# AP2 : Programmation évènementielle

## Sommaire

# 1. Découverte du langage et de l'EDI

## 1.1. Introduction

Le langage C# a été développé pour la plateforme .NET de Microsoft à partir de 2000 par l'équipe dirigée par Anders Hejlsberg (créateur de Turbo Pascal et de Delphi). Il a été normalisé par l'ECMA (ECMA-334) en décembre 2001 et par l'ISO/CEI (ISO/CEI 23270) en 2003. Les modifications survenues dans la Version 2.0 ont été normalisées par l'ECMA (ECMA-334) en juin 2006 et par l'ISO/CEI (ISO/IEC 23270:2006) en septembre 2006.

A priori destiné à l'environnement Windows, il a été porté sur divers environnement par le projet Mono (http://monodevelop.com/) supporté par Novell ou par le DotGNUProject (http://www.dotgnu.org/). Toutes ces implémentations bénéficient d'un environnement de développement (Environnement de Développement Intégré ou Integrated Development Environment), mais nous allons par la suite utiliser l'outil fourni par Microsoft, dans sa version Visual Studio 2008 (librement téléchargeable dans sa version Express) qui correspond à la version 3.5 de la plateforme .NET.



C# est un langage moderne, orienté objet, proche de Java et dont les caractéristiques s'enrichissent au cours des versions (version 4.0 aujourd'hui avec Visual Studio 2010, mais nous nous travaillerons sur la version 3.5). Nous n'en utiliserons pas toutes les caractéristiques.

L'EDI Visual Studio est un environnement performant qui permet de développer, compiler, tester et surtout déboguer (assez) facilement les programmes. Il pourra vous servir pour C#, mais aussi pour C++, VB, le développement Web ou pour tout langage s'appuyant sur l'architecture .NET.

Une documentation extensive est accessible en ligne sur le site msdn (http://msdn.microsoft.com). De nombreux sites fournissent des conseils, didacticiels et exemples de code, parmi lesquels on peut citer et (éventuellement) recommander le site http://www.codeproject.com/ ou le (début du) cours de C# proposé par http://tahe.developpez.com/dotnet/csharp/.

## 1.2. Eléments du langage C#

### i. Espace de nom

Tous les objets définis en C# le sont dans un espace de nom, ce qui permet de ne se pas s'inquiéter d'éventuels homonymes entre composants : on pourra toujours les différencier par un nommage complet. Pour simplifier la tâche du compilateur, les espaces de noms nécessaires au fichier de code courant doivent être appelés explicitement par une clause `using`.

### ii. Type de données

Les types simples suivants sont disponibles. Ils ont les propriétés et opérateurs usuels.

| | | | |
|---|---|---|---|
| char | caractère | 2 octets | caractère Unicode[1] (UTF-16) |
| string | chaîne | | chaine de caractères Unicode |
| int | nombre entier | 4 octets | $[-2^{31}, 2^{31}-1]$ |
| uint | .. | 4 octets | [0, 232-1] |
| long | | 8 octets | $[-2^{63}, 2^{63}-1]$ |
| ulong | | 8 octets | $[0, 2^{64}-1]$ |
| sbyte .. | Octet | 1 octet | $[-2^{7}, 2^{7}-1]$ [-128,+127] |
| byte | | 1 octet | $[0, 2^{8}-1]$ [0,255] |
| short | Entier | 2 octets | $[-2^{15}, 2^{15}-1]$ [-32768, 32767] |
| ushort | | 2 octets | $[0, 2^{16}-1]$ [0,65535] |
| float | nombre réel | 4 octets | $[1.5\ 10^{-45}, 3.4\ 10^{+38}]$ en valeur absolue |
| double | | 8 octets | $[-1.7\ 10^{+308}, 1.7\ 10^{+308}]$ en valeur absolue |
| decimal | nombre décimal | 16 octets | $[1.0\ 10^{-28}, 7.9\ 10^{+28}]$ en valeur absolue |

[1] Le codage ASCII des caractères sur 7 bits ne permet de coder que $2^7 = 128$ caractères ce qui est bien peu. Ses extensions comme ISO-8859-1 en utilisent 8 pour coder 256 caractères, ce qui reste largement insuffisant. Pour internationaliser les logiciels et l'informatique en général, le consortium Unicode (http://unicode.org/) a défini un ensemble de standards qui permettent d'associer un code unique à chaque caractère de chaque langue connue. La version 6.0 d'Unicode décrit plus de 100,000 caractères, dont plus de 70,000 pour la seule langue chinoise. Windows utilise la norme UTF-16 dans laquelle tous les caractères sont codés sur au moins 16 bits (2 octets), certains pouvant l'être sur 4 (caractères chinois rares par exemple).

| | | | |
|---|---|---|---|
| bool | Booléen | 1 octet | true, false |

Diverses fonctions prédéfinies permettent de réaliser des conversions de type en particulier entre nombres et chaines de caractères ou l'inverse :

```
int i = 5;
string cinq = i.ToString();
int u = int.Parse("1235");
int valeur;
if (int.TryParse("356a", out valeur)){
}
else
{
};
```

La fonction TryParse renvoie vrai si la conversion est possible.

### iii. Enumérations

C# permet de définir des types énumérés tel que :

```
public enum Etat { Ouvert, Fermé }
```

La bibliothèque de classes en contient un grand nombre et nous les utiliserons fréquemment. Par exemple :

```
Color couleur = Colors.Red ;
dlg.ShowDialog() = DialogResult.OK
```

### iv. Structures de contrôle

Les structures de contrôles usuelles sont disponibles et s'expriment à la manière C ou C++. Les commentaires sont comme (presque) partout signalés par // et /* */

Choix :

```
if (condition)
{
    // actions_condition_vraie;
}
else {
    // actions_condition_fausse;
}
```

Choix multiple :

```
switch(expression)
{
    case v1:
        // actions1;
    break;
    case v2:
        // actions2;
        break;
. .. .. .. .. ..
    default:
        // actions par défaut;
    break;
}
```

Boucle pour :

```
for (int i=debut; i<=fin; i++)
{
    // instructions;
}
```

Boucle pour tout :

```
foreach (Type variable in collection)
{
    // instructions;
}
```

Boucle tant que :

```
while(condition)
{
    // instructions;
}
```

Boucle jusqu'à :

```
Do
{
    // instructions;
}while(condition);
```

### v. Classes

Un programme C# est constitué de classes, qui possèdent des attributs (membres), des méthodes (fonctions), un ou plusieurs constructeurs et dans certains cas des événements (c'est le cas de toutes les classes utilisées pour construire une interface graphique, les « contrôles »). Une classe peut se référer à elle-même grâce au mot clé `this`.

```
public class Personne
{
    public string Nom;
    public string Prénom;
    public Person(){}
public Personne(string Nom, string Prénom)
{
        this.Nom = Nom;
        this.Prénom = Prénom;
    }
    public override string ToString()
    {
        return Nom + " " + Prénom;
    }
}
```



L'environnement .NET fournit un très grand nombre de classes prédéfinies qu'il est naturellement hors de question de lister ici. Nous en découvrirons un grand nombre au fur et à mesure.

### vi. Variables

Les déclarations de variables se font simplement par :

```
type nom_variable;
```

Lorsqu'il s'agit de membres de classes, on peut spécifier qu'une variable est `private`, `public`, `protected` ou `internal` (`private` par défaut).

Toute variable doit être initialisée avant d'être utilisée. Lorsqu'il s'agit d'un type simple, l'initialisation a la forme suivante :

```
int x = 0;
string a = "Hello world";
Color couleur = Color.Red;
```

Pour une variable « classe », elle se fait par un appel à `new` et à l'un des constructeurs de la classe :

```
Point p = new Point(3,4);
DateTime dt = new DateTime(2010, 10, 10, 10, 47, 34);
Personne pers = new Personne();
Personne pers2 = new Personne("Dupont","Jean");
OpenFileDialog opfd = new OpenFileDialog();
```

C# permet de regrouper très facilement des variables du même type, sous forme de tableaux ou (par exemple) de listes :

```
int[] entier = new int[3] { 1, 2, 3 };
List<string> chaines = new List<string>();
List<Personne> personnes = new List<Personne>();
```

Notons enfin que puisque C# (comme Windows dans son ensemble) utilise le codage Unicode des chaines de caractères, y compris pour les identificateurs, les déclarations suivantes sont parfaitement valides (le langage distingue les majuscules des minuscules, il est « case-sensitive ») :

```
string Prénom = "Pierre";
float المتغير = 1.0F ;
byte ชื่อ = 0x2c;
int 号 = 0;
```

### vii. Fonctions et appel de fonction

Les fonctions (qui sont toujours des membres d'une classe), renvoient une donnée d'un certain type (éventuellement `void`), et peuvent recevoir des paramètres, passés par valeur, par référence – mot clé `ref` - ou comme paramètre de sortie – mot clé `out` - (ne nécessitant pas d'initialisation).

Elles peuvent être surchargées comme dans la classe `Personne` ci-dessus.

### viii. Propriétés

La notion de propriété permet d'encapsuler les membres d'une classe (« accesseurs »), en séparant le membre qui reste `private` de la propriété qui est `public`.

La classe ci-dessous a un seul membre « `seconds` », privé, donc inaccessible directement. Il est cependant accessible par la propriété « `Seconds` » et ses accesseurs `get` et `set`. Elle a par contre une seconde propriété, qui repose sur le même membre, mais de façon indirecte, à travers un calcul, et seulement en lecture.

```
public class TimePeriod
{
    private double seconds;
public double Seconds
{
        get { return seconds; }
        set { seconds = value; }
    }
public double Hours
{
        get { return seconds / 3600; }
    }
}
```

L'EDI permet de transformer aisément une déclaration du type :

```
public double seconds;
```

en un couple déclaration privée/propriété, grâce au menu contextuel `Refactoriser|Encapsuler le champ`[2]. Dans la liste des constituants d'une classe (obtenue par la boîte `Propriétés` ou par la complétion automatique), variables et propriétés sont clairement distinguées.

### ix. Exceptions

Pour le traitement d'erreurs, C# offre un traitement des exceptions :

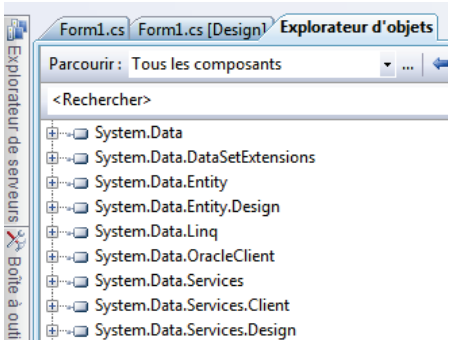
```
try
{
// code à exécuter
...
}
catch (Exception ex)
{
// traitement de l'erreur
...
}
finally
{
// à exécuter dans tous cas de figure
...
}
```

### x. Bibliothèques de classes

Les structures que nous venons de présenter permettent naturellement d'écrire tous les programmes imaginables, mais pas forcément de façon simple.

Comme tous les environnements actuels, la plateforme .NET propose une vaste bibliothèque de classes (*framework*) destinée à simplifier le développement et dont le contenu est présenté dans l'onglet `Explorateur d'objet`.

Cette bibliothèque est organisée selon un ensemble de hiérarchies issues « d'espaces de nommage », comme sur l'image ci-contre.



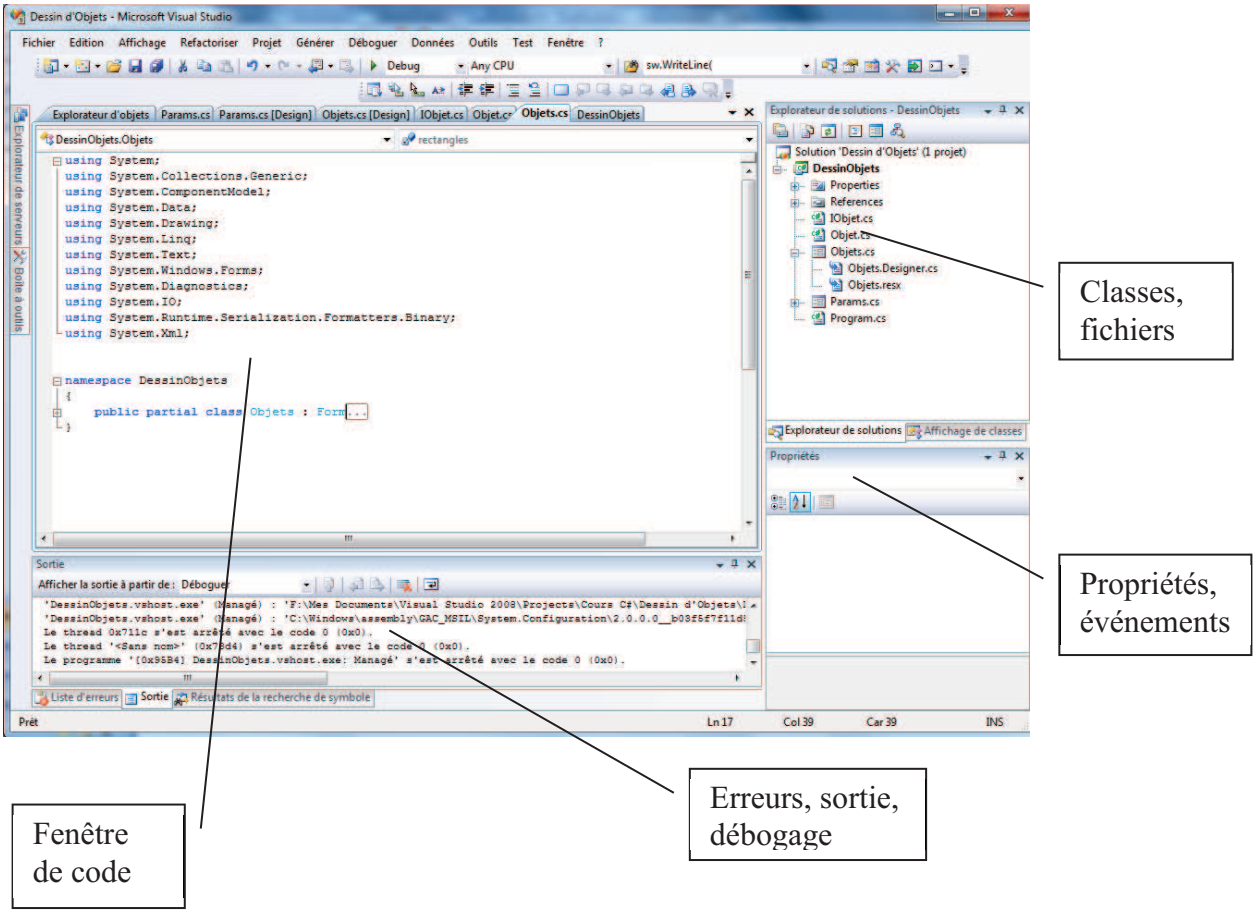

Pour être accessible dans le programme, ils doivent être explicitement appelés par une clause `using` (et éventuellement « référencés »).

