Then
  MessageBox.Show("Vous êtes mineur")
Else
  MessageBox.Show("Vous êtes majeur")
EndIf
```

Ici, il y a un gros problème au niveau du sens. On est considéré mineur alors que l'âge n'est pas valide du tout. Intégrez donc toujours un mécanisme de vérification de la validité des valeurs, et lorsqu'il y a un problème, définissez un comportement particulier.

```
Dim age As Integer
age = -1
If (age < 0 Or age > 130) Then
  MessageBox.Show("L'âge donné n'est pas valide")
ElseIf (age < 18) Then
    MessageBox.Show("Vous êtes mineur")
Else
    MessageBox.Show("Vous êtes majeur")
EndIf
```

Idem pour les exceptions. Travaillez avec des exceptions sur plusieurs niveaux, de la plus spécialisée à la plus générale, et n'omettez pas les cas d'erreur par défaut :

```
Try
  Dim resultat As Integer
  resultat = dividende \ diviseur
  MessageBox.Show("Résultat = " & resultat)
Catch exc As System.DivideByZeroException
  MessageBox.Show(exc.Message)
  MessageBox.Show(exc.StackTrace)
  diviseur = 1
Catch exc As System.IllegalArgumentsException
  MessageBox.Show(exc.Message)
  MessageBox.Show(exc.StackTrace)
  MessageBox.Show("Les arguments ne sont pas valables")
Catch exc As System.Exception
  MessageBox.Show("Il y a eu une exception d'ordre général")
End Try
```

Les recommandations peuvent être nombreuses, par exemple toujours écrire une seule classe dans un même fichier. Mais si vous appliquez celles-ci à la lettre, vos programmes seront fiables et agréables à utiliser. Par ailleurs, il vous sera plus facile de vous replonger dans votre code après de longs temps d'interruption.

9.6. Bien dissocier les parties de votre programme

Depuis le début, nous sommes partis du principe que vous étiez le seul à créer un projet. Chacun des programmes et des exemples qui ont été présentés était totalement indépendant et fournissait une réponse à un besoin, sans aucune interaction avec un quelconque élément externe.

Or, dans votre vie de programmeur, ce ne sera certainement pas toujours le cas. C'est pourquoi il faut avoir à l'esprit que vos travaux pourront être utilisés par d'autres. Il faut donc prendre des mesures et vous arranger pour rendre vos programmes le plus réutilisables possible.

Pour développer ce point, nous allons voir une application indépendante qui, même si elle répond à une demande de manière efficace, peut provoquer un problème au moment d'interagir avec un autre

programme. C'est pourquoi nous verrons dans un second temps comment modifier ce même programme, de manière à le rendre réutilisable.

C'est dans cet esprit qu'il vous faudra créer vos futurs programmes, car cela n'enlève en rien les possibilités que vous avez précédemment produites, mais en les rendant réutilisables vous leur donnez une valeur ajoutée qui sera grandement appréciée par vos collaborateurs.

Une application indépendante

Pour montrer les limites d'une application indépendante où tout est codé dans le même projet, créons une application complète dans un unique projet.

Imaginons une application qui donne les futurs résultats du loto. Vous conviendrez que beaucoup de monde peut être attiré par cette perspective. Malheureusement, la formule magique n'existant pas, nous allons créer un générateur de six nombres aléatoires. Avec un peu de chance, parmi les essais que vous ferez, vous trouverez les bons.

1. Créez un nouveau projet Application Windows.

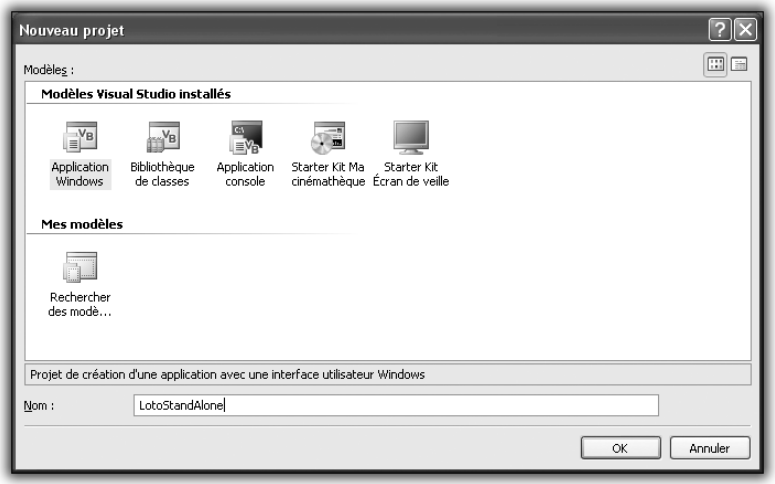

Figure 9.46 : *Nouvelle application Windows*

2. Créez maintenant une petite interface permettant de visualiser vos résultats.

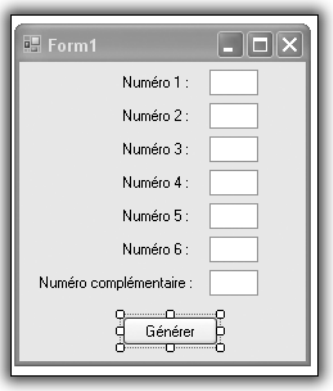

Figure 9.47 :
Création de l'interface

Créons ensuite le code principal de votre programme. Nous allons lui faire créer sept numéros entre 1 et 49 que nous afficherons dans cette interface.

Générer un nombre aléatoire est très facile grâce au framework .Net. Il suffit d'instancier un objet de la classe `Random` et de faire appel à la méthode `Next`. Cette dernière prend deux entiers en paramètres correspondant aux limites basse et haute. Dans notre cas, ces paramètres sont 1 et 49, les limites du Loto.

3 Double-cliquez sur le bouton afin que Visual Studio crée la méthode destinée à gérer l'événement du `Click`. Vous pourrez ensuite y mettre votre méthode.

```
 Private Sub Button1_Click(ByVal sender As System
✂ .Object, _
       ByVal e As System.EventArgs) Handles
     ✂ Button1.Click
    Dim generator As New Random()

    Dim number As Integer = generator.Next(1, 49)
    TextBox1.Text = number
    number = generator.Next(1, 49)
    TextBox2.Text = number
    number = generator.Next(1, 49)
   TextBox3.Text = number
    number = generator.Next(1, 49)
    TextBox4.Text = number
    number = generator.Next(1, 49)
    TextBox5.Text = number
    number = generator.Next(1, 49)
    TextBox6.Text = number
    number = generator.Next(1, 49)
```

```
            TextBox7.Text = number
        End Sub
```

Vous avez maintenant un programme capable de vous donner un tirage du loto. Si un nombre apparaît plusieurs fois, cliquez de nouveau pour avoir un nouveau tirage.

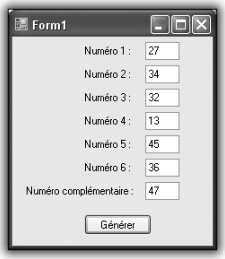

Figure 9.48 :
Tirage du loto

De mon côté, j'ai également une application. Pour l'instant, elle ne fait pas grand-chose, mais je ne désespère pas de pouvoir l'enrichir.

```
Module Module1

    Sub Main()
        Console.WriteLine("Bonjour le monde.")
        Console.ReadLine()
    End Sub

End Module
```

À ce stade, tout cela n'est pas d'une grande utilité. J'y inclurais bien la possibilité de générer un tirage du loto. De cette manière, quelqu'un qui utiliserait mon application pourrait voir "*Bonjour le monde.*" mais également voir le tirage du loto, qui pourrait lui amener la richesse.

Cela tombe bien, vous avez réalisé une application capable de faire ceci.

Quel malheur, c'est une application indépendante et je ne peux donc pas l'intégrer à mon projet personnel. Mon application est condamnée à afficher simplement "*Bonjour le monde.*".

La même application réutilisable

La clé pour pouvoir réutiliser une application est déjà de bien discerner les grandes parties qui la composent, et comment elle pourrait être divisée.

La vôtre pourrait se diviser comme ceci :

1. La partie qui génère les nombres.
2. La partie qui les affiche.

Pour l'instant, tout est au même endroit, et c'est cela qu'il faudrait modifier pour rendre cette application réutilisable.

Le générateur de tirage

Nous avons identifié comme une grande partie de votre programme celle qui fait le calcul des tirages.

Pour l'instant, il n'est aucunement question d'affichage ici, donc, une bibliothèque de classe nous permet de gérer cette partie.

1. Créons donc un nouveau projet bibliothèque de classe.

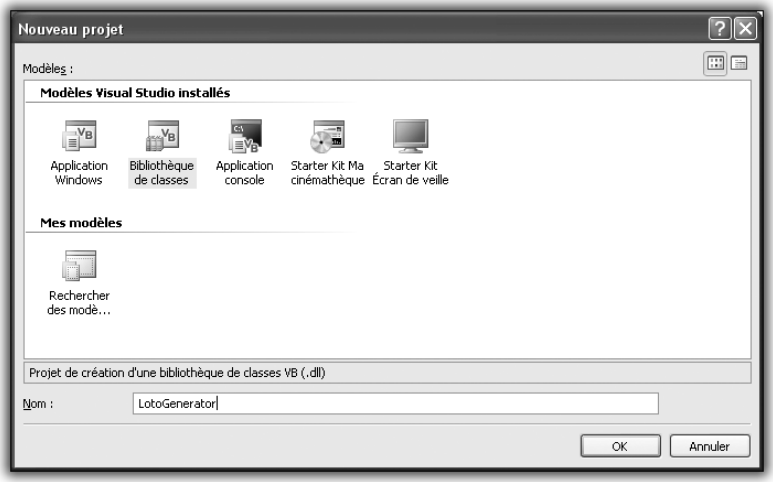

Figure 9.49 : *Nouvelle bibliothèque de classe*

Il nous faudra maintenant remplir cette bibliothèque avec de nouvelles classes.

2. Créons donc une classe qui va générer nos nombres.

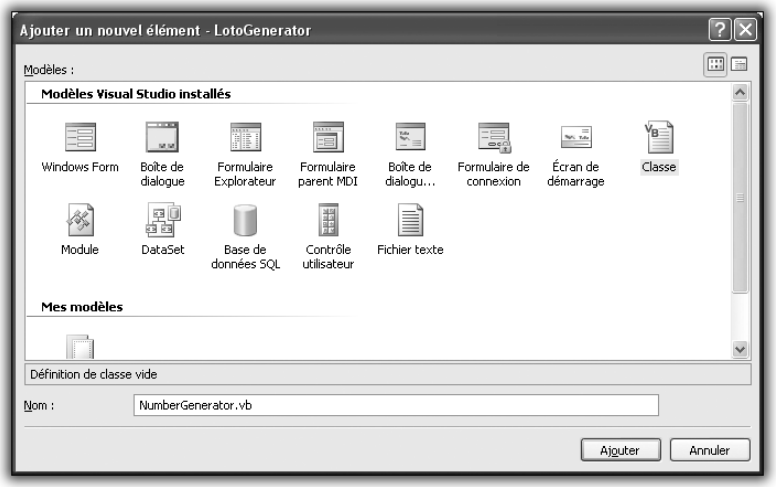

Figure 9.50 : *Nouvelle classe NumberGenerator*

Puis il va falloir remplir cette classe afin qu'elle fasse bien le traitement que l'on veut. C'est là qu'on doit faire preuve d'un peu de réflexion pour savoir comment gérer ceci.

Tout d'abord, posons-nous la question : "*De quoi a-t-on besoin ?*"

Réponse : rien ! En effet, nous avons toutes les informations nécessaires. Notre but étant simplement de générer sept nombres compris entre 1 et 49. Nous n'aurons donc pas d'attributs membres.

Autre question, maintenant : "*Que va-t-on faire ?*"

Notre but est de générer sept nombres que l'on va pouvoir utiliser. Faisons donc une méthode qui renvoie une liste de nombres. Ce qui amène la question suivante : "*Quels seront les paramètres de cette méthode ?*"

Là encore, aucun. Nous savons que ces nombres seront compris entre 1 et 49.

Dernière question pour compléter la création de la méthode : "*A-t-on besoin d'une instance particulière ?*" Réponse : non. Cette méthode étant totalement utilitaire, elle ne possède aucune information pertinente que l'on ait envie de garder. Dans ce cas, on peut la marquer `Shared`, ce qui nous permettra d'accéder à la méthode sans avoir besoin d'instancier un objet de type `NumberGenerator`.

Ce qui nous donne la signature suivante pour cette méthode :

```
 Public Shared Function GenerateNumbers() As List(Of
✂ Integer)

 End Function
```

Classes et méthodes Public

Pour être utilisées par d'autres, la classe et la méthode devront être `Public`.

Nous avons maintenant fait le plus difficile. Le remplissage de la méthode est alors basique. Il vous suffit de créer le résultat et de le remplir en utilisant ce que vous avez fait précédemment dans l'autre programme.

```
 Public Shared Function GenerateNumbers() As List(Of
✂ Integer)
     Dim result As List(Of Integer) = New List(Of
    ✂ Integer)(7)
     Dim i As Integer
     Dim generator As Random = New Random()
     For i = 1 To 7
         result.Add(generator.Next(1, 49))
     Next
     Return result
 End Function
```

Parfait, nous venons de terminer la première partie de votre programme. La compilation de ce projet permettra de générer une DLL qui contiendra la classe `NumberGenerator`, qui expose une méthode `GenerateNumbers`.

Occupons-nous maintenant de la seconde partie, l'affichage. Comme nous ne voulons pas modifier notre bibliothèque de classe, ajoutons un projet "*Application Windows*" qui se chargera d'afficher les résultats obtenus depuis notre classe `NumberGenerator`.

En fait, reprenons l'ancien projet : `LotoStandAlone`. Nous gardons l'interface, mais supprimons tous les traitements qui sont faits.

```
 Private Sub Button1_Click(ByVal sender As System
✂ .Object, _
    ByVal e As System.EventArgs) Handles Button1.Click

 End Sub
```

Or c'est maintenant notre DLL `LotoGenerator` qui fait tous les traitements. Il faudra donc l'inclure au projet.

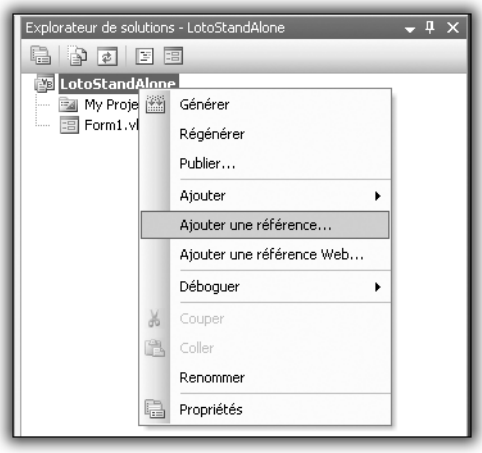

Figure 9.51 : *Nouvelle référence*

Cherchez la DLL, qui doit être dans le dossier d'exécution du projet `LotoGenerator`.

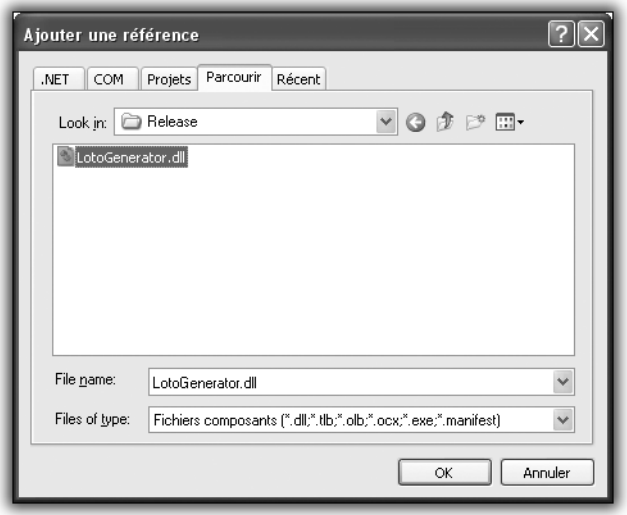

Figure 9.52 : *Inclusion de la DLL*

Pour pouvoir utiliser les classes de la DLL, il vous faut inclure son *namespace*.

```
Imports LotoGenerator
```

Maintenant, il faut appeler la méthode `GenerateNumbers` et adapter l'affichage à partir du résultat.

```
 Private Sub Button1_Click(ByVal sender As System
✂ .Object, ByVal e As System.EventArgs) Handles Button1
✂ .Click
     Dim tirage As List(Of Integer) = NumberGenerator
    ✂ .GenerateNumbers()
     TextBox1.Text = tirage.Item(0)
     TextBox2.Text = tirage.Item(1)
     TextBox3.Text = tirage.Item(2)
     TextBox4.Text = tirage.Item(3)
     TextBox5.Text = tirage.Item(4)
     TextBox6.Text = tirage.Item(5)
     TextBox7.Text = tirage.Item(6)
End Sub
```

Vous remarquez qu'on distingue directement les deux parties du programme. Une instruction fait la récupération des nombres, le reste des instructions sert à afficher. Et, finalement, l'application est exactement la même.

Figure 9.53 :
Application finale identique

De votre point de vue, cela n'a effectivement rien changé. Maintenant, en ce qui me concerne, je peux enrichir mon application qui faisait "Bonjour le monde.".

Il me suffit d'ajouter votre DLL à mon projet et d'ajouter le *namespace*, comme vous l'avez fait dans la partie affichage. Et je peux maintenant utiliser votre méthode de génération de nombres. Puis j'adapte mon propre affichage pour qu'il soit en accord avec mon application :