[2] De la même façon, l'élément de menu « Refactorise|Extraire la méthode » permettra de transformer un bloc de code en appel de fonction.

## 1.3. Visual Studio 2010

L'environnement de développement intégré Visual Studio en est maintenant à sa version 2010, que nous utiliserons durant ce cours.

L'environnement est assez riche et offre de multiples fonctionnalités à travers ses fenêtres, menus, barres d'outils et menu contextuels. Il est largement personnalisable, mais avant de le personnaliser, il convient de s'assurer que l'on saura retrouver les éléments supprimés parce qu'on les croyait inutiles.

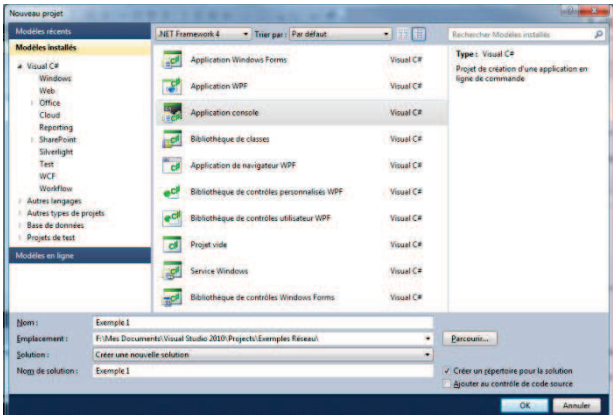


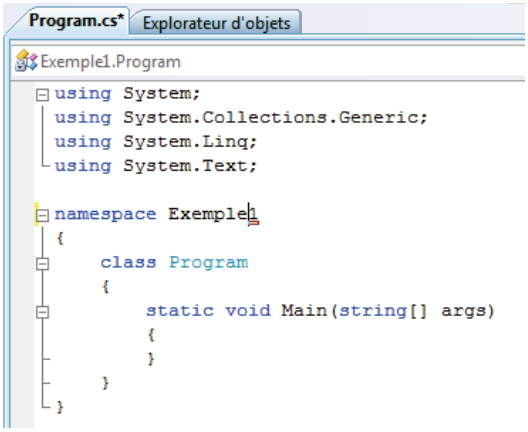
# 2. Premières Applications

## 2.1. Ecriture d'une application console

Dans le seul but de prendre en main l'environnement de programmation et le langage C#, commençons par créer une « application console » c'est-à-dire une application sans interface graphique. C'est la seule de ce type que nous créerons !

Après avoir lancé Visual Studio 2010, utilisons l'élément de menu `Fichier|Nouveau…|Projet` qui ouvre une fenêtre où nous choisissons de créer une « Application console » dont nous pouvons choisir le nom et l'emplacement dans les zones de texte au bas de la feuille, et que nous pouvons appeler Exemple1




Au bout d'un certain temps, une fenêtre de code s'ouvre, dans laquelle on note :

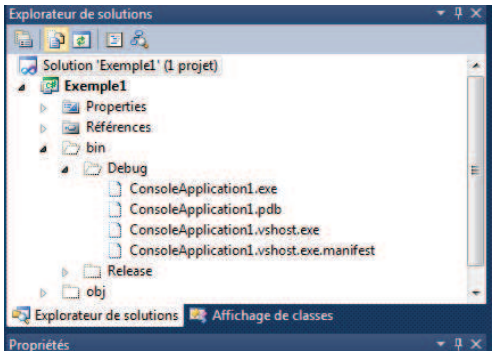
- la coloration syntaxique : les divers éléments apparaissent avec des couleurs différentes selon leur nature et un « problème de couleur » peut être le signe d'une erreur,
- l'indentation automatique : quand Visual Studio ne met pas correctement en forme votre code, c'est probablement qu'il y a un problème,
- les clauses `using`,
- la ligne `namespace` ,
- le mot-clé `class` : les programmes C# sont constitué d'un ensemble d'éléments de type class. Il en faut au moins une, la classe `Program` qui possède une seule fonction (ou *méthode*), `Main()`, qui sert de point d'entrée au programme,
- les symboles ⊟ ou ⊞ qui permettent de réduire ou développer par un simple clic des portions entières de code.

Saisissons maintenant une ligne de code, qui permet d'écrire du texte sur le « périphérique standard » :

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

namespace Exemple1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("1er exercice avec C#");
        }
    }
}
```

On aura ici remarqué la « complétion automatique » en cours saisie : Visual Studio ne propose a priori que ce qui est possible. S'il ne propose rien, il y a probablement un problème !

Le programme peut s'exécuter simplement par l'élément de menu `Déboguer|Exécuter sans débogage` (ou directement par Ctrl + F5). Une fenêtre console[3] s'ouvre et le programme s'exécute. On peut aussi l'exécuter depuis le sous répertoire `\bin\Debug`.




## 2.2. Fichiers générés

La création du programme et sa compilation ont généré un certain nombre de fichiers que l'on trouve dans le sous répertoire `\bin\Debug` et dont on peut faire apparaître la liste dans l'Explorateur de solutions, en cliquant sur le second bouton (`Afficher tous les fichiers`), comme le montre l'illustration ci-contre.



## 2.3. Débogage

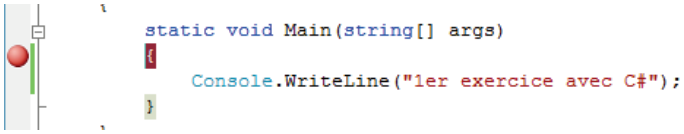
L'utilisation d'espions (ici des « `Console.Writeline` », plus tard des « `MessageBox.Show` ») pour déboguer une application reste évidemment possible dans avec Visual Studio. On peut aussi utiliser des fonctions de suivi d'exécution qui n'interrompent pas le déroulement du code, comme la fonction[4] :

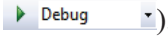
```
Trace.WriteLine(entier[1].ToString())
```

Les valeurs que l'on souhaite afficher apparaissent alors dans l'onglet Sortie du volet inférieur.

Mais nous disposons ici d'outils beaucoup plus élaborés avec un débogueur symbolique assez agréable à utiliser.

Pour le découvrir, posons un « point d'arrêt » en cliquant sur la barre verticale grise, au niveau de la ligne choisie (image ci-contre). Un marqueur rouge apparaît, qu'un nouveau clic au même endroit ferait disparaître.




On peut alors exécuter le programme en mode débogage grâce à l'élément de menu `Déboguer|Démarrer le débogage` (ou par F5 ou, encore plus simple, par un clic sur la flèche verte 

).

L'exécution s'arrête sur l'accolade ouvrante. Le mode pas-à-pas  (bouton du milieu) permet de progresser dans le code.

Pour expérimenter plus avant, on pourra saisir les lignes suivantes, dont le sens devrait être limpide (notons la coloration du code).

```
int[] entiers = new int[3] { 1, 2, 3 };
for (int i = 0; i < entiers.Length; i++)
    Console.WriteLine(i);
List<string>[5] chaines = new List<string>();
chaines.Add("Janvier");
chaines.Add("Février");
foreach (string s in chaines)
    Console.WriteLine(s);
```

Posons un point d'arrêt sur la première ligne de ce code, et démarrons le débogage. Apparaît alors, dans le volet inférieur de la fenêtre, à côté de « Liste d'erreurs », « Pile des appels », etc. un nouvel onglet « Espion 1 ». Dans la colonne « Nom », on peut saisir le nom de l'une des variables, par exemple « entiers » et voir s'afficher dans les autres colonnes, la valeur de cette variable, et son type (illustration ci-dessous).

---

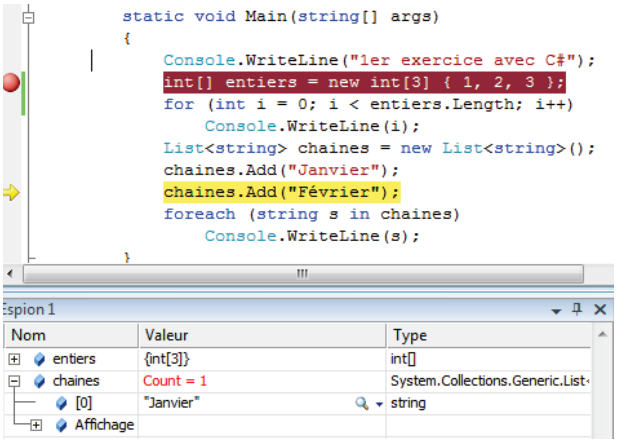
[3] Aussi appelée « Invite de commande » ou fenêtre MS-DOS, même si MS-DOS est enterré depuis bien longtemps.

[4] que l'on peut appeler si on a pris la précaution d'inclure la clause :

```
using System.Diagnostics;
```

[5] Le type `List<>` permet de fabriquer des listes d'objets de type divers (comme ici des chaînes). On peut ajouter des éléments par la méthode `Add`, s'y référer comme à des éléments d'un tableau. Il est inutile de connaître la taille de la liste à la création (contrairement à un tableau). La structure `foreach` permet de parcourir aisément une liste, comme on écrirait : pour tout x dans l ($\forall x \in l$).

Faisant de même avec la variable « chaines », on voit lors du pas à pas la liste se remplir progressivement. On peut aussi suivre les valeurs d'une variable en positionnant le curseur sur la variable.




Notons enfin, que l'on peut modifier le code durant le débogage, sans être obligé de recompiler et relancer le code[6]. Ceci permet de corriger assez rapidement les erreurs les plus simples (les modifications possibles sont malgré tout limités : on ne peut tout de même pas rajouter de membre à une classe, seulement modifier le code d'une méthode).



## 2.4. Ecriture d'une application Windows Forms

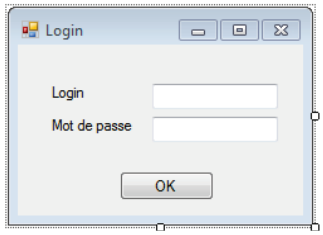
L'exemple précédent était celui d'un programme classique, s'exécutant en ligne de commande : tous les programmes de ce type que vous avez écrits jusques ici peuvent naturellement être réécrit dans cet environnement.

Nous allons dans la suite nous consacrer à des applications offrant une interface graphique. Même simples, celles-ci nécessitent l'écriture d'une quantité considérable de code : créations de fenêtres et d'éléments visuels divers, code de gestion des événements générés par l'utilisateur,... Heureusement, une grande partie de ce code est routinière et elle peut être standardisée et incluse dans des bibliothèques.

Comme première application graphique, considérons un exemple très simple (et sans grand intérêt !) : la construction d'une boîte de login.

Il convient cette fois de créer un projet de type `Windows Forms`, que l'on pourra nommer `Login`.

L'application s'ouvre sur un onglet de nom Form1.design (« la fenêtre de conception) présentant le cadre de base d'une fenêtre, du type de celles utilisées pour un formulaire (d'où le nom de « `Windows Form` »). On peut la compiler et l'exécuter, mais naturellement elle ne fait pas grand-chose !




Nous allons la développer par l'ajout d'éléments, appelés contrôles, ce qui se fait en mode graphique par simple glisser-déplacer depuis la boîte à outils accessible sur la gauche de la fenêtre principale : un bouton OK, 2 étiquettes (`Label`), 2 champs de saisie (`Textbox`) (illustration ci-contre).

Grâce à la fenêtre de Propriétés, en bas à droite, on peut modifier les propriétés des contrôles et changer le nom de la fenêtre en l'appelant 'LoginForm' ainsi que son titre pour afficher `Login` (comme sur l'illustration ci-dessus). On peut aussi modifier texte des boutons, nom des contrôles, champ mot de passe ou valeur de retour du bouton.

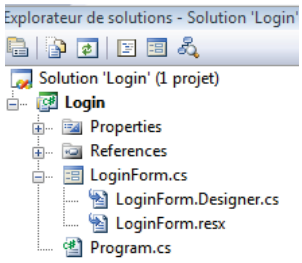
Un clic sur le bouton droit, puis l'élément de menu `Afficher le code`, montre le code généré :

```
using System.Text;
using System.Windows.Forms;

namespace Login
{
    public partial class LoginForm : Form
    {
        public LoginForm()
        {
            InitializeComponent();
        }
    }
}
```

Cela paraît bien peu pour faire tout ça. C'est qu'en fait ce n'est là que la partie du code que l'on peut être amené à modifier.

[6] Si vous travaillez sur une version 64 bits de Windows, vous devez développer en mode x86 (Projet|Propriétés, Application, Plateforme cible) pour que les modifications du code pendant le débogage soient possibles.

Examinons plus en détails le code généré, grâce à l'onglet `Explorateur de solutions` qui montre les divers fichiers créés par l'application (et qui permet par exemple de les renommer). Sur l'exemple ci-contre, on voit sur l'onglet `Explorateur` que quatre fichiers (au moins !) ont été généré (`LoginForm.cs`, `LoginForm.Designer.cs`, `Program.cs` et `LoginForm.resx`).




L'environnement a donc généré beaucoup code, dont l'essentiel se trouve dans le fichier `LoginForm.designer.cs`. En voici un extrait :



```
namespace Login
{
    partial class LoginForm
    {
        /// <summary>
        /// Variable nécessaire au concepteur.
        /// </summary>
        private System.ComponentModel.IContainer components = null;
        #region Code généré par le Concepteur Windows Form
        /// <summary>
        /// Méthode requise pour la prise en charge du concepteur - ne modifiez pas
        /// le contenu de cette méthode avec l'éditeur de code.
        /// </summary>
        private void InitializeComponent()
        {
            this.loginLabel = new System.Windows.Forms.Label();
            this.login = new System.Windows.Forms.TextBox();
            this.SuspendLayout();
            //
            // loginLabel
            //
            this.loginLabel.AutoSize = true;
            this.loginLabel.Location = new System.Drawing.Point(24, 31);
            this.loginLabel.Name = "loginLabel";
            this.loginLabel.Size = new System.Drawing.Size(33, 13);
            this.loginLabel.TabIndex = 0;
            this.loginLabel.Text = "Login";
            //
            // login
            //
            this.login.Location = new System.Drawing.Point(107, 31);
            this.login.Name = "login";
            this.login.Size = new System.Drawing.Size(100, 20);
            this.login.TabIndex = 1;
            //
            // LoginForm
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(227, 135);
            this.Controls.Add(this.loginLabel);
            this.Controls.Add(this. login);
            this.Name = "LoginForm";
            this.Text = "Login";
            this.ResumeLayout(false);
            this.PerformLayout();
        }
        #endregion
        private System.Windows.Forms.Label loginLabel;
        private System.Windows.Forms.TextBox login;
    }
}
```

Les deux contrôles que nous avons créés sont définis dans les deux dernières lignes du fichier. Ils sont initialisés dans la fonction `InitializeComponent()` (générée par Visual Studio[7]) par les lignes comme :

```
this.login = new System.Windows.Forms.TextBox();
```

[7] Et à laquelle il est préférable de ne pas toucher, puisque c'est elle que l'EDI utilise pour générer l'interface graphique – et réciproquement !

Le reste de la fonction définit leurs propriétés de base.

On peut en outre remarquer deux structures intéressantes. La structure « summary » permet de disposer d'une « documentation automatique » au format XML (que l'on peut générer à la compilation). C'est aussi la source des informations sur le type affiché dans IntelliSense et l'Explorateur d'objets :

```
/// <summary>
/// Teste la présence d'un point
/// </summary>
/// <param name="pt">Point à tester</param>
/// <returns>true si la forme contient le point</returns>
public bool Contains(Point pt)
{
    return forme.Contains(pt);
}
```



La structure region/endregion permet d'encadrer des zones de code que l'on peut réduire ou développer pour faciliter la lecture (une façon assez commode de commenter son code) et faire ainsi tenir tout son code sur une même page :

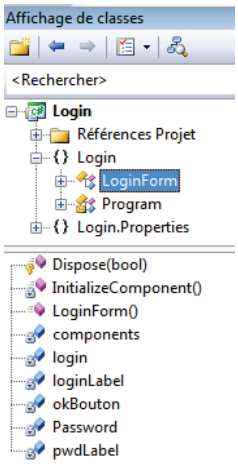
```
#region Code généré par le Concepteur Windows Form
…
#endregion
```

Reste à localiser la fonction `Main()` ! Elle se trouve dans le troisième fichier, `Program.cs.` , qui contient toujours une classe `Program` et une fonction `Main()` comme point d'entrée. Elle est simple, courte et nous n'aurons jamais à la modifier[8] !! Voici l'intégralité de ce fichier :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
namespace Login
{
    static class Program
    {
        /// <summary>
        /// Point d'entrée principal de l'application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new LoginForm());
        }
    }
}
```

Les clauses `using` indiquent les bibliothèques nécessaires, comme `System.Windows.Forms` dans laquelle se trouve le code des composants que nous avons inséré.

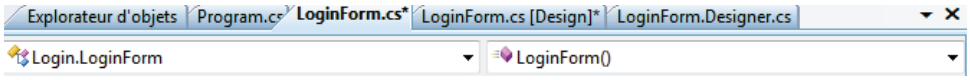
On peut maintenant compiler et exécuter notre programme, qui se contente d'ouvrir une boîte de dialogue.




L'onglet `Affichage de classes` (illustration ci-contre) nous montre la structure du programme et l'on peut voir par exemple que la fenêtre LoginForm a six variables privés (de `components` à `pwdLabel`) et trois méthodes (de `Dispose()` à `LoginForm()`).

Un double clic sur l'un de ces éléments amène à sa définition dans le code.

On peut aussi y accéder grâce aux listes déroulantes qui apparaissent en haut de la page de code :




Ecrivons maintenant la réponse à l'évènement `Click` du bouton Ok. Pour créer le prototype de la fonction, il suffit d'un double clic sur le bouton dans la fenêtre de conception :

[8] Ni même à la comprendre !

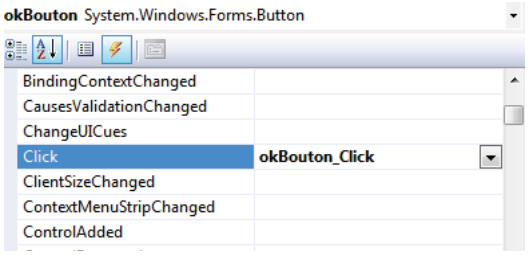
```
private void okBouton_Click(object sender, EventArgs e)
{

}
```

Cette fonction possède deux paramètres, l'un qui précise quel est l'objet émetteur, l'autre qui fournit des détails sur l'événement qui a déclenché son appel. On pourra par exemple rajouter du code qui ouvre une boîte de dialogue standard demandant à l'utilisateur s'il est bien sûr de vouloir fermer la fenêtre lorsque l'on clique sur le bouton :

```
if (MessageBox.Show("Etes vous sûr ?", "Fermeture", MessageBoxButtons.OKCancel,
MessageBoxIcon.Question) == DialogResult.OK)
{
    Close();
}
```

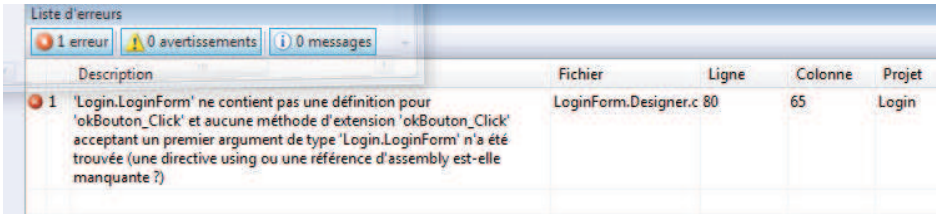


On peut constater sur l'Affichage de classe qu'une fonction a été ajoutée à la fenêtre puis regarder les modifications apparues dans les propriétés du bouton OK (grâce au quatrième bouton de cette fenêtre  qui fait apparaître la liste des événements), on aurait pu passer par là pour créer la fonction.



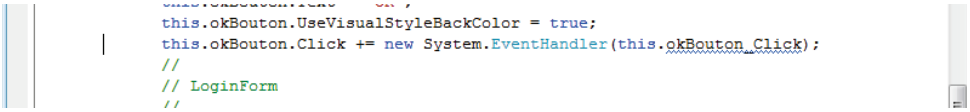

Compilons et testons ce programme.

Supprimons maintenant la méthode `okBouton_Click` que nous venons de créer. Recompilons et exécutons le programme. Que se passe-t-il ?




L'onglet `Liste d'erreur` nous donne la réponse et un double clic sur la ligne décrivant l'erreur, y donne un accès direct, dans le fichier `LoginForm.Designer.cs`.

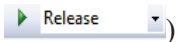
L'erreur est soulignée en bleu dans l'image ci-dessous :




On peut maintenant la corriger en supprimant la ligne incriminée, et apprendre en même temps comment une méthode de réponse est liée à l'événement : l'objet oKBouton que nous avons créé possède un « événement » de nom Click et lorsque nous avons créé la réponse à cet événement, l'EDI a associé cette réponse à l'événement en insérant la ligne suivante qui a ajouté un gestionnaire d'événement (`EventHandler`) à cet événement :

```
this.okBouton.Click += new System.EventHandler(this.okBouton_Click)
```

## 2.5. Distribution du programme

Une fois le programme parfaitement débuggé, on pourra le générer en mode Release ( 

) pour l'exécuter indépendamment de l'environnement Visual Studio, comme toute autre programme Windows. L'exécutable ainsi généré sera naturellement plus petit, car ne comportant plus d'informations de debugging. On le trouvera dans le répertoire `bin|Release`. L'environnement offre aussi des procédures de distribution du logiciel que nous ne détaillerons pas ici.

# 3. Evénements de souris et éléments simples d'interface

## 3.1. Dessin à main levée

### i. Version simple


Nous allons maintenant voir comment gérer plusieurs évènements utilisateurs et leurs interactions en créant un programme très simple permettant de dessiner à main levée (illustration ci-contre) : une version très simplifiée de programmes comme Paint, que tout le monde a utilisé un jour ou l'autre[9].

Pour ce faire, nous créons un nouveau projet de type Application Windows Form et de nom `DessinMainLevee`, puis nous renommons en `Dessin` la feuille `Form1` juste créée (on peut aussi changer son titre en « Dessin à main levée »).

Le mécanisme est simple : le dessin commence par un clic gauche sur la feuille, se poursuit en dessinant une courbe lorsque l'on déplace la souris en maintenant le bouton gauche enfoncé et s'achève lors du relâchement du bouton.

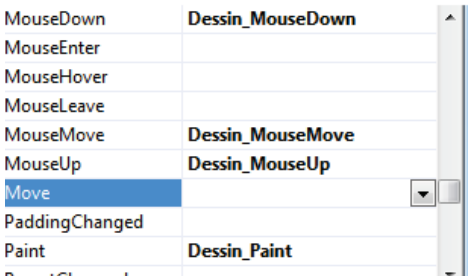


Plus précisément, le clic initial (événement `MouseDown`) va créer le premier point et mettre le programme « en mode dessin ».

Le dessin va se poursuivre lors du déplacement de la souris (événement `MouseMove`) en rajoutant des points à une liste de points qui constituera la partie visible du dessin, celle que nous devrons afficher.

Il va s'achever lorsque le bouton sera relâché (événement `MouseUp`) en ajoutant le dernier point et en mettant fin au mode dessin.

Du point de vue du programmeur, la construction du dessin se fait donc simplement en traitant la réponse aux trois événements générés par la souris dont nous venons d'énumérer la liste. L'affichage du dessin sera réalisé par la réponse à l'événement `Paint` qui devra être généré chaque fois que la fenêtre doit se réafficher.




La création des méthodes de réponse à ces quatre événements se fait simplement en double-cliquant sur le nom des évènements dans la fenêtre Propriétés pour obtenir le résultat présenté ci-contre.

Dans le fichier de code dont le nom se termine par Designer.cs, le résultat est l'ajout de quatre lignes qui réalisent le lien entre les événements et les fonctions de réponde. Par exemple :

```
this.MouseUp += new System.Windows.Forms.MouseEventHandler(this.Objets_MouseUp);
this.Paint += new System.Windows.Forms.PaintEventHandler(this.Objets_Paint);
```

Comme nous venons de le dire, l'élément central du dessin, ses données, sera constitué par une liste de points, que nous appelons `ligne`, et dans laquelle seront stockées les coordonnées du point courant : toutes les informations sur le dessin seront contenues dans cette liste.

Ces coordonnées seront fournies par le paramètre `e` de type `MouseEventArgs`, transmis à chacune des méthodes de réponses à un événement de souris et qui fournit divers paramètres comme la position du curseur lors de l'appel de la fonction[10].

L'affichage du dessin est réalisé dans la méthode `Dessin_Paint`, qui est appelée chaque fois que la fenêtre doit être redessinée parce que Windows a généré un événement `Paint` lui demandant de le faire.

---

[9] Sinon, il ne faut pas hésiter à essayer !

[10] La documentation MSDN fournit la description des propriétés de la classe MouseEventArgs (extrait) :

| | |
|---|---|
| Button | Obtient le bouton de la souris sur lequel l'utilisateur a appuyé. |
| Location | Obtient l'emplacement de la souris pendant la génération d'événement de souris. |
| X | Obtient la coordonnée x de la souris pendant la génération d'événement de souris. |
| Y | Obtient la coordonnée y de la souris pendant la génération d'événement de souris. |

La méthode reçoit en paramètre un objet de type `PaintEventArgs`, qui contient, parmi ses propriétés, un objet de type « `Graphics` » qui possède de nombreuses méthodes permettant d'écrire ou de dessiner sur le formulaire, dont la méthode `DrawLines` :

```
e.Graphics.DrawLines(Pens.Black, trait.ToArray());.
```

Dans notre exemple, elle trace un ensemble de lignes reliant les points stockés dans la liste[11], en utilisant un crayon standard de couleur noire (issu de l'énumération `Pens`). Pour que le dessin apparaisse au fur et à mesure de sa construction, la méthode `Dessin_MouseMove` appelle la méthode `Refresh()` qui génère un événement `Paint` chaque fois qu'un nouveau point est ajouté, et force ainsi le réaffichage.