```
Sub Main()
    Console.WriteLine("Bonjour le monde.")
```

```
        Dim tirage As List(Of Integer) = NumberGenerator
       ✂ .GenerateNumbers()
        Console.WriteLine(tirage.Item(0))
        Console.WriteLine(tirage.Item(1))
        Console.WriteLine(tirage.Item(2))
        Console.WriteLine(tirage.Item(3))
        Console.WriteLine(tirage.Item(4))
        Console.WriteLine(tirage.Item(5))
        Console.WriteLine(tirage.Item(6))
        Console.ReadLine()
    End Sub
```

Figure 9.54 : *Ma version du tirage du loto*

Et, grâce à votre DLL, j'ai pu enrichir mon application. De votre côté, vous avez rendu votre travail réutilisable, ce qui lui a donné une énorme valeur ajoutée. Si vous aviez gardé la version d'origine, où tout est fait au même endroit (traitement + affichage), je n'aurais pu l'utiliser, et je ne l'aurais certainement jamais fait. Alors qu'en la découpant comme il faut, en la divisant en deux parties distinctes et dissociées, vous m'avez permis de l'utiliser dans une application totalement extérieure à votre projet.

Cette capacité à bien discerner les parties d'un projet demande de l'expérience et de la pratique. Il n'est pas intuitif de faire un découpage qui n'a pas lieu d'être au premier abord, surtout dans une optique où vous avez le contrôle total de votre application. Qui plus est, cela demande plus de temps, car ce sont autant d'éléments à prendre en compte, en plus de ceux qui existent déjà. Mais, pour deux applications

qui sont identiques visuellement, la réutilisabilité peut faire la différence entre la très bonne application et celle qui est bien, tout simplement.

9.7. Utiliser de bons outils pour de bons résultats

Pour réaliser un bon programme, produire du code de bonne qualité est une chose, mais ce n'est pas l'unique qualité que peut avoir votre programme. La capacité à suivre son déroulement ou encore la manière dont il est documenté sont autant de points positifs pour faire un bon programme.

En effet, vous pouvez avoir fait le meilleur code du monde, si vous êtes le seul à pouvoir le lire il y a de fortes chances qu'il tombe aux oubliettes. Si tout le temps qu'est censé faire gagner votre programme doit être utilisé pour le comprendre ou pour analyser son comportement en cas de problème, son intérêt est totalement perdu.

Cependant, il ne vous est pas forcément nécessaire de vous occuper de toutes ces étapes vous-même. Fort heureusement, d'autres l'ont fait avant vous et, honnêtement, il serait dommage de ne pas en profiter.

Parmi ces outils, nous allons nous attarder sur quelques outils de référence qui vous feront gagner un temps précieux et amélioreront fortement la qualité de votre projet, sans que vous ayez pour autant beaucoup de code à produire.

Logger avec log4net

Dans la partie des gestions d'erreurs, nous avons vu comment créer des journaux d'événements qui permettent de garder une trace d'un comportement particulier de votre programme.

Reportez-vous à ce sujet au chapitre ***Création de journaux d'événements.***

Or il n'y a pas que les erreurs qui sont bonnes à être enregistrées. Avoir une vue directe sur les arguments passés à vos méthodes ou encore savoir quelles méthodes votre programme exécute sont autant

d'informations essentielles qui vous permettront de comprendre rapidement ce qui s'est passé dans votre programme et donc de réagir en conséquence.

Un outil de référence nous permet de gérer ceci : Log4Net.

Présentation de l'outil

Log4Net est un outil qui vous permettra de gérer le déroulement de votre programme à travers des "logs", c'est-à-dire des fichiers textuels qui détailleront le comportement de votre application, un peu à la manière des journaux d'événements, qui permettent d'enregistrer les erreurs.

Un des aspects intéressants de cet outil est sa capacité à être configuré à l'exécution. De cette manière, vous n'avez pas à modifier votre programme ni à le recompiler avec des logs plus ou moins détaillés. Il offre de nombreuses possibilités telles les suivantes :

- **La compatibilité avec de nombreux frameworks**. Il est possible de l'exécuter avec la plupart des versions du framework .Net, même celle destinée aux téléphones portables, ou encore Mono, l'implémentation de .Net portable sur d'autres systèmes d'exploitation, comme Linux.
- **La multiplicité des sorties possibles**. Les informations recueillies par Log4Net peuvent vous être restituées sous forme de fichiers textuels, c'est le fonctionnement normal. Mais vous pouvez également les retranscrire sur la Console, ou encore envoyer des courriers électroniques, ou tout simplement les garder en mémoire…
- **La configuration dynamique**. Log4Net offre la possibilité d'être configuré à travers un fichier XML. De cette manière, sans modifier votre application, vous pouvez changer le niveau de log que vous voulez voir afficher, la sortie que vous souhaitez utiliser ou encore le chemin qui contiendra votre fichier de log.
- **Le contexte de log**. Outre les informations que vous avez explicitement demandées, Log4Net enregistre un certain nombre d'informations qui lui sont propres, comme l'heure à laquelle il a effectué le traitement ou encore le thread ayant appelé l'opération.
- **Une hiérarchie de criticité**. Nous sommes d'accord pour dire que toutes les informations n'ont pas la même importance. Savoir que

le programme est entré dans telle méthode est moins important qu'une erreur ayant surgi sans prévenir. Pour marquer cette importance, Log4Net définit plusieurs niveaux dans les logs :

- *Debug*. Ce sont les informations les moins importantes. On pourra y mettre les entrées et les sorties de méthodes ou encore des messages informatifs pour nous dire dans quelle partie du programme nous sommes passés.
- *Info*. Ce niveau de log correspond à une information pertinente. Cela pourra correspondre à une action importante ou encore à l'affichage des données utilisées dans notre méthode.
- *Warn*. Ce niveau marque un avertissement. Un avertissement n'est pas une erreur en soi, mais marque un état qui peut provoquer un comportement instable. Il n'y a pas lieu de s'affoler, mais il est quand même bon de savoir que quelque chose de troublant s'est passé.
- *Error*. C'est le dernier niveau de log utilisé, et le plus important. Il est généralement utilisé en cas d'erreur. En effet, étant donné qu'une erreur n'est jamais faite volontairement, il ne faut surtout pas la rater. En loggant vos erreurs avec ce niveau, vous êtes certain de les voir apparaître le cas échéant.

Après cette rapide présentation de l'outil, entrons dans le vif du sujet.

Utilisation dans vos applications

Pour pouvoir utiliser Log4Net, il faut tout d'abord vous le procurer. Ce projet est un projet Open Source, donc totalement ouvert à l'utilisation, et ceci gratuitement. Vous pouvez l'obtenir sur le site **http://logging.apache.org/log4net/index.html**.

Une fois que vous aurez récupéré le package de Log4Net, il vous faudra prendre la bonne version ; pour nous, la DLL compatible avec la version du framework .Net 2.0. Après avoir décompressé l'archive de Log4Net, vous prendrez la DLL `log4net.dll`, qui se trouve dans le dossier *bin/net/2.0/release*.

Pour pouvoir utiliser Log4Net dans vos projets dès maintenant, vous devez l'inclure dans vos applications. Pour cela, ajoutez la DLL dans les références de vos projets.

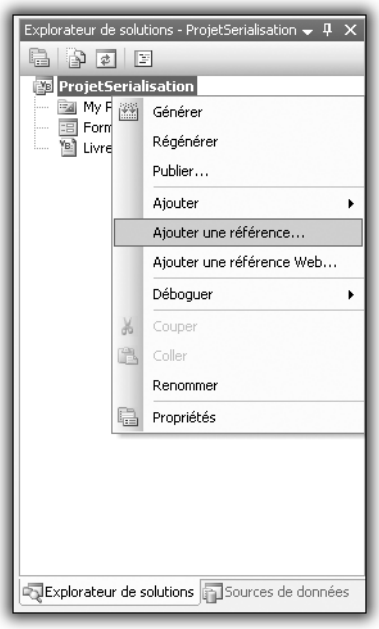

Figure 9.55 : *Ajout d'une référence*

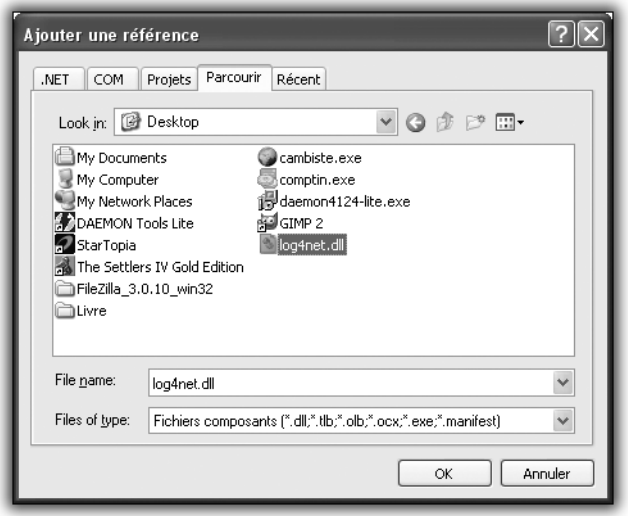

Figure 9.56 : *Chercher la DLL*

Une fois que la référence est ajoutée, dans chaque fichier de votre projet où vous voudrez utiliser Log4Net vous inclurez le namespace `log4net` et `Config`.

```
Imports log4net
Imports log4net.Config
```

Une fois toutes ces étapes réalisées, vous pourrez utiliser Log4Net dans vos projets.

À titre d'exemple, vous pouvez créer une application légère contenant cette méthode, en ayant bien effectué avant les opérations décrites ci-dessus.

```
Module Module1

    Sub Main()
        Dim log As ILog = LogManager
       ✂ .GetLogger(GetType(Module1))

        BasicConfigurator.Configure()
        log.Info("Entering application.")
        log.Info("Exiting application.")

        Console.WriteLine("Action finished.")
        Console.ReadLine()
    End Sub

End Module
```

Voici le résultat de ce bout de programme :

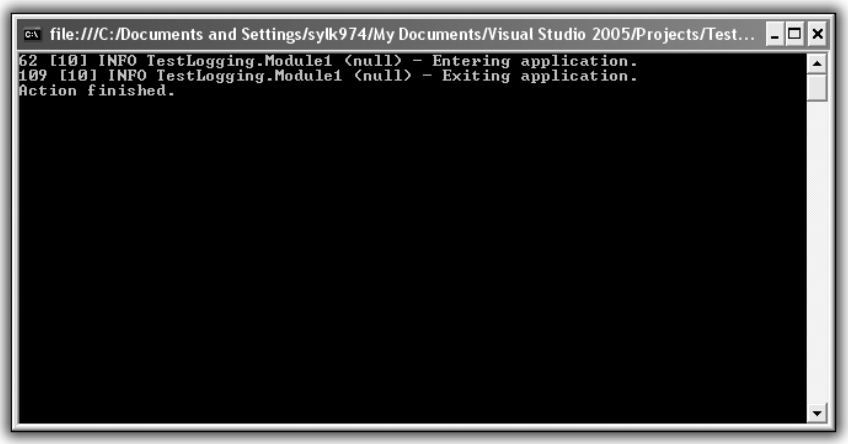

Figure 9.57 : *Premier programme utilisant Log4Net*

On ne dirait pas, mais dans ce bout de programme nous avons fait pas mal de choses.

```
Dim log As ILog = LogManager.GetLogger(GetType(Module1))
```

Cette ligne crée le `logger`. C'est avec cet objet que vous allez effectuer tous vos traitements de log. Le paramètre de la méthode `GetLogger` est le type de la classe ou du module utilisé. Ce paramètre n'est pas obligatoire, mais il est possible de faire apparaître cette information dans les logs, c'est pourquoi il est bon de la donner au moment d'instancier le `logger`.