Les lignes qui suivent contiennent l'intégralité du code que devra contenir le fichier principal de notre application (Form1.cs). Il ne s'agit évidemment pas de le recopier servilement[12], mais de le recréer en s'efforçant d'en comprendre tout les aspects[13].



```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Windows.Forms;
namespace DessinMainLevee
{
    public partial class Dessin : Form
    {
        private List<Point> ligne = new List<Point>
        private bool enDessin = false;
        public Dessin()
        {
            InitializeComponent();
        }
        private void Dessin_MouseDown(object sender, MouseEventArgs e)
        {
            enDessin = true;
            ligne.Add(e.Location);
        }
        private void Dessin_MouseMove(object sender, MouseEventArgs e)
        {
            if (enDessin)
            {
                ligne.Add(e.Location);
                Refresh();
            }
        }
        private void Dessin_MouseUp(object sender, MouseEventArgs e)
        {
            enDessin = false;
        }
        private void Dessin_Paint(object sender, PaintEventArgs e)
        {
            if (ligne.Count > 1)
            {
                e.Graphics.DrawLines(Pens.Black, trait.ToArray());
            }
        }
    }
}
```

## ii. Deuxième version

Le programme simple que nous venons d'écrire ne fait pas tout à fait ce que l'on peut souhaiter, puisqu'il ne sépare pas les éléments construit séparément au cours du dessin, mais les dessine en continu.

Pour l'améliorer, on peut définir un nouveau type d'objet, qui va représenter un trait continu, en créant une classe regroupant tous les éléments nécessaires (on peut la mettre dans un nouveau fichier par `Projet|Ajouter une classe…` ou à la suite de la classe précédente) :

---

[11] La fonction `DrawLines` prend en paramètre un tableau de points alors que nous disposons d'une liste. D'où la conversion `points.ToArray();`

[12] Ce qui ne pas fonctionner !

[13] Les exemples ultérieurs seront en général incomplets

```
public class Ligne
{
    private List<Point> points;
    private Color couleur;
    private int épaisseur;
    public Ligne(Color c, int e)
    {
        this.couleur = c;
        this.épaisseur = e;
        points = new List<Point>();
    }
    public void AddPoint(Point p)
    {
        points.Add(p);
    }
    public void Dessine(Graphics g)
    {
        g.DrawLines(new Pen(couleur, épaisseur), points.ToArray());
    }
}
```



Elle comporte trois variables privées, un constructeur, une méthode pour ajouter un point et une méthode de dessin à laquelle il faut passer comme paramètre l'objet graphique sur lequel la ligne doit se dessiner, à l'aide d'un crayon ayant la couleur et l'épaisseur désirée (`Pen`).

Il faut alors modifier légèrement le code de la classe `Dessin`, puisque le dessin créé sera maintenant constitué d'une liste de lignes (cette fois encore, le code est présenté dans sa quasi-intégralité) :

```
public partial class Dessin : Form
{
    private Ligne ligne;
    private List<Ligne> lignes = new List<Ligne>();
    private Color couleurParDefaut = Color.Blue;
    private int epaisseurParDefaut = 2;
    private bool enDessin = false;
    public Dessin()
    {
        InitializeComponent();
    }
    private void Dessin_MouseDown(object sender, MouseEventArgs e)
    {
        enDessin = true;
        ligne = new Trait(couleurParDefaut,epaisseurParDefaut);
        lignes.Add(trait);
        ligne.AddPoint(e.Location);
    }
    private void Dessin_MouseMove(object sender, MouseEventArgs e)
    {
        if (enDessin)
        {
            ligne.AddPoint(e.Location);
            Refresh();
        }
    }
    private void Dessin_MouseUp(object sender, MouseEventArgs e)
    {
        enDessin = false;
    }
    private void Dessin_Paint(object sender, PaintEventArgs e)
    {
        foreach(Ligne t in lignes)
            t.Dessine(e.Graphics);
    }
}
```

### iii. Modification des paramètres par défaut

Dans le code que nous venons d'écrire, les paramètres d'épaisseur et de couleur par défaut sont fixés au démarrage (« codés en dur »).

Insérons maintenant sur la feuille (par glisser déplacer) un élément `ToolStrip` pour disposer d'une barre d'outils.

On peut alors ajouter un nouveau bouton à la barre d'outils, par exemple un `ToolStripSplitButton` (un menu déroulant), dont on changera le nom en `Epaisseur` et le texte affiché en « Epaisseur du trait » et auquel on ajoutera les éléments de nom `Epaisseur1` et de texte « 1 », `Epaisseur2` » et de texte « 2 », etc.

A chacun de ces éléments, on associera une fonction de gestion de l'événement par la méthode habituelle. On modifiera le code, par exemple en écrivant[14] :

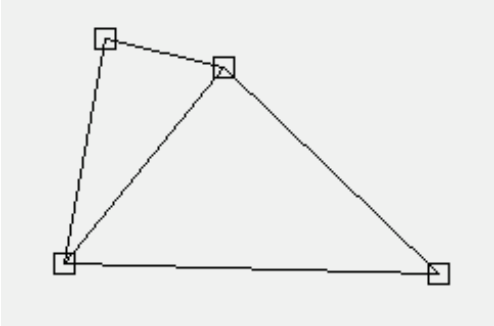
```
private void Epaisseur4_Click(object sender, EventArgs e)
{
    epaisseurParDefaut = 4;
}
```

Complément 1. *Comment modifier la couleur par défaut[15] ?*

## 3.2. Dessin d'objets



Nous allons maintenant créer un nouveau projet de type « Application Windows Form », pour dessiner des objets un peu plus complexes (en l'occurrence des rectangles reliés par des lignes droites comme sur la figure ci-contre) que l'on pourra modifier en déplaçant ou supprimant des éléments.



### i. Dessin de rectangles

Sur le modèle de la classe `Ligne`, nous allons créer une nouvelle classe, que nous appellerons `Rect` (faute de mieux car le nom `Rectangle` est déjà pris par une classe prédéfinie dans l'espace de noms `System.Drawing` !), réunissant un rectangle (classe standard `Rectangle`) et divers paramètres complémentaires qui nous permettront de savoir comment le dessiner (couleur, épaisseur du trait). Elle possède (au moins) un constructeur et une méthode de dessin.

```
public class Rect
{
    private Rectangle rect;
    private Color couleur;
    private int épaisseur;
    public Rect(Rectangle r, Color c, int e)
    {
        rect = r;
        couleur = c;
        épaisseur = e;
    }
    public void Dessine(Graphics g)
    {
        Pen p = new Pen(couleur, épaisseur);
        g.DrawRectangle(p, rect);
    }
}
```

La création d'un rectangle peut se faire par un simple clic sur la fenêtre (`Dessin_MouseDown`), puis par son insertion dans une `List<Rect>` que nous appellerons `rectangles` et qui est une propriété de la classe principale. Le dessin lui-même aura évidemment lieu dans la méthode `Dessin_Paint`.

```
Rect r = new Rect(new Rectangle(e.Location, new Size(10, 10)), couleur, epaisseur);
rectangles.Add(r);
```

Complément 2. *Modifier le programme pour qu'il permette de choisir l'épaisseur par défaut du rectangle (on pourra insérer une barre d'outils).*

Complément 3. *Ajouter un texte au rectangle et l'afficher (la méthode `DrawString` de l'objet `Graphics` nécessite la création d'un objet de type `Font`[16]).*

---

[14] On peut évidemment faire plus malin !

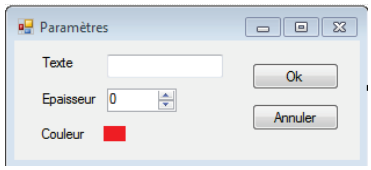
[15] Exercice complémentaire à réaliser si le temps le permet.

[16] La classe `Font` possède de nombreux constructeurs différents et de nombreux paramètres possibles.

## ii. Création d'une boîte de dialogue simple

Ajoutons un nouveau formulaire (`Projet|Ajouter un formulaire Windows`), que l'on appelle `Params` et sur lequel on insère des éléments permettant de modifier le texte, l'épaisseur du trait (`NumericUpDown`) et sa couleur (en utilisant un `Label` coloré). On rajoute aussi trois `Label` permettant d'indiquer le but de chacun de ces contrôles.

A chacun des contrôles destinés à modifier les paramètres du rectangle, on associera une propriété.




On insère ensuite deux boutons (`Button`). Leur propriété `DialogResult` sera mise selon le cas à `OK` ou `Cancel`. La boîte de dialogue résultante est représentée sur la figure ci-contre.

Il faut maintenant sélectionner un rectangle par un clic sur le bouton droit de la souris, puis ouvrir la boîte de dialogue.

Il faut tout d'abord modifier légèrement la méthode `MouseDown` pour qu'elle distingue entre ses deux boutons gauche et droit :



```
private void Dessin_MouseDown(object sender, MouseEventArgs e)
{
    if (e.Button == MouseButtons.Left)
    {
        curRect = new Rect(new Rectangle(e.Location, new Size(10, 10)), couleur,
                  epaisseur);
        rectangles.Add(curRect);
    }
    else
    {
    }
}
```

La sélection d'un rectangle nécessite de parcourir la liste des rectangles et de tester pour chacun d'eux si le clic de la souris a bien eu lieu à l'intérieur du rectangle (on pourra par exemple ajouter à la classe `Rect` une méthode booléenne `Contains` utilisant la fonction `Rectangle.Contains`).

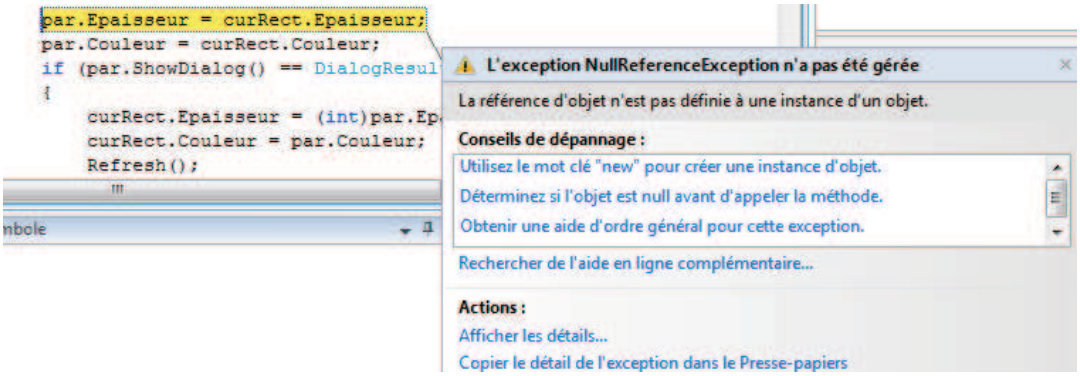
Une méthode auxiliaire permettra d'écrire un code plus clair :

```
private Rect TrouveObjet(Point p)
{
    foreach (Rect re in rectangles)
    {
        if (re.Contains(p))
        {
            return re;
        }
    }
    return null;
}
```

Une fois le rectangle sélectionné, le principe de l'utilisation de la boîte est très simple :

```
curRect = TrouveObjet(e.Location);
Params par = new Params();
par.Epaisseur = curRect.Epaisseur;
par.Couleur = curRect.Couleur;
if (par.ShowDialog() == DialogResult.OK)
{
    curRect.Epaisseur = par.Epaisseur;
    curRect.Couleur = par.Couleur;
    Refresh();
}
```

Ce code n'est pas très fiable. Il suffit de cliquer en dehors d'un carré pour s'en apercevoir : la fenêtre de l'illustration ci-dessous apparaît tout de suite, pointant sur une ligne surlignée en jaune :




En passant la souris sur cette ligne, et plus précisément sur la variable `curRect`, on voit apparaître une petite bulle d'aide affichant : `curRect | null` . Pourquoi ? Que faut-il faire ?



Complément 4. *Modifier la couleur*

Complément 5. *Comment modifier la police avec laquelle le texte est affiché ?*

### iii. Dessin des traits

Commençons par améliorer notre programme pour qu'il ne puisse pas dessiner deux rectangles au même endroit. Pour cela nous pouvons utiliser la fonction de recherche que nous avons écrite précédemment et modifier légèrement notre code pour obtenir :

```
private void Dessin_MouseDown(object sender, MouseEventArgs e)
{
    Rect curRect = TrouveObjet(e.Location);
    if (e.Button == MouseButtons.Left)
    {
        if (curRect == null)
        {
            curRect = new Rect(new Rectangle(e.Location, new Size(10, 10)),
               couleur, epaisseur);
            rectangles.Add(curRect);
            Refresh();
        }
     }
    else
    {
        // réponse au clic droit écrite précédemment
    }
}
```

Nous allons maintenant dessiner des traits rectilignes reliant les rectangles.

Le dessin du trait (qui ira du centre du premier rectangle jusqu'au centre du second[17]) va utiliser la même technique que précédemment : début dans la méthode `MouseDown`, fin dans `MouseUp` et insertion dans une liste de traits.

Plusieurs choix sont en fait possibles : nous allons ici choisir de commencer un trait lorsque l'on a cliqué sur un rectangle et de le terminer sur un autre rectangle ou s'il n'y a pas de second rectangle, de le terminer en créant un rectangle.

Les traits dont nous avons besoin ici sont plus simple que ceux de l'exemple précédents puisqu'ils sont déterminés par deux points et on pourrait les dessiner simplement par une ligne de code du type :

```
e.Graphics.DrawLine(Pens.Red, pt1, pt2);
```

Mais en fait, les traits joignent deux rectangles et non deux points, et au lieu de créer une classe comportant deux points et les paramètres nécessaires pour spécifier l'épaisseur et la couleur, l'origine et l'extrémité, il est préférable d'utiliser deux rectangles, l'un comme source du trait, l'autre comme destination :

```
public class Trait
{
    private Rect source;
    private Rect destination;
    private Color couleur;
```

[17] La classe `Rect` n'a pas de membre `Centre`. Il faut donc créer une propriété rendant ce service.

```
    private int epaisseur;
    public Trait(Rect sour, Rect dest, Color c, int e)
    {
        source = sour;
        destination = dest;
        couleur = c;
        epaisseur = e;
    }
    public void Dessine(Graphics g)
    {
        Pen p = new Pen(couleur, epaisseur);
        g.DrawLine(p, source.Centre, destination.Centre);
    }
}
```

Un peu de réflexion est maintenant nécessaire pour modifier le code des méthodes `MouseDown`, `MouseMove` et `MouseUp` qui vont nous permettre de créer le trait.

La première va devoir tester s'il y a un rectangle à l'emplacement du clic (c'est déjà fait) et dans ce cas commencer le dessin du trait.

La seconde peut ne rien faire.

La troisième va aussi tester s'il y a un rectangle à l'emplacement du clic, en créer un s'il n'y en a pas et créer le trait joignant les deux rectangles qu'elle va stocker dans la liste des traits.



Complément 6. *Pour que le dessin soit plus agréable, on peut pendant le mouvement de la souris dessiner un trait (d'une autre couleur ou en pointillé, éventuellement avec un rectangle au bout) montrant ce que sera le futur trait et où sera positionné le futur rectangle.*

### iv. Déplacement des rectangles

Pour déplacer un rectangle (en faisant en sorte que les traits suivent le déplacement du rectangle sur la feuille), il faut d'abord le sélectionner, comme nous savons le faire. Mais il faut surtout être capable de distinguer cette action de celle qui conduirait à commencer le dessin d'un trait.

Un nouvel indicateur d'état est donc nécessaire. On peut créer un nouveau booléen, ou se contenter d'un élément de barre d'outils (qu'il faut naturellement rajouter), dont on vérifiera l'état (`dessin.Checked` ci-dessous).

A l'intérieur de la méthode `MouseDown`, on verra donc apparaître un nouveau test :

```
private void Objets_MouseDown(object sender, MouseEventArgs e)
{
    if (dessin.Checked)
    {
        Rect curRect = TrouveObjet(e.Location);
        if (e.Button == MouseButtons.Right)
        {
            Edition des propriétés
        }
        else
        {
            Dessin
        }
    }
    else
    {
        Déplacement
    }
}
```

Dans la région de code Déplacement, la méthode devra se mettre en mode déplacement (nouveau booléen), et notera le rectangle à déplacer. La méthode `MouseMove` déplacera le rectangle et la méthode `MouseUp` mettra fin au déplacement. L'écrire d'un petit automate fini n'est pas forcément inutile ici.

Complément 7. *Supprimer un rectangle.*

Complément 8. *Sélectionner un trait*[18]*, modifier son épaisseur et sa couleur, le supprimer.*

Complément 9. *Déplacer un ensemble de rectangles. Pour ce faire, on crée un rectangle de sélection que l'on affiche en pointillé pendant la procédure de sélection (il disparaîtra à la fin), puis on sélectionne les objets contenus dans le rectangle de sélection grâce à une variante de la fonction* `TrouveObjet` *décrite plus haut.*

#### v. Extension possible

On souhaite ajouter la possibilité de dessiner d'autres formes, comme des cercles (ou des ellipses). Différents des rectangles, ils ont tout de même quelques caractéristiques communes : un trait d'une épaisseur donné, d'une couleur donnée, l'inscription dans un rectangle donné, la possibilité de se dessiner et celle de nous dire qu'ils contiennent un point donné.

Ces propriétés peuvent être résumées dans une interface que les deux types d'objets devront implémenter. Nous pouvons extraire l'interface par un « refactoring » de la classe `Rect` pour obtenir :



```
interface IForme
{
    void Dessine(System.Drawing.Graphics g);
    bool Contains(System.Drawing.Point pt);
    Color Couleur { get; set; }
    int Epaisseur { get; set; }
    Rectangle Rect { get; set; }
}
```

La définition de la classe `Rect` a été modifiée par l'EDI en

```
public class Rect : IForme
```

On peut modifier la liste de rectangles dont on disposait jusque là en une :

```
List<IForme> rectangles = new List<IForme>();
```

Et recompiler pour s'assurer que ça fonctionne bien. Comme on va maintenant mettre dans la liste `rectangles` autre chose que des rectangles, il peut être judicieux de la renommer en `formes`. Ceci se fait simplement grâce au clic droit, `Refactoriser|Renommer`.

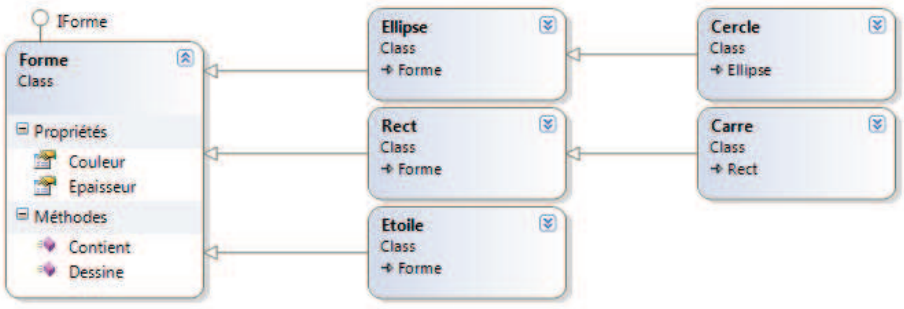
On peut maintenant créer une classe `Cercle` dérivée de l'interface `IObjet`

```
public class Cercle : IForme
{
}
```

Puis par clic droit sur `IForme`, `|Implémenter l'Interface`, construire le squelette du code. Reste à remplir les trous…

Complément 10. *Modifier l'application pour qu'elle donne le choix entre le dessin de ronds et de rectangles.*

Remarque : les classes `Rect` et `Cercle` étant forcément très semblables (seul l'affichage diffère), on pourrait d'abord créer une classe `Forme` dont on ferait dériver `Rect` et `Cercle` et qui contiendrait toutes les méthodes communes. En réalité, on souhaiterait même avoir un ensemble de classes plus cohérent que celui que nous venons de décrire comme par exemple :




### 3.3. Sauvegarde

#### i. Sérialisation

La sauvegarde et la relecture d'un dessin se fera en principe à la demande de l'utilisateur, et nous allons donc rajouter sur notre barre d'outils de nouveaux boutons comme sur l'illustration ci-contre (on peut le faire facilement en cliquant sur la flèche qui apparaît en haut à droite 

[18] Il va falloir ici se souvenir de la façon d'écrire l'équation d'une droite.

de la barre d'outils lorsqu'elle est sélectionnée : on clique alors sur `Insérer des éléments standard` puis on supprime les éléments en trop).

La méthode usuelle permet de créer les méthodes de réponse. Reste maintenant à en écrire le code

La solution la plus simple pour sauvegarder les données de notre dessin est la « sérialisation », autrement dit la sauvegarde « tel quel » des données stockées en mémoire. Elle nécessite la présence des clauses :

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

La sauvegarde se fait dans une première fonction dont l'essentiel figure ci-dessous :

```
Stream stream = File.Open("Dessin.des", FileMode.Create);
BinaryFormatter bformatter = new BinaryFormatter();
bformatter.Serialize(stream, formes);
bformatter.Serialize(stream, traits);
stream.Close();
```



La relecture se fait par une fonction tout à fait analogue. On notera simplement que lorsque la fonction relit un élément dans le fichier elle ne connaît pas à priori son type et qu'on doit le préciser grâce à un « transtypage » :

```
Stream stream = File.Open("Dessin.des", FileMode.Open);
bformatter = new BinaryFormatter();
formes = (List<IObjet>)bformatter.Deserialize(stream);
traits = (List<Trait>) bformatter.Deserialize(stream);
stream.Close();
```

Pour que tout cela fonctionne il faut préciser que toutes les classes que nous avons créées (`Rect`, `Trait`,...) sont « serialisables », la plupart des types standard l'étant :

```
[Serializable]
public class Rect : DessinObjets.IObjet
```

### ii. Exceptions

Les entrées-sorties sont des causes fréquentes d'erreurs à l'exécution, dont la nature n'est pas toujours prévisible (fichier ou répertoire inexistant, structure du fichier incorrecte, droits d'accès insuffisants, ...), mais qu'il est relativement facile de traiter grâce au mécanisme d'exception :

```
Stream stream = null;
try
{
    stream = File.Open("Dessin.des", FileMode.Open);
    //reste du code
}
catch (Exception ex)
{
    MessageBox.Show("Erreur : " + ex.Message);
}
finally
{
    if (stream != null)
    {
        stream.Close();
    }
}
```

### iii. Sauvegarde au format XML

La sauvegarde par sérialisation produit un fichier binaire dont il faut connaître a priori la structure pour pouvoir le relire. Il est donc difficile de transmettre les données à quelqu'un qui utilise un autre programme.

Une autre possibilité pour la sauvegarde est naturellement la sauvegarde dans un fichier texte facilement lisible par des programmes très simples. Un rectangle noir de dimension 10, 10, 50, 40 et d'épaisseur 2 peut, par exemple, être stocké sous la forme :

```
2 - Noir - 10,10,50,40
```

Mais pour qu'un autre utilisateur (ou un programme) puisse interpréter ces données il faut bien sûr rajouter d'autres informations. Il est alors souhaitable d'utiliser le format XML, un format « standard » d'échange de données dans lequel les informations sont stockées sous forme arborescente, chaque nœud étant encadré par une balise ouvrante et une balise fermante, comme dans le fragment ci-dessous.

```
<?xml version="1.0" encoding="UTF-8" ?>
<DESSIN>
   <RECTANGLE>
      <EPAISSEUR>
         1
      </EPAISSEUR>
      <COULEUR>
         Color [Black]
       </COULEUR>
       <CENTRE>
         {X=98,Y=71}
       </CENTRE>
   </RECTANGLE>
   <RECTANGLE>
      <EPAISSEUR>
         1
      </EPAISSEUR>
      <COULEUR>
         Color [Black]
      </COULEUR>
       <CENTRE>
         {X=178,Y=172}
      </CENTRE>
   </RECTANGLE>
</DESSIN>
```



Le code suivant (qui n'est que de la manipulation de texte) suffit à construire ce fichier :

```
StreamWriter sw = new StreamWriter(@"F:\Dessin.Xml"[19]);
sw.WriteLine("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>");
sw.WriteLine("<DESSIN>");
foreach (Rect r in rectangles)
{
    sw.WriteLine("   <RECTANGLE>");
    sw.WriteLine("      <EPAISSEUR>");
    sw.WriteLine("         " + r.Epaisseur.ToString());
    sw.WriteLine("      </EPAISSEUR>");
    sw.WriteLine("      <COULEUR>");
    sw.WriteLine("         " + r.Couleur.ToString());
    sw.WriteLine("      </COULEUR>");
    sw.WriteLine("      <CENTRE>");
    sw.WriteLine("         " + r.Centre.ToString());
    sw.WriteLine("      </CENTRE>");
    sw.WriteLine("  </RECTANGLE>");
}
sw.WriteLine("</DESSIN>");
sw.Close();
```

Mais le centre n'est absolument pas suffisant pour reconstruire le rectangle lui-même, qui est une donnée privée de la classe `Rect` et n'est donc pas accessible depuis l'application de dessin.

On préfèrera donc ajouter à la classe `Rect` d'une méthode spécifique qui fera la sauvegarde, sans modifier la visibilité des attributs de la classe :

```
public string ToXML()
{
    string text = "<RECTANGLE>";
    text+="   <EPAISSEUR>";
    text+="     " + Epaisseur.ToString();
    text+="   </EPAISSEUR>";
    text+="   <COULEUR>";
    text+="      " + Couleur.ToString();
    text+="   </COULEUR>";
    text+="   <FORME>";
    text+="      " + rect.ToString();
    text+="   </FORME>";
    text+="</RECTANGLE>";
    return text;
}
```

[19] Le caractère \ est un caractère spécial qui ne peut apparaître normalement dans une chaîne de caractères. Il faut soit le redoubler ("F:\\Dessin.Xml") soit faire précéder la chaîne d'une arobase (@"F:\Dessin.Xml").