```
BasicConfigurator.Configure()
```

En instanciant le `logger` sans faire d'autre action, celui-ci ne fera rien. En effet, pour que ses actions aient une répercussion visible par l'utilisateur, il faut le configurer en conséquence. Comme nous n'avons pas encore vu les détails concernant la configuration d'un `logger`, nous utilisons la configuration par défaut, accessible *via* la classe `BasicConfigurator`.

```
log.Info("Entering application.")
log.Info("Exiting application.")
```

Voici comment on utilise le `logger`. Il suffit tout simplement d'appeler la méthode correspondant au niveau de log que l'on veut donner et de passer en paramètre le message que l'on veut voir apparaître.

Et, comme nous l'avons vu après l'exécution de notre application, la configuration par défaut renvoie en fait les messages de log sur la `Console`, nous permettant ainsi de suivre ce qui se passe sans avoir à écrire d'instruction `Console.WriteLine`.

Dans la sortie de notre programme, vous avez pu remarquer que l'on trouve un certain nombre d'informations complémentaires qui nous sont fournies par Log4Net.

Nous trouvons entre autres le niveau de log utilisé (`INFO`) ou encore le module qui a exécuté l'action (`TestLogging.Module1`). Les premiers nombres sont des informations de temps et le thread dans lequel l'instruction a été faite.

Maintenant, nous allons créer une nouvelle méthode et placer le `logger` à un endroit accessible par toutes les méthodes.

```
Module Module1
    Dim log As ILog = LogManager
   ✂ .GetLogger(GetType(Module1))

    Sub HelloWorld()
```

```
        log.Debug("Entering Hello World method")
        Console.WriteLine("Hello world.")
        log.Debug("Leaving Hello World method")
    End Sub

    Sub Main()
        BasicConfigurator.Configure()
        log.Info("Entering application.")
        HelloWorld()
        log.Info("Exiting application.")

        Console.WriteLine("Action finished.")
        Console.ReadLine()
    End Sub

End Module
```

Voici maintenant ce que l'on peut observer :

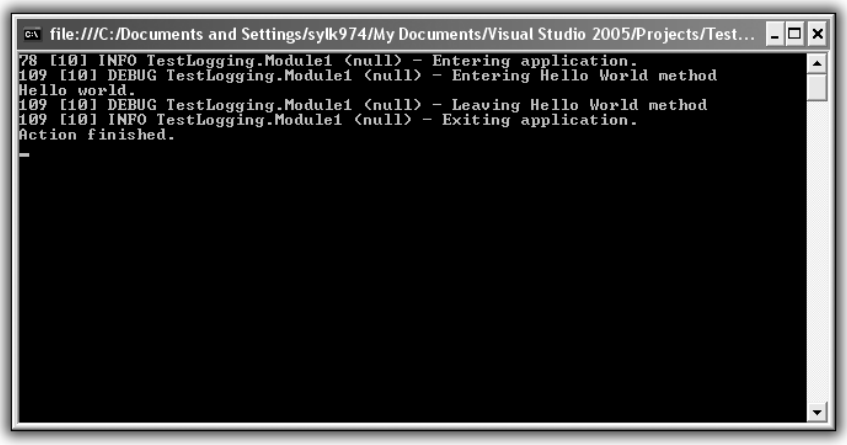

Figure 9.58 : *Bonjour le monde !*

Nous pouvons voir l'exécution de la méthode `HelloWorld`, ainsi que l'affichage de toutes les informations de log qui y sont associées. Nous pouvons en déduire que la configuration par défaut affiche les logs jusqu'au niveau le plus bas.

Très bien, nous pouvons maintenant enrichir nos applications d'informations pertinentes qui seront affichées sur la Console et qui nous permettront de suivre notre programme. En revanche, et je pense que vous serez d'accord avec moi, on s'y perd un peu. Notre message "*Hello World*", correspondant au réel traitement de notre méthode, passe

un peu inaperçu à cause de la pollution visuelle provoquée par tous ces messages de log.

Pour éviter ceci, nous allons configurer nous-mêmes le `logger`. La configuration d'un `logger` se fait en utilisant le format XML. Nous la ferons dans un fichier nommé *Logger.config*. Pour ce faire, nous allons remplacer le `BasicConfigurator` par un `XmlConfigurator` qui prend en paramètre un `FileInfo` lié à notre fichier *Logger.config.*

```
XmlConfigurator.Configure(New System.IO.FileInfo("Logger
✂ .config"))
```

Voyons maintenant comment écrire un fichier de configuration du `logger`.

Tout d'abord, il faut créer le squelette, c'est-à-dire une balise `log4net` qui contiendra toutes les informations du `logger`.

```
<log4net>
</log4net>
```

Comment configurer un `logger` ?

En fait, Log4Net fonctionne avec un système d'`Appender`.

Un `Appender` est une manière de restituer ce qui a été traité par le `logger`. Par exemple, l'écriture sur la Console correspond à un `Appender`, l'écriture dans un fichier correspond à un autre type d'`Appender`. Il faudra donc marquer dans le fichier de configuration tous les `Appender` que votre `logger` devra traiter. Il est défini par un nom, un type et un format de sortie. Pour les détails à propos des formats de sortie, je vous invite à chercher dans la documentation de Log4Net. Pour notre application, nous utiliserons le format de sortie par défaut.

```
  <appender name="A1" type="log4net.Appender
✂ .ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
     ✂ %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>
```

Maintenant que nous avons défini un `Appender`, il faut que nous précisions que le `logger` utilise cet `Appender`. En utilisant la balise `root`, nous définissons une configuration globale pour tous les `logger`.

Dans cette balise, il nous faut également définir le niveau minimal d'affichage des logs.

```
  <root>
    <level value="DEBUG" />
    <appender-ref ref="A1" />
  </root>
```

Voici donc notre fichier de configuration au complet :

```
<log4net>
  <appender name="A1" type="log4net.Appender
 ✂ .ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
     ✂ %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>
  <root>
    <level value="DEBUG" />
    <appender-ref ref="A1" />
  </root>
</log4net>
```

En exécutant de nouveau notre programme, nous retrouvons notre sortie d'origine sauf que, cette fois-ci, nous avons utilisé un fichier de configuration.

Tentons de changer le niveau de log, en mettant `INFO` à la place de `DEBUG`.

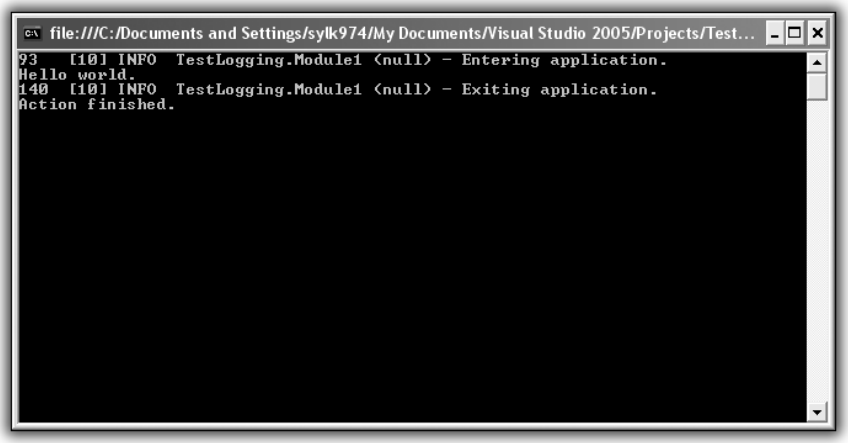

Figure 9.59 : *Niveau de log INFO*

Cela devient un peu plus lisible. Et nous avons pu faire ceci sans retoucher à notre programme.

Très bien mais, pour l'instant, nous affichons toujours les logs sur la Console. Nous allons maintenant tenter d'afficher nos logs dans un fichier.

Pour cela, il faut ajouter un `Appender` dans notre fichier de configuration : le `FileAppender`. Il faut également y définir le nom du fichier de sortie, le format de sortie, mais également si on écrase les anciens fichiers ou non. Voici ce que l'on doit ajouter à notre fichier de configuration :

```
  <appender name="FileAppender" type="log4net.Appender
✂ .FileAppender">
    <file value="log.txt" />
    <appendToFile value="false" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
     ✂ %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>
```

Nous avons ajouté notre nouvel `Appender`, il faut donc maintenant préciser dans `root` que l'on veut utiliser celui-ci :

```
  <root>
    <level value="DEBUG" />
    <appender-ref ref="FileAppender" />
  </root>
```

Le fichier au complet donne ceci :

```
<log4net>
  <appender name="A1" type="log4net.Appender
✂ .ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
       %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <appender name="FileAppender"
            type="log4net.Appender.FileAppender">
    <file value="log.txt" />
    <appendToFile value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp
         [%thread] %-5level %logger %ndc -
       ✂ %message%newline" />
```

```
  </layout>
  </appender>
  <root>
    <level value="DEBUG" />
    <appender-ref ref=" FileAppender " />
  </root>
</log4net>
```

Voici ce que l'on obtient en exécutant de nouveau notre programme :

Figure 9.60 : *Logging dans un fichier*

La Console n'est maintenant plus polluée par tous les messages de Log4Net.

A-t-on perdu toutes nos informations pour autant ? Que nenni ! Si nous allons dans le dossier de notre projet, nous pouvons remarquer la création d'un nouveau fichier *Log.txt* (comme le nom que nous avons donné à notre fichier de sortie). Et qu'y a-t-il dedans ?

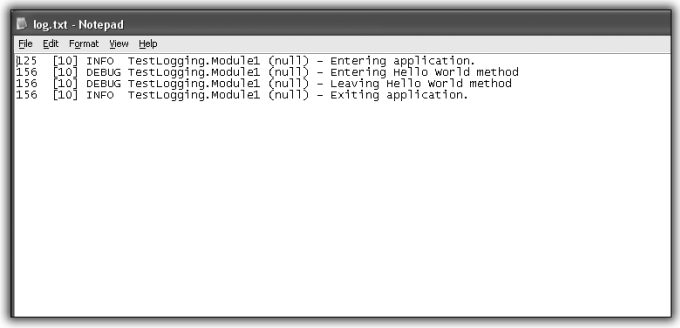

Figure 9.61 : *Fichier de sortie*

Il y a toutes nos informations de logs. Grâce à ce fichier de log, nous pouvons maintenant tracer *a posteriori* ce qui a été réalisé par notre programme.

Le `FileAppender` est la forme la plus simple d'`Appender` sur les fichiers. Or Log4Net possède un autre `Appender` pour les fichiers : le `RollingFileAppender`. Cet `Appender` permet d'utiliser plusieurs fichiers de logs de taille plus petite. En effet, selon votre niveau de logs et la manière dont vous les utilisez dans votre programme, un fichier de log peut très vite devenir énorme.

Essayons de répéter `HelloWorld` 100 000 fois.

```
Sub Main()
    XmlConfigurator.Configure(New System.IO.FileInfo _
        ("Logger.config"))
    log.Info("Entering application.")
    Dim i As Integer
    For i = 0 To 100000
        HelloWorld()
    Next
    log.Info("Exiting application.")

    Console.WriteLine("Action finished.")
    Console.ReadLine()
End Sub
```

Le programme prend un peu de temps, mais allons voir ce qui s'est passé du côté de notre fichier.

Il fait maintenant près de trente mégaoctets. C'est énorme ; si vous essayez de l'ouvrir avec un éditeur de texte, vous risquez d'avoir un peu de mal. Le `RollingFileAppender` permet de scinder un fichier de log en plusieurs fichiers plus légers.

Comme pour tous les `Appender`, il faut lui définir un nom et un format de sortie. Comme il s'agit d'un `Appender` sur fichiers, il faut également préciser le nom du fichier et si on écrase les anciens. De plus, pour le `RollingFileAppender`, il va falloir préciser la taille maximale d'un fichier et le nombre maximal de fichiers.

Taille des fichiers Log

Si l'ensemble des logs dépasse la taille maximale pour tous les fichiers, les dernières informations seront perdues.

```
<log4net>
  <appender name="A1" type="log4net.Appender.ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
         %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <appender name="FileAppender"
            type="log4net.Appender.FileAppender">
    <file value="log.txt" />
    <appendToFile value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
         %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <appender name="RollingFile"
            type="log4net.Appender.RollingFileAppender">
    <file value="log.txt" />
    <appendToFile value="true" />
    <maximumFileSize value="10000KB" />
    <maxSizeRollBackups value="2" />

    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
         %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <root>
    <level value="DEBUG" />

    <appender-ref ref="RollingFile" />
  </root>
</log4net>
```

Si vous ouvrez maintenant le dossier de votre projet, vous remarquez un fichier *Log.txt*, mais également un fichier *Log.txt.1*. Ce dernier fichier est en fait le premier fichier de log, qui étant devenu trop volumineux a laissé sa place à un autre. De cette manière, pour les analyser, vous pouvez regarder des fichiers d'une taille que vous aurez choisie.

Avec ce que nous avons vu au cours de ce chapitre, vous pouvez renforcer vos applications avec une sorte de mouchard capable de vous dire tout ce qu'il fait, quand il le fait, et avec quoi il l'a fait. Cependant, Log4Net expose infiniment plus de possibilités que celles que nous

avons vues. N'hésitez pas à regarder la documentation de Log4Net et à faire toutes sortes de tests et d'essais pour connaître son comportement.

Avoir une documentation professionnelle : Ndoc

Au risque de me répéter, j'aimerais encore appuyer sur le fait qu'il est important de bien commenter votre code et de le documenter, de manière qu'une autre personne qui utilise vos travaux puisse les comprendre. En effet, vous n'allez peut-être pas toujours être tout seul à travailler sur une application et, si le projet commence à être un peu important, on peut vite s'y perdre.

Reportez-vous à ce sujet au chapitre ***Quelques bonnes habitudes à prendre****.*

Admettons que vous veniez de terminer un travail : vous avez créé une DLL capable de faire des opérations mathématiques. Vous proposez à qui veut bien s'en servir la possibilité de faire des opérations basiques sur des nombres. Voici ce qu'une personne qui possède le code pourra voir :

```
 Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
     Return x + y
 End Function

 Public Function Substract(ByVal x As Integer, ByVal y
✂ As Integer) As Integer
     Return x - y
 End Function

 Public Function Times(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
     Return x * y
 End Function

 Public Function Divide(ByVal x As Double, ByVal y As
✂ Double) As Double
     Return x / y
 End Function
```

Quelqu'un qui ne connaît pas du tout le Visual Basic peut se retrouver perdu. Pour aider ces personnes, il est judicieux d'ajouter quelques commentaires qui permettront de mieux comprendre les méthodes.

```
 Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
     ' Retourne l'addition entre x et y
     ' Cette opération se fait sur nombres entiers
     Return x + y
 End Function

 Public Function Substract(ByVal x As Integer, ByVal y
✂ As Integer) As Integer
     ' Retourne la soustraction entre x et y
     ' Cette opération se fait sur nombres entiers
     Return x - y
 End Function

 Public Function Times(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
     ' Retourne la multiplication entre x et y
     ' Cette opération se fait sur nombres entiers
     Return x * y
 End Function

 Public Function Divide(ByVal x As Double, ByVal y As
✂ Double) As Double
     ' Retourne la division entre x et y
     ' Cette opération se fait sur nombres décimaux
     ' En effet, le résultat d'une division n'étant pas
    ✂ exact,
     ' il est préférable de travailler avec des Double
     Return x / y
 End Function
```

Cette étape est déjà pertinente, car de cette manière quelqu'un qui ne connaît pas le Visual Basic peut lire les commentaires et comprendre le résultat des méthodes. De plus, ces commentaires expliquent pourquoi l'addition, la multiplication et la soustraction travaillent sur des nombres entiers, alors que la division travaille avec des nombres décimaux.

Or, dans ce cas, il faut absolument que cette personne ait le code de votre application pour comprendre ce que vous avez voulu faire.

Il existe cependant un moyen de produire une documentation externe à votre programme, de manière à pouvoir l'exploiter sans posséder le code.

La documentation XML

Le framework .Net et Visual Studio permettent de produire cette documentation en utilisant le format XML. Il faudra alors définir pour chacune de vos méthodes une portion XML correspondant à votre documentation. Écrivez trois apostrophes ("’’’") au-dessus d’une de vos méthodes :

```
''' <summary>
'''
''' </summary>
''' <param name="x"></param>
''' <param name="y"></param>
''' <returns></returns>
''' <remarks></remarks>
Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
    ' Retourne l'addition entre x et y
    ' Cette opération se fait sur nombres entiers
    Return x + y
End Function
```

Visual Studio va ajouter automatiquement des balises XML vous permettant de décrire vos applications. Il ne vous restera alors plus qu’à la remplir.

- `Summary` donne une description générale de la méthode. Vous pourrez y décrire le traitement effectué par votre méthode.
- `Param` est la balise qui vous permettra de décrire les arguments de votre méthode. Le nom du paramètre est déjà marqué pour vous permettre de savoir quel argument vous êtes en train de décrire.
- `Returns` vous permet de décrire le résultat de votre méthode.
- Dans la balise `Remarks`, vous pouvez marquer tout autre commentaire que vous jugerez utile.

Il ne vous reste plus qu’à remplir ces balises.

```
''' <summary>
''' Cette méthode effectue une addition entre deux entiers
''' </summary>
''' <param name="x">Premier membre de l'opération</param>
''' <param name="y">Deuxième membre de l'opération</param>
''' <returns>Le résultat de l'addition, qui est une
✂ opération entière</returns>
''' <remarks>Nous avons fait une opération entière car
''' une addition entre 2 entiers sera toujours
✂ entière</remarks>
```

```
    Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
        ' Retourne l'addition entre x et y
        ' Cette opération se fait sur nombres entiers
        Return x + y
    End Function
```

Maintenant, si vous compilez votre projet, vous trouverez dans le dossier de sortie un fichier XML contenant votre documentation.

```
<member name="M:TestAppliBackgroundWorker.Form1.Add(System
✂ .Int32,System.Int32)">
  <summary>
 Cette méthode effectue une addition entre deux entiers
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de l'addition, qui est une
✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une addition entre 2 entiers sera toujours entière<
✂ /remarks>
</member>
```

Maintenant, même si le format XML n'est pas le plus lisible du monde par un être humain, vous possédez quand même un fichier indépendant de votre programme qui permettra à une personne extérieure de comprendre ce que vous avez voulu faire.

Vous devez maintenant renseigner toutes les opérations :

```
    ''' <summary>
    ''' Cette méthode effectue une addition entre deux entiers
    ''' </summary>
    ''' <param name="x">Premier membre de l'opération</param>
    ''' <param name="y">Deuxième membre de l'opération<
✂ /param>
    ''' <returns>Le résultat de l'addition, qui est une
✂ opération entière</returns>
    ''' <remarks>Nous avons fait une opération entière car
    ''' une addition entre 2 entiers sera toujours
✂ entière</remarks>
    Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
        ' Retourne l'addition entre x et y
        ' Cette opération se fait sur nombres entiers
        Return x + y
    End Function
```