La sauvegarde des formes pourra alors se faire par un code du type suivant (a condition d'avoir prévue la méthode au niveau de l'interface) :

```
foreach (IForme r in rectangles)
{
    sw.WriteLine(r.ToXML());
}
```

Comme pour un fichier texte ordinaire, la relecture est évidemment plus complexe que la sauvegarde. L'avantage du format Xml est qu'il contient à travers ces balises beaucoup plus d'informations qu'une simple succession de lignes de texte et que l'on dispose d'outils standard pour les analyser. Ainsi la lecture du fichier dans un objet de type `XmlDocument` va directement reconstruire la structure arborescente que nous avons sauvegardée. L'analyse peut alors se faire avec un code du type suivant[20] :

```
XmlDocument doc = new XmlDocument();
doc.Load(@"F:\Dessin.Xml");
foreach (XmlNode xN in doc.ChildNodes)
{
    if (xN.Name == "DESSIN")
    {
        foreach (XmlNode xNN in xN.ChildNodes)
        {
            if (xNN.Name == "RECTANGLE")
            {
                //faire quelquechose
            }
        }
    }
}
```



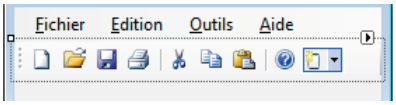
Complément 11. *L'une des difficultés qui reste à traiter est la reconstruction les liens entre les traits et les rectangles relus depuis le fichier Xml.*

[20] D'autres approches sont possibles comme on verra au paragraphe 6.2.ii.

# 4. Contrôles prédéfinis

## 4.1. Ouvrir un fichier image et l'afficher

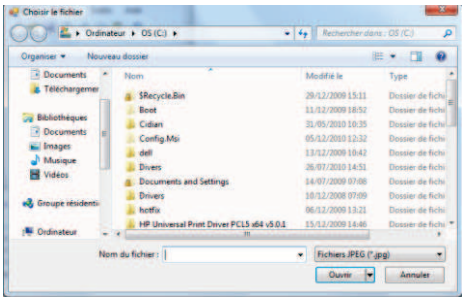
Créons un nouveau projet, que nous appellerons par exemple Afficheur. Après avoir renommé les divers éléments qui en ont besoin, ajoutons tout d'abord à la feuille principale un élément `MenuStrip` et un élément `ToolStrip` pour disposer d'une barre de menus et d'une barre d'outils. On peut générer automatiquement des éléments standards grâce à la flèche en haut à droite (et s'il y en a trop, on peut toujours en supprimer !).




En double cliquant sur le premier bouton (ou sur le premier élément de menu), on peut associer une réponse à l'événement `Nouveau`, qui va ouvrir une Boîte de dialogue d'ouverture de fichier, dont les paramètres peuvent être précisés (chemin, filtre). La méthode `ShowDialog()` permet alors d'afficher la boîte de dialogue.



```
private void nouveauToolStripButton_Click(object sender, EventArgs e)
{
    OpenFileDialog opfd = new OpenFileDialog();
    opfd.Filter = "Fichiers JPEG|*.jpg";
    opfd.Title = "Choisir le fichier";
    opfd.InitialDirectory = @"F:\Mes images";
    if (opfd.ShowDialog() == DialogResult.OK)
    {

    }
}
```



Lors de l'exécution du programme, on constatera l'effet des modifications qui ont été apportées à la boîte de dialogue.

La fonction d'ouverture de la boîte renvoie un paramètre appartenant à l'énumération `DialogResult` et qui dans ce cas là peut valoir `OK` ou `Cancel`.

On peut alors trouver le nom du fichier :

```
string filename = opfd.FileName;
```

Il y a plusieurs façons d'afficher l'image. Nous allons dans un premier temps le faire à l'aide d'un contrôle PictureBox que nous allons insérer sur la feuille par simple glisser-déplacer et que l'on pourra renommer par exemple en `boiteImage`. On peut lui affecter l'image par l'instruction :

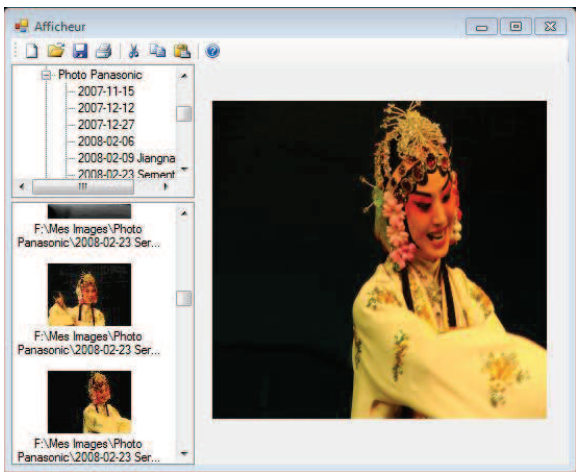
```
boîteImage.Image = Image.FromFile(filename);
```

Le résultat n'est pas forcément convaincant. Reste maintenant à choisir les bons paramètres pour respecter les proportions de l'image affichée, en adaptant les proportions de la `boiteImage`.

## 4.2. Parcourir tout un répertoire

### i. Parcours de répertoire

Nous allons maintenant réaliser une application un peu plus élaborée, ressemblant à la figure ci-contre.




Supprimons tout d'abord la `PictureBox` (par Ctrl+X) et insérons un contrôle `SplitContainer`, qui va venir occuper tout l'espace en le partageant en 2. Puis dans le panneau de gauche rajoutons un second `SplitContainer` et choisissons un fractionnement horizontal.

Remettons à droite la `PictureBox` (par Ctrl+V), et ajoutons en haut à gauche un contrôle `TreeView` appelé `repertoire` (ancré dans le container parent, grâce au paramètre `Dock`) et en bas à droite une `ListView` appelée `fichiers`.

En écrivant une méthode de réponse à l'élément de menu `ouvrirToolStripButton`, nous allons maintenant parcourir le répertoire choisi et insérer son contenu dans l'arbre. C'est un nouveau composant qu'il va nous falloir utiliser :

```
FolderBrowserDialog fld = new FolderBrowserDialog();
fld.Description = "Choisir le répertoire d'images";
fld.RootFolder = Environment.SpecialFolder.MyPictures;
```

Comme tout à l'heure, le contrôle va nous permettre de parcourir les répertoires. Il retournera la valeur sélectionné dans sa propriété `SelectedPath`.

On pourra alors lire la liste des sous-répertoires et la charger dans le contrôle `TreeView` grâce à l'appel de fonction `Directory.GetDirectories()`[21] :

```
foreach (f in Directory.GetDirectories(fld.SelectedPath))
{
    TreeNode tn = new TreeNode(f);
    repertoire.Nodes.Add(tn);
}
```

Pour afficher toute l'arborescence, un traitement récursif paraît nécessaire, mais n'est pas forcément raisonnable : le parcours complet peut être assez long. Il vaut beaucoup mieux ne remplir les sous-arbres qu'au fur et à mesure[22] !



### ii. Liste d'images

Lorsqu'un répertoire est sélectionné (son nom étant dans une chaîne `path`), l'événement `AfterSelect` de l'objet `TreeView` est déclenché. On peut alors obtenir la liste de ses fichiers grâce à la méthode `Directory.GetFiles(path)` et les ajouter aux `Items` du contrôle `ListView fichiers`[23], en mode `Details` (attribut `View`).

En reprenant le code écrit précédemment, on peut réaliser l'affichage des fichiers en parcourant la liste, toujours en utilisant le contrôle `PictureBox`.

Pour remplir la liste des fichiers avec des miniatures des images plutôt qu'avec le nom du fichier, on devra stocker ces miniatures dans une liste d'images, créer explicitement des « `ListViewItem` » et les associer aux fichiers dans la `ListView`, mise en mode `LargeIcon`. Les miniatures peuvent être obtenues par la méthode `GetThumbnailImage` de la classe `Image`. Cf. image ci-contre.

Le code ressemblera à ceci :

```
fichiers.View = View.LargeIcon;
imageList1.ImageSize = new Size(80, 60);
string[] files = Directory.GetFiles(path);
foreach (string s in files)
{
    if (s.ToUpper().EndsWith(".JPG"))
    {
        Image im = Image.FromFile(s);
        imageList1.Images.Add(im.GetThumbnailImage(160, 120, null, IntPtr.Zero));
        im.Dispose();
        ListViewItem lw = new ListViewItem(s, imageList1.Images.Count - 1);
        fichiers.Items.Add(lw);
    }
}
```

Complément 12. *Permettre à l'utilisateur de choisir entre les divers modes (`List`, `Details`, `Large Icon`, ...) de la `ListView fichiers`, en utilisant sa propriété `View`. En mode `List`, on pourra ajouter des colonnes affichant des données comme la taille, la date de création de l'image et permettre de trier sur une colonne par un clic sur l'en tête de la colonne (voir par exemple http://support.microsoft.com/kb/319401).*

## 4.3. Extensions

### i. Afficher d'un diaporama

La création d'un diaporama peut se faire à l'aide d'une nouvelle feuille `Diaporama` dont le constructeur reçoit une liste d'images ou de noms d'images.

L'affichage d'une image peut se faire par la réponse à l'énénement Paint :

```
private void Diaporama_Paint(object sender, PaintEventArgs e)
```

[21] Pour cela la clause `using System.IO;` est nécessaire

[22] On peut aussi remplir avec « un cran d'avance » pour que les répertoires n'apparaissent pas vides s'ils ne le sont pas.

[23] Seulement s'il s'agit de fichiers images naturellement.

```
{
    Image im = Image.FromFile(files[current]);
    e.Graphics.DrawImage(im, this.Bounds);
    im.Dispose();
}
```

Naturellement, ce code déforme l'image si ses proportions ne sont pas exactement celles de l'écran. Il faut donc faire ici quelques calculs (comme précédemment).

Un contrôle `Timer` permettra de déclencher un événement à intervalle régulier (1 seconde par exemple), un « Tick », et la réponse à cet événement fera passer d'une image à la suivante.

Pour que le résultat soit bien un diaporama, une modification de l'affichage de la feuille supprimera son bord, occupant tout l'écran.

On pourra arrêter le diaporama en appuyant sur la touche escape et proposer une interface clavier permettant d'avancer, reculer ou de s'arrêter. Ceci peut-être réalisé en gérant la réponse à l'événement `KeyDown`.



Reste à ajouter dans la feuille principale un élément de menu `Diaporama` et la méthode de réponse qui lancera le dit diaporama. Un paramètre peut aussi permettre de modifier la fréquence de défilement des images.

Complément 13. *Afficher en diaporama le contenu d'un répertoire et de ses sous-répertoires.*

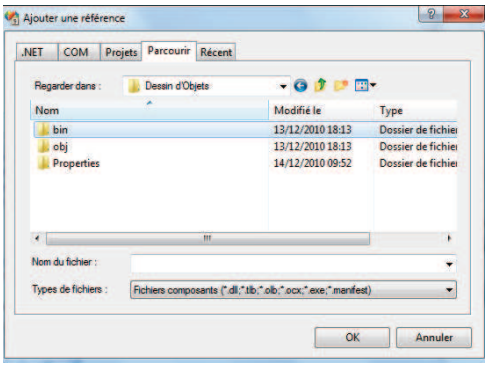
### ii. Utilisation d'une bibliothèque externe

Une image JPEG contient un ensemble d'informations connues sous le nom de paramètres EXIF (pour Exchangeable Image File Format), informations qui sont affichés par de très nombreux utilitaires (l'explorateur de Windows en affiche une partie, d'autres logiciels sont plus exhaustifs).

Selon la source de l'image, ces paramètres peuvent être divers et nombreux, offrant des informations très détaillées : nom et marque de l'appareil photo, vitesse d'obturation, résolution, coordonnées GPS de la prise de vue, voire même le nom des personnes figurant sur la prise de vue (pour peu que quelqu'un ait pris la peine de les saisir…).

L'analyse d'un fichier JPEG pour rechercher ces informations est relativement simple[24], mais nous allons utiliser une bibliothèque de fonctions déjà écrite (beaucoup de variantes sont disponibles sur Internet).

Pour ce faire, nous allons récupérer le fichier ExifLibrary.dll (fourni sur le répertoire étudiant) puis ajouter au projet la référence à la bibliothèque ExifLibrary : `Explorateur de solutions|Référence, clic droit, Ajouter une référence|Parcourir` (cf. image ci-contre).




L'explorateur d'objet permet maintenant de parcourir les classes et fonctions offertes par cette bibliothèque et de trouver la bonne fonction pour extraire les paramètres de l'image.

Pour référencer simplement les objets de la bibliothèque, on devra rajouter la clause :

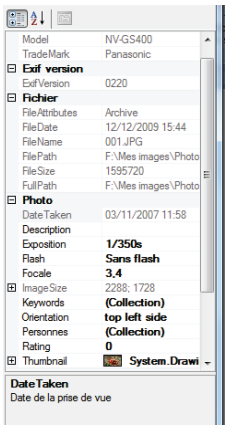
```
using ExifLibrary ;
```

Affichage des informations dans un contrôle prédéfini de type `PropertyGrid` (illustration ci-contre), en utilisant sa propriété `SelectedObject`.

[24] Un fichier JPEG commence par les valeurs 0xff 0xd8 (en hexadécimal). Les données sont stockées dans des blocs comportant un marqueur sur deux octet (dont le premier vaut 0xff) suivi sur deux autres octets de la longueur du bloc. Les principales données utiles sont dans le premier bloc portant le marqueur 0xe1. Il commence (après les deux octets de longueur) par un texte en ASCII sur 5 caractères, qui peut être (en général) soit « JFIF\0 » soit « Exif\0 ». Un octet de remplissage suit. Puis une chaine de deux caractères valant « MM » ou « II » selon que la suite est codée en « Little Endian » ou en « Big Endian ». Suivent alors un entier 0x002A et un offset, normalement 0x00000008. Commencent ensuite une succession de blocs de 12 octets, les « tags », comportant sur les octets 0 à 1 le code du tag, sur les octets 2 à 3 son type, puis de 4 à 7, sa longueur, enfin sur les octets 8 à 11 l'offset de la donnée correspondante ou sa valeur si elle occupe moins de 4 caractères. Le décodage est alors relativement simple, pour peu que l'on dispose des descriptions des tags (on pourra par exemple consulter http://www.sno.phy.queensu.ca/~phil/exiftool/TagNames/EXIF.html). D'autres blocs de type « Exif\0 » peuvent apparaître et contenir par exemple des données au format XMP d'Adobe.