```
    ''' <summary>
    ''' Cette méthode effectue une soustraction entre deux
✂ entiers
    ''' </summary>
    ''' <param name="x">Premier membre de l'opération<
✂ /param>
    ''' <param name="y">Deuxième membre de l'opération<
✂ /param>
    ''' <returns>Le résultat de la soustraction, qui est
✂ une opération entière</returns>
    ''' <remarks>Nous avons fait une opération entière car
    ''' une soustraction entre 2 entiers sera toujours
✂ entière
    ''' Cependant, le résultat peut être négatif si x est
✂ plus grand que y</remarks>
    Public Function Substract(ByVal x As Integer, ByVal y
✂ As Integer) As Integer
        ' Retourne la soustraction entre x et y
        ' Cette opération se fait sur nombres entiers
        Return x - y
    End Function
    ''' <summary>
    ''' Cette méthode effectue une multiplication entre
✂ deux entiers
    ''' </summary>
    ''' <param name="x">Premier membre de l'opération<
✂ /param>
    ''' <param name="y">Deuxième membre de l'opération<
✂ /param>
    ''' <returns>Le résultat de la multiplication, qui est
✂ une opération entière</returns>
    ''' <remarks>Nous avons fait une opération entière car
    ''' une multiplication entre 2 entiers sera toujours
✂ entière
    ''' Cependant, le résultat peut être négatif si un des
✂ nombres est négatif</remarks>
    Public Function Times(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
        ' Retourne la multiplication entre x et y
        ' Cette opération se fait sur nombres entiers
        Return x * y
    End Function
    ''' <summary>
    ''' Cette méthode effectue une division entre deux
✂ nombres décimaux
    ''' </summary>
    ''' <param name="x">Premier membre de l'opération<
✂ /param>
    ''' <param name="y">Deuxième membre de l'opération<
✂ /param>
```

```
    ''' <returns>Le résultat de la division, qui est une
   ✂ opération décimale</returns>
    ''' <remarks>Nous avons fait une opération décimale
   ✂ car
    ''' une division entre 2 nombres peut être décimale
    ''' De plus, le résultat peut être négatif si un des
   ✂ nombres est négatif</remarks>
     Public Function Divide(ByVal x As Double, ByVal y As
   ✂ Double) As Double
          ' Retourne la division entre x et y
          ' Cette opération se fait sur nombres décimaux
          ' En effet, le résultat d'une division n'étant pas
         ✂ exact,
          ' il est préférable de travailler avec des Double
          Return x / y
     End Function
```

Vous aurez maintenant un fichier de documentation XML complet à propos de votre projet.

```
<member name="M:TestAppliBackgroundWorker.Form1.Add(System
✂ .Int32,System.Int32)">
  <summary>
 Cette méthode effectue une addition entre deux entiers
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de l'addition, qui est une
 ✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une addition entre 2 entiers sera toujours entière<
 ✂ /remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .Substract(System.Int32,System.Int32)">
  <summary>
 Cette méthode effectue une soustraction entre deux
 ✂ entiers
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de la soustraction, qui est une
 ✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une soustraction entre 2 entiers sera toujours entière
 Cependant, le résultat peut être négatif si x est plus
 ✂ grand que y</remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .Times(System.Int32,System.Int32)">
  <summary>
 Cette méthode effectue une multiplication entre deux
 ✂ entiers
```

```
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de la multiplication, qui est une
 ✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une multiplication entre 2 entiers sera toujours entière
 Cependant, le résultat peut être négatif si un des
 ✂ nombres est négatif</remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .Divide(System.Double,System.Double)">
  <summary>
 Cette méthode effectue une division entre deux nombres
 ✂ décimaux
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de la division, qui est une
 ✂ opération décimale</returns>
  <remarks>Nous avons fait une opération décimale car
  une division entre 2 nombres peut être décimale
 De plus, le résultat peut être négatif si un des nombres
 ✂ est négatif</remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .BackgroundCount">
  <summary>
 Méthode de comptage utilisant un BackgroundWorker
 </summary>
  <remarks></remarks>
</member>
```

Bon, ce n'était déjà pas tellement lisible sur une méthode… Maintenant qu'il y en a quatre, ce n'est plus du tout possible de comprendre quoi que ce soit. Heureusement qu'il existe des outils capables d'analyser les fichiers de documentation XML pour en faire des fichiers plus lisibles pour un homme.

Le logiciel NDoc

Le logiciel de référence pour analyser les fichiers de documentation XML et les transformer en fichier d'aide est NDoc. À partir de votre DLL et du fichier de documentation XML qui a été généré, NDoc pourra vous fournir une documentation dans différents formats.

Déjà, préparons le projet.

1. Créez un projet de bibliothèque de classe que l'on appellera Calculatrice.

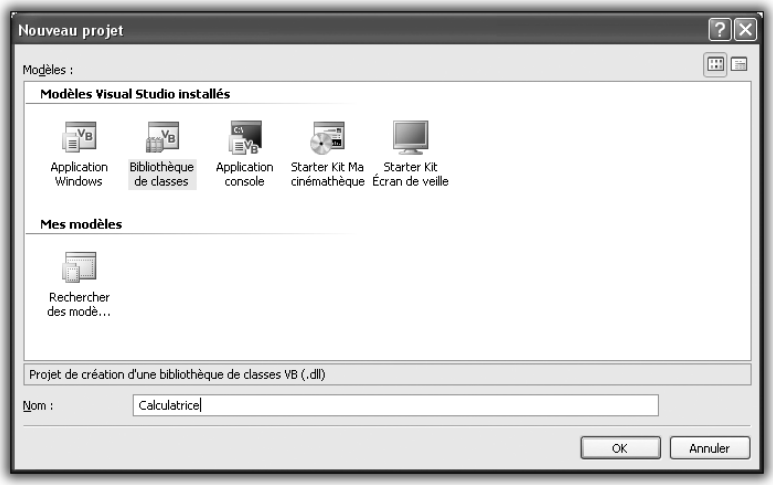

Figure 9.62 : *Nouvelle bibliothèque de classe*

2 Créez une nouvelle classe et mettez-y le code créé précédemment. Compilez. Vous devriez avoir plusieurs fichiers dans le répertoire de sortie.

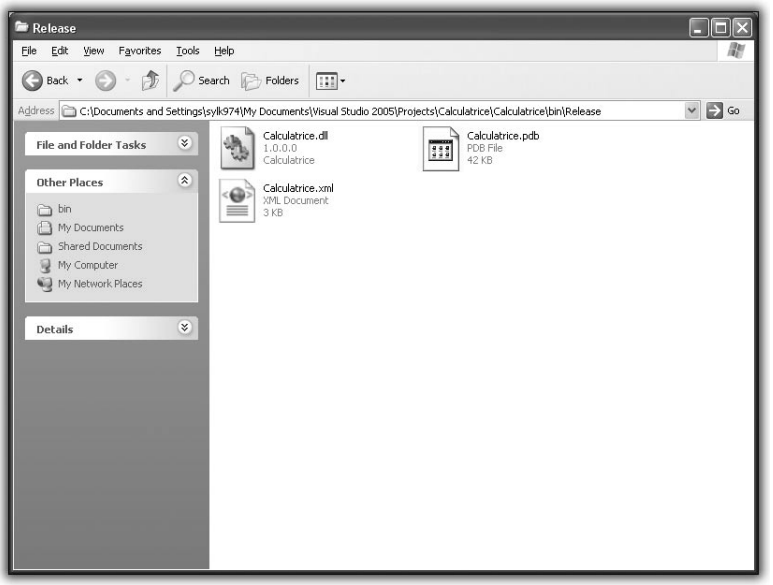

Figure 9.63 : *Fichiers générés par votre programme*

Laissons là un instant le code pour vous procurer NDoc. Le site officiel est **http://ndoc.sourceforge.net/**.

Vous pourrez y trouver de nombre d'aides et de détails à propos de cette application. Cependant, la version officielle de NDoc n'est pas compatible avec le framework .Net version 2.0. Fort heureusement, la communauté (NDoc est un projet OpenSource) a créé un portage pour la version 2 du framework .Net. En revanche, ce portage n'est pas officiel, et la version existante n'est pas stable. Mais elle permet quand même un nombre d'actions suffisant pour être intéressante. Vous pourrez l'obtenir sur le site **http://www.kynosarges.de/NDoc.html**.

NDoc nécessite une application de Microsoft : HTML Help Workshop. Il faudra vous la procurer avant de pouvoir utiliser NDoc. Elle est disponible à l'adresse **http://www.microsoft.com/downloads/details.aspx?FamilyID=00535334-c8a6-452f-9aa0-d597d16580cc&displaylang=en**.

Une fois NDoc téléchargé et décompressé, exécutez le fichier `Ndocgui.exe`.

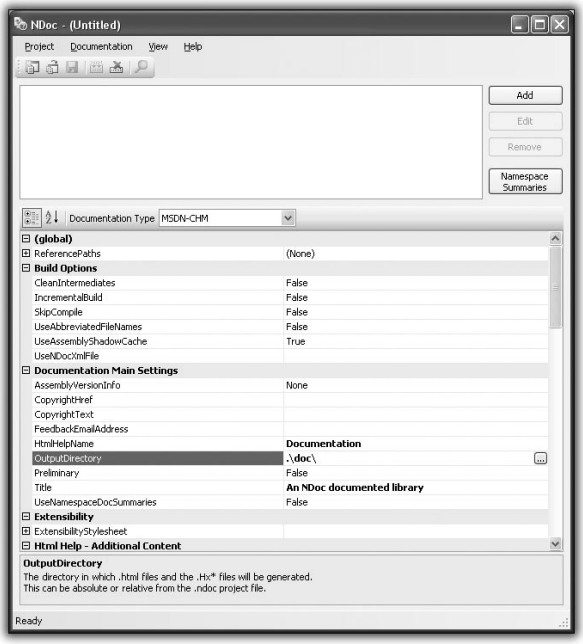

Figure 9.64 :
Lancement de NDoc

De nombreuses options sont configurables dans NDoc. Il n'est pas nécessaire de s'y attarder pour l'instant. Cependant, dans la liste

déroulante, sélectionnez *MSDN-CHM*. Vous aurez de cette manière un fichier *CHM* proche de la documentation MSDN de Microsoft.

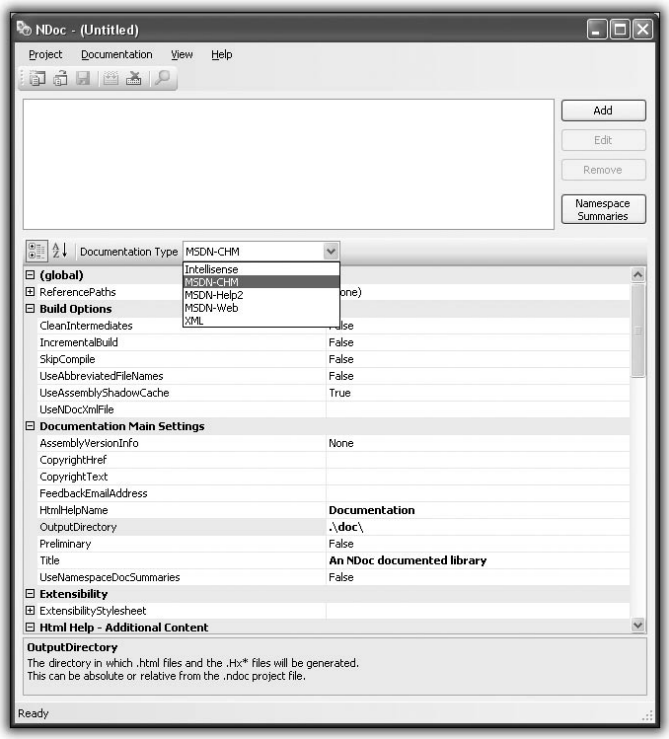

Figure 9.65 : *Sélection du type de documentation*

Maintenant, nous allons ajouter votre application pour que NDoc crée sa documentation. Pour cela, cliquez sur **Add**, et vous arriverez sur un menu où vous devrez sélectionner la DLL.

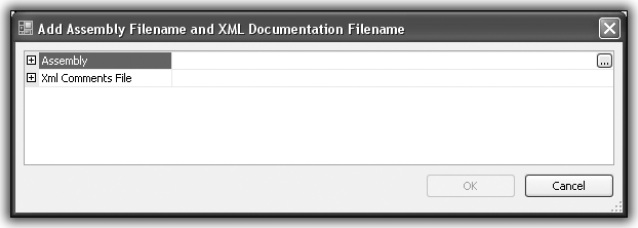

Figure 9.66 : *Sélection de la DLL*

En cliquant sur "**...**" vous pourrez choisir votre DLL *via* un explorateur de fichiers.

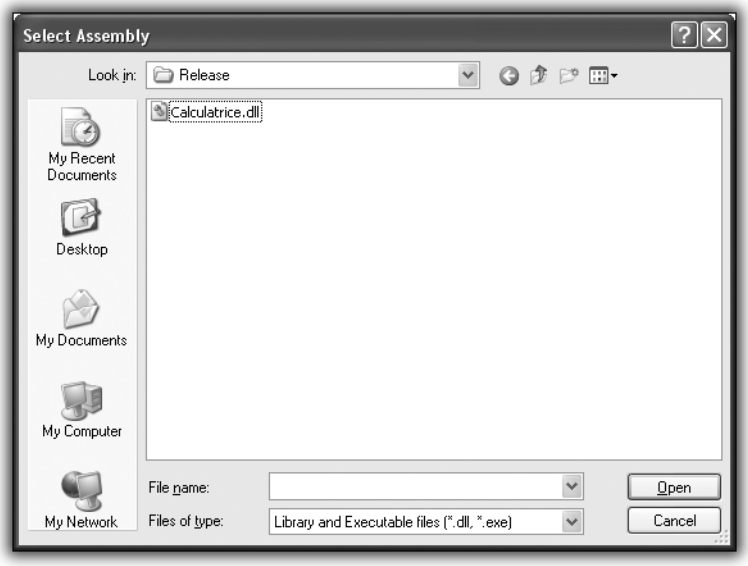

Figure 9.67 : *Exploration de fichiers*

Quand vous aurez sélectionné la DLL, NDoc cherchera automatiquement le fichier de documentation XML au même endroit. Faites OK pour valider l'ajout de votre DLL.

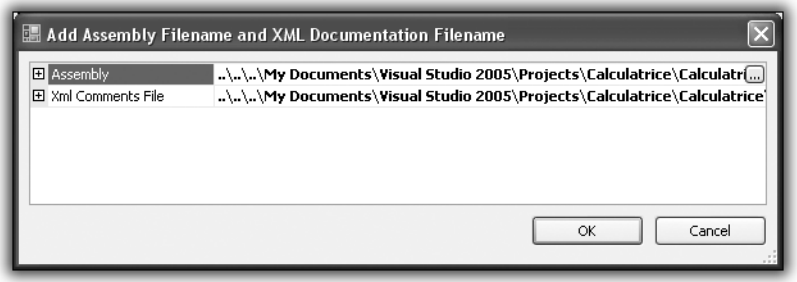

Figure 9.68 : *DLL + documentation XML*

Vous revenez maintenant sur l'écran principal de NDoc, avec votre DLL qui a été ajoutée à la liste des projets à documenter. Pour lancer la création de la documentation, il ne vous reste plus qu'à cliquer sur **Build docs**.

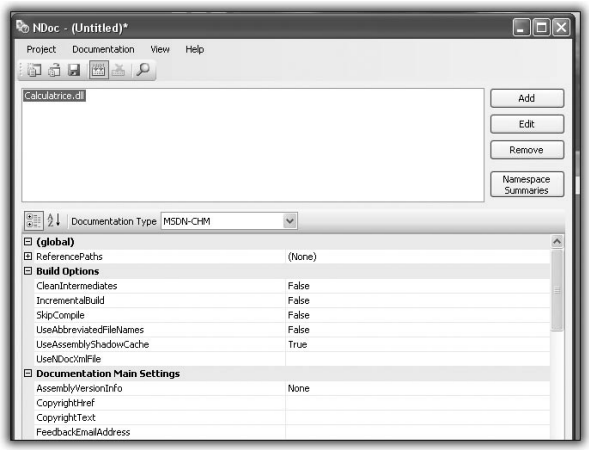

Figure 9.69 : *Lancement de la création de la documentation*

Vous pourrez voir le déroulement grâce à la *ProgressBar* de l'interface de NDoc, qui redeviendra normale une fois l'opération terminée.

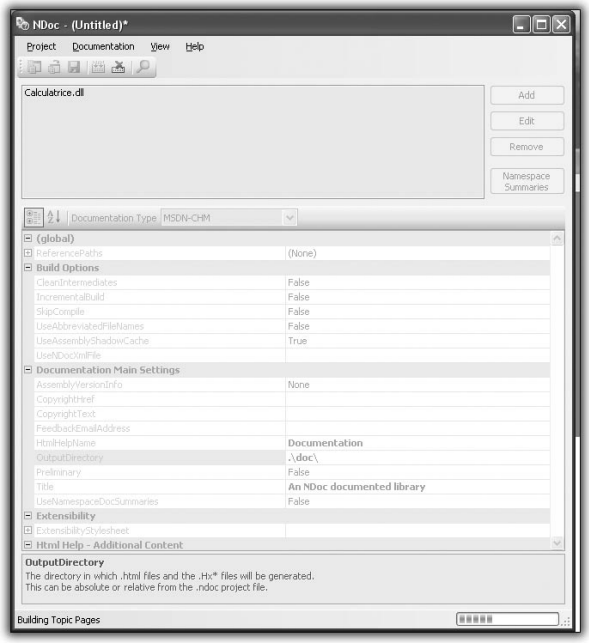

Figure 9.70 : *Progression du traitement*

La documentation générée se trouve dans le dossier *doc* de NDoc. Elle s'appelle *Documentation.chm*.

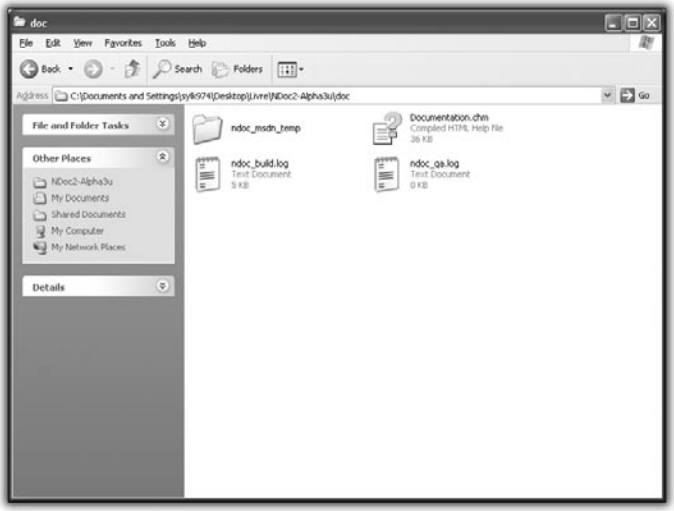

Figure 9.71 : *Fichier généré*

Maintenant, tout simplement en double-cliquant sur ce fichier, vous pouvez voir le résultat produit par NDoc.

Figure 9.72 : *Documentation finale*

Reconnaissez qu'elle a un aspect très propre et très professionnel. Pourtant, vous conviendrez que cette étape n'a pas été particulièrement difficile. Elle demande un peu de méthode et de méthodologie, un bon outil et un peu de bonne volonté. Vos utilisateurs et vos collaborateurs vous en remercieront car, grâce à cela, ils n'auront pas besoin de décortiquer votre code pour savoir ce que vous avez voulu faire (aussi tentés qu'ils soient, ce qui n'est pas toujours le cas). Merci, NDoc !

9.8. Garder l'interactivité avec l'utilisateur

Après les derniers chapitres que vous avez lus, il est certain que vous avez passé un certain nombre d'étapes dans la connaissance de la programmation. La programmation objet, les modèles de conception sont autant de concepts que vous avez appréhendés et qui marquent déjà une évolution. Mais la route de la connaissance est encore longue, et vous allez maintenant vous heurter à un des concepts les plus subtils de la programmation : le multithreading.

Le multithreading est en fait la capacité de diviser vos programmes en plusieurs sous-programmes réalisant chacun un certain nombre d'opérations en parallèle.

Mais, avant d'entrer dans le vif du sujet, il faut que vous connaissiez quelques notions de base pour comprendre la suite de ce chapitre.

Tout d'abord, qu'est-ce qu'un processus ?