Complément 14. *Utilisation de ces valeurs pour afficher les images dans le sens prévu lors de la prise de vue (portrait ou paysage).*

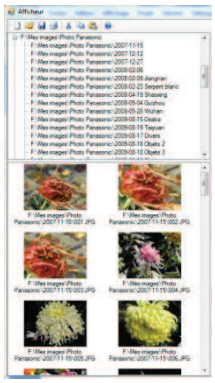
Complément 15. *Un peu d'algorithmique : réécrire (partiellement) la fonction d'analyse du fichier JPEG. En principe les indications de la note suffisent.*



# 5. Contrôles utilisateurs

## 5.1. Création d'un contrôle Choix de fichier

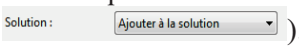
### i. Création du contrôle



Nous avons, au chapitre précédent, utilisé un contrôle de type `TreeView` pour parcourir un répertoire. Ce code peut naturellement être utile dans d'autres applications.

Pour pouvoir effectivement le réutiliser, on peut songer à la technique basique du copier-coller, mais une approche beaucoup plus efficace consiste à créer un Contrôle Utilisateur, c'est-à-dire un contrôle personnalisé qui se comportera comme les contrôles standard fournis par Windows.



Nous allons maintenant créér un nouveau projet de type Bibliothèque de classes et de nom `Controls` que nous allons ajouter à la solution (en faisant le choix appropriée : 

).

Ajoutons à ce nouveau projet un Contrôle Utilisateur que nous appellerons `Navigue`. On constate qu'il dérive du type `UserControl`.

Dans la fenêtre (sans bord) qui est créée insérons un `TreeView` de nom `repertoire` et une liste d'images de nom `listeImages`. L'insertion préalable d'un `SplitContainer` permettra de mieux positionner ces objets.

Une fonction d'initialisation

```
public bool Init(string path)
```

va recevoir le chemin initial et remplir l'arbre (seulement au premier niveau). La sélection d'un nœud de l'arbre représentant un répertoire développe le répertoire (on peut naturellement reprendre le code du chapitre précédent).

On peut maintenant générer le projet (sans l'exécuter). Le contrôle utilisateur que nous venons de créer apparaît maintenant dans la boîte à outils, tout en haut de la liste. Comme d'habitude, on peut l'insérer par tirer-déplacer. Mais pour qu'il soit utile, il faut naturellement appeler sa fonction Init en lui fournissant un nom de chemin.

Notons aussi que si nous faisons afficher tous les fichiers dans l'explorateur de solutions, nous voyons apparaître un nouveau fichier `Controls.dll` : c'est lui qui contient le contrôle que nous avons créé. Il suffira d'importer ce fichier dans un autre projet pour pouvoir réutiliser les contrôles que nous avons créés.

Nous allons maintenant utiliser la liste d'images que nous aurons remplie d'icônes appropriées.[25]

```
this.repertoire.ImageList = this.listeImages;
```

On les utilise par (le preier entier désigne l'image utilisé normalement, le second quand le nœud est sélectionné :

```
TreeNode tn = new TreeNode(d,0,1);
```

### ii. Création d'un événement

Reste maintenant à récupérer dans l'application principale le nom du fichier sélectionné, comme nous l'avons fait en répondant à l'événement `AfterSelect` de l'arbre. Nous allons pour cela créer dans le contrôle un évenement qui pourra être utilisé dans l'application appelante.

Cette création se fait en créant un « délégué »[26], qui a la forme d'une fonction sans corps, déclarée de la façon suivante :

```
public delegate void SelectedIndexEvent(SelectedIndexEventArg ev);
```

La classe `SelectedIndexEventArg` a été créée pour décrire les arguments de cette fonction :

```
public class SelectedIndexEventArg : EventArgs
{
   public string fileName;
   public SelectedIndexEventArg(string file)
   {
```

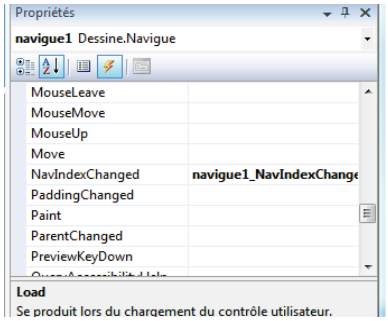
[25] Un ensemble d'icônes est fournie par Visual Studio sous le répertoire C:\Program Files\Microsoft Visual Studio 9.0\Common7\VS2008ImageLibrary\1036\VS2008ImageLibrary.zip

[26] Ce n'est pas le seul usage possible du type « delegate », mais c'est le seul que nous ferons.

```
        fileName = file;
    }
}
```

Ces deux déclarations peuvent être faites dans le corps de la classe `Navigue`, à l'extérieur, ou même dans un fichier séparé pour être réutilisées dans d'autres circonstances.

Dans la classe `Navigue`, on peut maintenant déclarer une variable de type `event` (événement).




```
public event SelectedIndexEvent NavIndexChanged;
```

Lorsque le fichier sélectionné aura été modifié, donc dans le corps de la méthode de réponse à l'événement `AfterSelect` du contrôle `TreeView`, on insérera un appel à l'événement `NavIndexChanged` :

```
private void repertoire_AfterSelect(object sender, TreeViewEventArgs e)
{
    //Code existant
    if (NavIndexChanged != null)
    {
        NavIndexChanged(new SelectedIndexEventArg(name));
    }
}
```



Le test permet de s'assurer que l'application appelante a bien créé une fonction de réponse à l'événement que nous venons de définir (sinon il y a évidemment une erreur).

En générant notre code, on voit maintenant dans le concepteur de vue que le composant possède un nouvel événement que l'on pourra utiliser dans notre formulaire principal comme dans le cas d'un événement prédéfini.

La page de propriétés du contrôle (onglet événement) fait maintenant apparaître ce nouvel événement et nous pouvons l'utiliser en lui associant une fonction de réponse :

```
private void navigue1_NavIndexChanged(SelectedIndexEventArg ev)
{
    string fileName = ev.fileName ;
    // code utilisant le nom du fichier
}
```

On peut maintenant remplacer dans l'application le `TreeView` par notre contrôle personnalisé.

Complément 16. *On peut améliorer le contrôle en rajoutant une ligne de texte dans laquelle l'utilisateur pourra saisir le répertoire initial (penser à placer la `TextBox` et le `TreeView` dans les deux parties d'un `SplitContainer`). On peut remplacer la `TextBox` par une `ComboBox` qui conservera les valeurs des répertoires parcourus.*

Complément 17. *On peut aussi rajouter une propriété permettant de choisir si le contrôle doit afficher fichiers et répertoires ou seulement les répertoires (un filtre comme dans les boîtes de dialogue qui accèdent aux fichiers).*

## 5.2. Recherche de fichiers

### i. Création du contrôle

Comme nous l'avons fait remarquer précédemment, un parcours complet d'un répertoire et de ses sous-répertoires peut se révéler extrêmement long. Nous devons en tenir compte si nous voulons écrire un contrôle capable de rechercher sur un disque l'ensemble des fichiers d'un type donné (toutes les images, tous les fichiers mp3, …).

Un parcours récursif de l'arborescence de fichiers est bien nécessaire cette fois, mais comme sa durée est imprévisible, il ne doit pas être exécuté par le programme principal. La création d'un contrôle spécifique est une possibilité particulièrement simple, qui peut s'inspirer très largement du contrôle `Navigue` écrit précédemment.

Complément 18. *Ecrire le parcours récursif de l'arborescence.*

Reste à faire communiquer le contrôle avec le programme appelant, pour lui indiquer que le parcours est achevé ou pour lui fournir des indications de progression. Pour cela, à chaque nouveau fichier trouvé, le contrôle peut générer un événement qui sera traité par le programme appelant. Le code de parcours a alors la forme suivante :

```
private void Parcours(object path)
```

```
{
    string[] dirs = null;
    string[] files = null;
    try
    {
        dirs = Directory.GetDirectories((string)path);
        files = Directory.GetFiles((string)path);
    }
    catch
    {
        return; // cas des répertoires à accès interdit
    }
    foreach (string d in dirs)
    {
        Parcours(d);
    }
    foreach (string name in files)
    {
        if(!name.ToLower().EndsWith(".jpg")
        {
             if (FileFound != null)
                 FileFound(new FileArg(d));
        }
    }
}
```



Chaque fois que le contrôle trouve un fichier dont le nom se termine par « .jpg », il déclenche l'événement `FileFound` (qu'il faut créer en suivant la même procédure que précédemment). On peut aussi envisager de déclencher un autre événement lorsque le parcours est achevé.

Les données reçues (nom du fichier par exemple) peuvent maintenant être insérées dans un arbre ou affiché dans un `Label` ou une `TextBox` .

Compilons et exécutons le programme à un niveau assez élevé de l'arborescence du disque. On constate que le programme paraît bloqué et que les mises à jour ne se font pas comme on le souhaite.

### ii. Thread d'exécution

Pour régler ce problème, il suffit de faire en sorte que la fonction qui parcourt les répertoires s'exécute indépendamment du programme principal, dans un autre fil de calcul (un « thread ») avec un code très simple comme celui-ci (où `chemin` est le répertoire de départ) :

```
Thread t = new Thread(Parcours);
t.Start(chemin);
```

Le résultat n'est toujours pas convaincant, car la fonction de réponse va lever une exception au moment de mettre à jour l'interface :

```
Cross thread operation not valid: Control "XXXXXXXXXX" accessed from a thread other
than the thread it was created.
```

Ou en français :

```
Opération inter-threads non valide : le contrôle '' a fait l'objet d'un accès à
partir d'un thread autre que celui sur lequel il a été créé.
```

La raison en est simple : le programme principal et le parcours fait dans le contrôle utilisateur s'exécutent de façon indépendante, dans deux fils de calcul distincts. Or, pour éviter les conflits d'accès, les contrôles prédéfinis sont protégés contre les accès provenant d'un autre thread que celui qui les a créés. Et comme la fonction de callback est en fait exécutée par le thread du contrôle utilisateur alors que la `TextBox` a été créée par celui de la fenêtre appelante, il y a forcément problème.

Une solution simple, mais mauvaise, consiste à désactiver la sécurité par l'instruction :

```
TextBox.CheckForIllegalCrossThreadCalls = false
```

La bonne méthode consiste (encore une fois) à créer un délégué[27] :

```
public delegate void SetTextCallBack(TextBox ct, string txt);
```

Et à écrire une fonction du type suivant pour une `TextBox` :

```
public void SetText(TextBox ct, string txt)
```

[27] On peut naturellement faire un peu plus efficace en remplaçant `TextBox` par `Control` dans le code qui suit, ce qui permet de traiter en une fois les différents contrôles.

```
{
    // InvokeRequired required compares the thread ID of the
    // calling thread to the thread ID of the creating thread.
    // If these threads are different, it returns true.
    if (this.InvokeRequired)
    {
        SetTextCallBack d = new SetTextCallBack(SetText);
        this.Invoke(d, new object[] { ct, txt });
    }
    else
    {
        ct.Text = txt;
    }
}
```

A l'endroit où l'on aurait écrit :

```
textBox1.Text = "text";
```

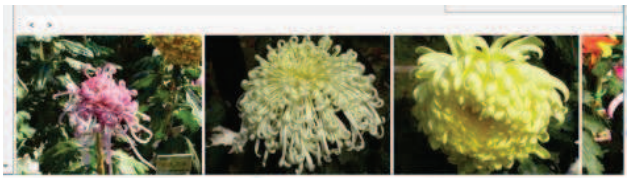
on écrira maintenant :

```
SetText(textBox1, "text");
```



Complément 19. *Pour faire un peu d'algorithmique, on peut demander le nombre total de fichiers (d'un certain type) dans l'arborescence, leur taille totale, le nombre de sous répertoires, la profondeur de l'arborescence…*

### 5.1. Sélecteur d'images



Dans la version précédente, nous avons utilisé une `ListView` pour afficher la liste des fichiers contenus dans un répertoire. L'affichage des miniatures n'était que partiellement convaincant.

Nous allons maintenant créer un contrôle utilisateur qui affiche les images de façon plus satisfaisante.

Pour cela, ajoutons un nouveau projet à notre solution, et créons un contrôle utilisateur. Comme précédemment, une fonction `Init()` permettra de lui transmettre la liste des images à afficher, ce dont se chargera la méthode `Paint()`.

Ecrire le code de la fonction `Paint()`.

Un contrôle de type `ScrollBar` (ou des boutons pour avancer et reculer) permettra de faire défiler la liste (qui le plus souvent ne rentrera pas intégralement dans la surface du contrôle).

Modifier le code de la fonction `Paint()` pour tenir compte de l'action du `ScrollBar`.

Ajoutons ensuite un évènement qui permettra de transmettre à l'application appelante l'image sélectionnée par un clic de souris.

Complément 20. *Ajouter un mode qui permet d'afficher les images sur plusieurs lignes en ajustant leur taille (comme dans une `ListView` de Windows), et qui permette toujours de sélectionner un élément.*

# 6. D'autres notions

## 6.1. Drag and drop

Le glisser-déplacer est une technique très utilisée dans les interfaces graphiques. Nous allons voir sur un exemple comment le mettre en œuvre avec C# dans l'environnement Windows.