Processus
Un processus est un programme, c'est-à-dire un élément qui exécute du code machine.

Il y a autant de processus que de programmes, et chacun d'eux peut faire quelque chose de différent. Un peut faire du calcul, l'autre, de l'affichage, un autre peut même être un jeu complet.

1. Appuyez sur Ctrl+Alt+Supp, et allez dans le gestionnaire des tâches de Windows. Puis allez dans l'onglet **Processus**. Vous verrez la liste des processus qui sont en ce moment exécutés sur votre ordinateur.

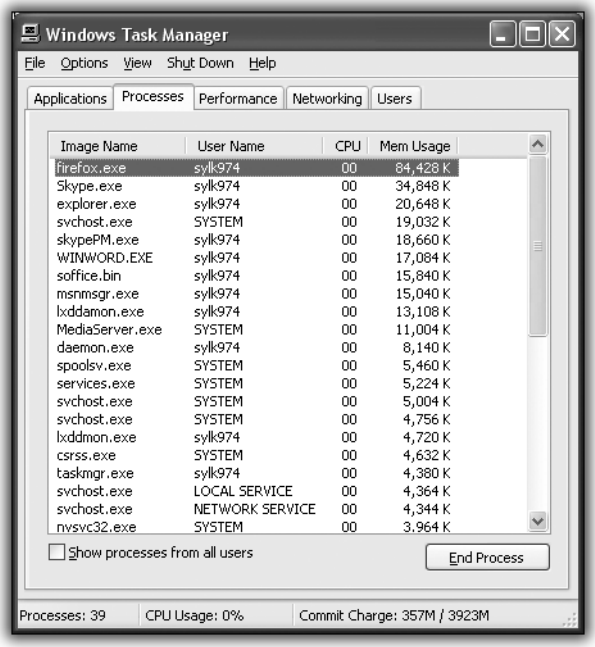

Figure 9.73 : *Processus exécuté sur votre ordinateur*

Comme vous pouvez le voir, énormément de programmes (et donc de processus) tournent en même temps sur votre ordinateur. Peut-être en connaissez-vous certains ?

Mais comment cela se peut-il ? On a vu qu'un programme était une liste d'instructions exécutées les unes après les autres. C'est effectivement le cas, sauf que votre système d'exploitation (ici Windows) leur fait exécuter leurs instructions à tour de rôle. Ils passent tous dans une file d'attente du système, puis chacun à son tour exécute un morceau de programme. C'est ensuite le tour d'un autre programme d'exécuter ses instructions, et ainsi de suite. Voici un schéma des états d'un processus qui sont chargés sur le système d'exploitation (voir Figure 9.74).

Le système d'exploitation va donc mettre un processus en sommeil, en activer un autre, le mettre en sommeil, en activer un autre, et ceci pour chacun des processus existants. Et, quand il a fini un tour complet, il recommence. Quel travail passionnant, me direz-vous...

Autre question qui se pose. Comment tous ces processus ne se marchent-ils pas sur les pieds ? Supposons qu'un de nos programmes modifie une de ses données. On vient de voir qu'il y avait un certain

nombre de processus qui tournaient en même temps sur notre machine. Or la mémoire de l'ordinateur est unique. Donc, rien n'empêche que tel programme ait une donnée commune avec tel autre.

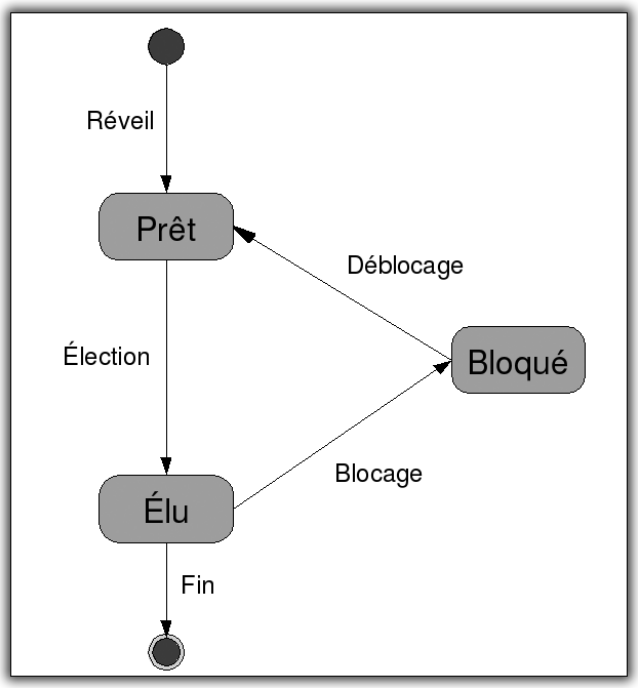

Figure 9.74 : *État d'un processus*

Là encore, c'est le rôle du système d'exploitation de gérer cela. Celui-ci va réserver un espace mémoire pour chacun des processus afin que celui-ci ait ses données propres et n'aille pas embêter les autres. Vous avez peut-être déjà eu l'occasion de voir une erreur qui disait `Access Violation`. Cela arrive quand un programme essaie d'accéder à une donnée ou à un espace mémoire qui ne lui appartient pas. Grâce à ce mécanisme, encore géré par le système d'exploitation (il en fait, des choses !), les modifications faites en mémoire par un programme ne se répercutent pas sur un autre. D'un autre côté, un programme ne pourra jamais voir directement les données d'un autre. De plus, étant donné que c'est le système d'exploitation qui décide de la quantité d'espace mémoire utilisée par un programme, vous pourriez avoir une erreur `OutOfMemoryException`, alors qu'il vous reste de l'espace en mémoire ou sur le disque dur.

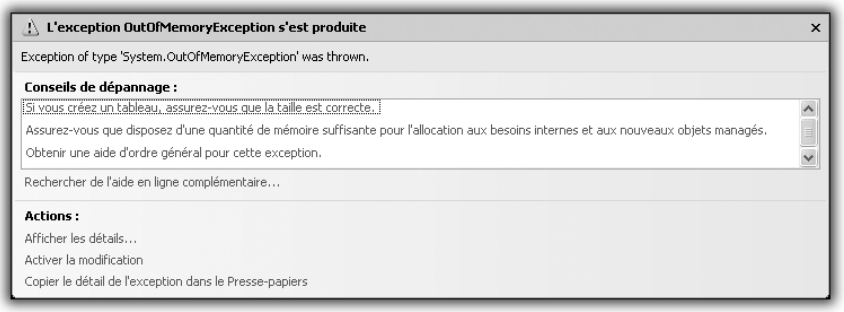

Figure 9.75 : *OutOfMemoryException : plus d'espace disponible pour votre processus*

On voit donc que le rôle du système d'exploitation est primordial pour la gestion de vos programmes.

Nous savons ce qu'est un processus ; maintenant, qu'est-ce qu'un thread ?

Thread

Un thread est également un processus, car il exécute des instructions. Cependant, il est lié à un processus principal dont il partage l'espace mémoire. On dit que les threads sont des processus légers.

Un thread est une sorte de sous-processus. Il vit à l'intérieur d'un processus et partage son espace mémoire, c'est-à-dire qu'il peut voir et agir sur les données de son processus père.

Au même titre qu'un processus, il est dans la file d'attente du système d'exploitation, qui le laissera à son tour exécuter un certain nombre d'instructions.

Maintenant que nous avons dégrossi les concepts de processus et de threads, nous allons voir ce qu'est le multithreading, quelle est son utilité, mais également quels problèmes cela peut poser. Nous verrons bien sûr comment répondre à ces problèmes. Comme je le disais en introduction de ce chapitre, le multithreading est un concept très très subtil, ainsi qu'une grosse source d'erreurs, même dans des applications professionnelles. À utiliser avec beaucoup d'attention !

Nous verrons ensuite un moyen simple de profiter du multithreading pour exécuter des instructions en tâche de fond, de manière à ne pas bloquer l'utilisateur.

Puis comment réagir sur l'interface graphique, de manière à pouvoir suivre ce qui est fait par vos threads. Et, enfin, comment gérer les erreurs dans les threads, car cela peut très vite devenir le bazar, et vous pouvez chercher des heures la source d'un problème qui est en fait dû à l'action d'un thread sur un autre qui a provoqué un état instable qui a fait planter votre programme. Du bonheur en perspective !

Introduction au multithreading et à ses problématiques

Maintenant que vous savez ce que sont les threads et les processus, vous vous demandez bien quel intérêt cela peut-il avoir. En ce qui concerne les processus, très peu. Il s'agit d'un programme, il exécute vos instructions, c'est très bien, mais vous ne pouvez pas faire grand-chose de plus avec.

Alors qu'un thread est un objet que vous pouvez contrôler dans vos programmes. Vous pouvez en instancier de nouveaux, les supprimer, les stopper, les démarrer, bref, les contrôler comme une donnée de vos programmes ! À ceci près qu'un thread est une donnée qui peut exécuter du code.

De ce fait, en instanciant plusieurs threads, vous pouvez exécuter du code en parallèle. On a vu que, comme le processus principal, un thread est un processus qui est dans la file d'attente des processus du système d'exploitation.

Imaginez qu'il n'y ait pas de thread supplémentaire dans votre processus. À chaque tour complet de la file d'attente du système d'exploitation, votre programme n'aura exécuté qu'un nombre d'instructions correspondant à une unité. Maintenant, si ce même processus utilise une centaine de threads, à chaque tour complet votre programme (c'est-à-dire le processus principal plus l'ensemble de ces threads fils) aura exécuté cent une unités d'instructions. Une pour chacun des cent threads plus une pour le processus principal.