Créons une forme avec deux `ListBox`, `listBox1` et `listBox2`. Pour `listBox2`, on précise qu'elle accepte le glisser-déplacer par l'instruction :

```
listBox2.AllowDrop = true;
```

Le déplacement commence quant un élément de la liste de gauche est sélectionné par un clic sur la souris :

```
private void listBox1_MouseDown(object sender, MouseEventArgs e)
{
    if (listBox1.Items.Count == 0)
        return;
    int index = listBox1.IndexFromPoint(e.X, e.Y);
    string s = listBox1.Items[index].ToString();
    DragDropEffects dde1 = DoDragDrop(s, DragDropEffects.Move);
    if (dde1 == DragDropEffects.All)
    {
        listBox1.Items.RemoveAt(listBox1.IndexFromPoint(e.X, e.Y));
    }
}
```



Lorsque la souris arrive sur la seconde liste, celle-ci précise le type de drop qu'elle accepte[28].

```
private void listBox2_DragOver(object sender, DragEventArgs e)
{
   e.Effect = DragDropEffects.All;
}
```

Lorsque l'on relâche la souris la méthode de réponse à l'événement DragDrop vérifie que l'élément « transporté » est bien une chaîne et le cas échéant, l'insère dans la listbox

```
private void listBox2_DragDrop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string str = (string)e.Data.GetData(DataFormats.StringFormat);
        listBox2.Items.Add(str);
    }
}
```

On pourra aussi étudier un exemple de code plus complet sur le site msdn, à l'adresse :

http://msdn.microsoft.com/fr-fr/library/system.windows.forms.control.dodragdrop(v=VS.90).aspx

A quoi sert le bloc `if` dans la méthode `listBox1_MouseDown` ?

Complément 21. *Modifier le code pour que l'élément déplacé ne soit pas forcément supprimé de la liste de gauche.*

Complément 22. *Ajouter la possibilité de ramener les éléments à gauche, par glisser déplacer ou par Ctrl-Z (annulation).*

[28] Un traitement plus complexe est possible, comme par exemple :

```
private void listBox2_DragOver(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        e.Effect = DragDropEffects.All;
    }
    else
    {
        e.Effect = DragDropEffects.None;
    }
}
```

## 6.2. LINQ

LINQ (Language Integrated Query) est un ensemble de fonctionnalités introduites dans Visual Studio 2008 pour ajouter des fonctions de requête à C#. Ces fonctions sont basées sur une syntaxe de type SQL et peuvent être utilisées de façon assez uniforme dans de nombreux contextes.

### i. LINQ to objets

Voici un petit exemple :

```
List<string> liste1 = new List<string>;
liste1.Add("A");
liste1.Add("AA");
liste1.Add("B");
liste1.Add("CA");
liste1.Add("DA");
var query = from s in liste1
            where (s.IndexOf("A") != 0)
            select s;
List<string> l = query.ToList();
```



On peut ainsi exprimer avec une simple requête une recherche qui serait réalisée par un parcours de la liste de la façon classique.

La même approche peut être employée avec une ListBox contenant des chaînes :

```
var items = from s in listBox1.Items.Cast<string>()
            where (s.IndexOf("A") != 0)
            select (s);
l = items.ToList();
```

On peut aussi utiliser des types plus complexes. Soit un type :

```
public class Etudiant
{
    public string Nom { get; set; }
    public List<int> Note { get; set; }
}
```

Construisons une liste d'étudiants :

```
List<Etudiant> etudiants = new List<Etudiant>
{
    new Etudiant {Nom="A", Note= new List<int> {15, 8, 12, 10}},
    new Etudiant {Nom="B", Note= new List<int> {14, 9, 15, 6}},
    new Etudiant {Nom="C", Note= new List<int> {12, 17, 11, 10}},
    new Etudiant {Nom="D", Note= new List<int> {12, 16, 11, 18}},
    new Etudiant {Nom="E", Note= new List<int> {12, 10, 15, 10}}
};
```

La requête suivante va fournir la liste des étudiants ayant des notes supérieures à 14.

```
var requete = from etd in etudiants
                from note in etd.Note
                where note >= 14
                select new {Nom = etd.Nom, note };
var resultat = requete.ToList();
```

Espion 1

| Nom | Valeur | Type |
|---|---|---|
| resultat.ToList() | Count = 7 | System.Collections.Generic.l |
| [0] | { Nom = "A", note = 15 } | <Anonymous Type> |
| [1] | { Nom = "B", note = 14 } | <Anonymous Type> |
| [2] | { Nom = "B", note = 15 } | <Anonymous Type> |
| [3] | { Nom = "C", note = 17 } | <Anonymous Type> |
| [4] | { Nom = "D", note = 16 } | <Anonymous Type> |
| [5] | { Nom = "D", note = 18 } | <Anonymous Type> |
| [6] | { Nom = "E", note = 15 } | <Anonymous Type> |

Ces exemples utilisent des notions nouvelles comme le type var[29], l'initialisation en bloc d'une liste ou la création d'un type anonyme.

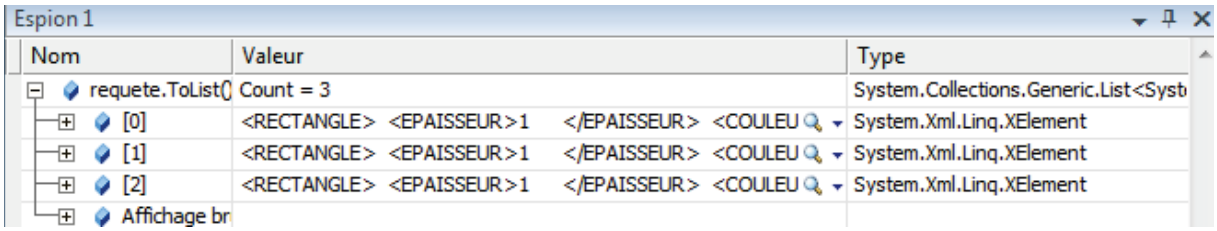
[29] Le type var permet de déclarer des variables dont le type peut être déterminé par le compilateur : les déclarations suivantes sont équivalentes :
var i = 10; // typage implicite
int i = 10; //typage explicite

### ii. LINQ to Xml

L'approche « LINQ » peut aussi être utilisé avec des documents XML. Reprenant l'exemple du chapitre 2, on peut écrire :

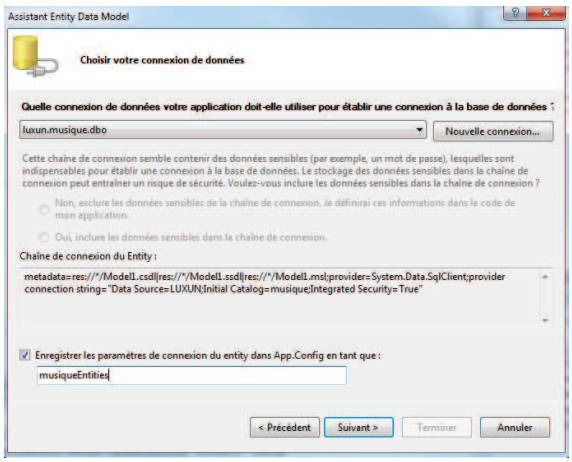
```
XDocument xdoc = XDocument.Load(@"F:\Dessin.Xml");
var requete = from d in xdoc.Descendants((XName)"RECTANGLE")
                                  where d.Name == "RECTANGLE"
                                  select d;
var liste = requete.ToList();
```

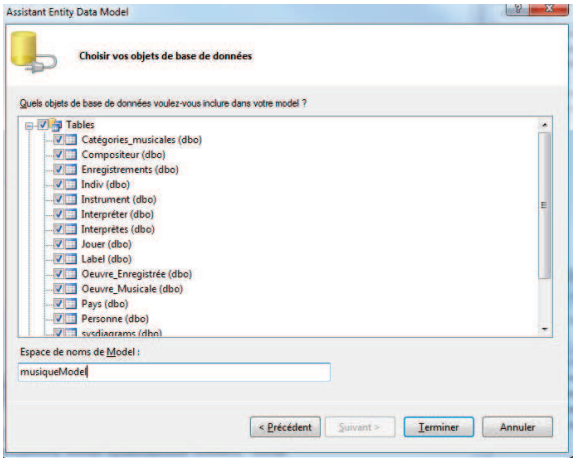
et obtenir le résultat suivant, c'est-à-dire tous les nœuds de type RECTANGLE :



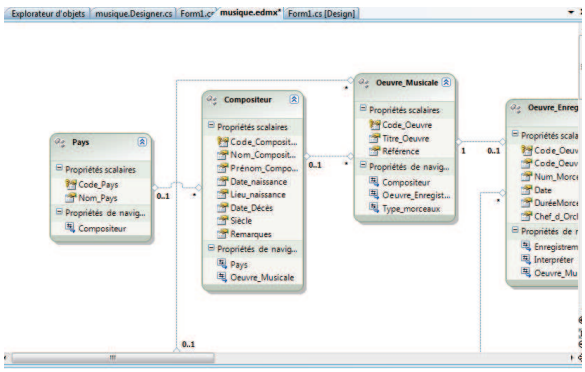

### iii. LINQ to SQL

Ce que nous venons de voir montre clairement que LINQ est inspiré de SQL. On peut effectivement se connecter assez simplement à une base de données en créant un « ADO.NET Entity Data Model » (`Projet|Ajouter un nouvel élément`). Quelques étapes successives nous permettent alors de choisir la base de données et de sélectionner les tables intéressantes




Visual Studio génère alors un modèle de données dont voici un extrait :

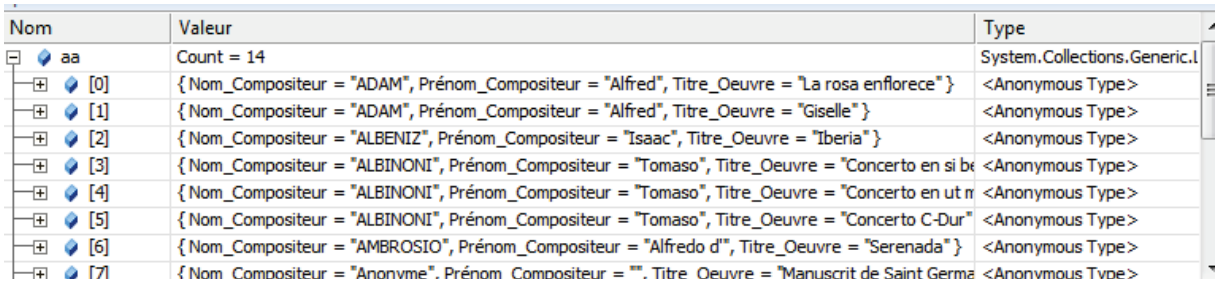



Il crée pour cela une classe pour chaque table sélectionnée dans la base de données, ainsi qu'une classe ayant le même nom que la base et qui servira d'interface dynamique avec la base de données : elle possède une propriété pour chaque table et un ensemble de méthodes permettant d'insérer des enregistrements dans les tables.

On pourra alors écrire des requêtes « relativement » standard pour interroger la base de données :

```
musiqueEntities musique = new musiqueEntities();
var comp = from a in musique.Compositeur
           join b in musique.Oeuvre_Musicale
           on a equals b.Compositeur
           where a.Nom_Compositeura.Nom_Compositeur.StartsWith("A")==0
           orderby a.Nom_Compositeur
           select new { a.Nom_Compositeur, a.Prénom_Compositeur, b.Titre_Oeuvre };
var aa = comp.ToList();
```

Cette requête fournit le résultat attendu, dont voici les premières lignes :



| Nom | Valeur | Type |
|---|---|---|
| aa | Count = 14 | System.Collections.Generic.I |
| [0] | { Nom_Compositeur = "ADAM", Prénom_Compositeur = "Alfred", Titre_Oeuvre = "La rosa enflorece" } | <Anonymous Type> |
| [1] | { Nom_Compositeur = "ADAM", Prénom_Compositeur = "Alfred", Titre_Oeuvre = "Giselle" } | <Anonymous Type> |
| [2] | { Nom_Compositeur = "ALBENIZ", Prénom_Compositeur = "Isaac", Titre_Oeuvre = "Iberia" } | <Anonymous Type> |
| [3] | { Nom_Compositeur = "ALBINONI", Prénom_Compositeur = "Tomaso", Titre_Oeuvre = "Concerto en si b | <Anonymous Type> |
| [4] | { Nom_Compositeur = "ALBINONI", Prénom_Compositeur = "Tomaso", Titre_Oeuvre = "Concerto en ut m | <Anonymous Type> |
| [5] | { Nom_Compositeur = "ALBINONI", Prénom_Compositeur = "Tomaso", Titre_Oeuvre = "Concerto C-Dur" | <Anonymous Type> |
| [6] | { Nom_Compositeur = "AMBROSIO", Prénom_Compositeur = "Alfredo d'", Titre_Oeuvre = "Serenada" } | <Anonymous Type> |
| [7] | { Nom_Compositeur = "Anonyme", Prénom_Compositeur = "", Titre_Oeuvre = "Manuscrit de Saint Germa | <Anonymous Type> |

On peut naturellement réaliser aussi des insertions, des mises à jour ou des suppressions, comme dans le fragment de code ci-dessous :

```
Personne pers = new Personne();
pers.Nom = "A";
pers.Prénom = "B";
musique.AddToPersonne(pers);
musique.SaveChanges();
```



ou celui-ci :

```
Personne pers = (from p in musique.Personne
                 where p.Code == 1
                 select p).First();
pers.MotdePasse = "secret";
musique.SaveChanges();
```