



Concepteur Développeur en Informatique

Développer des composants d'interface

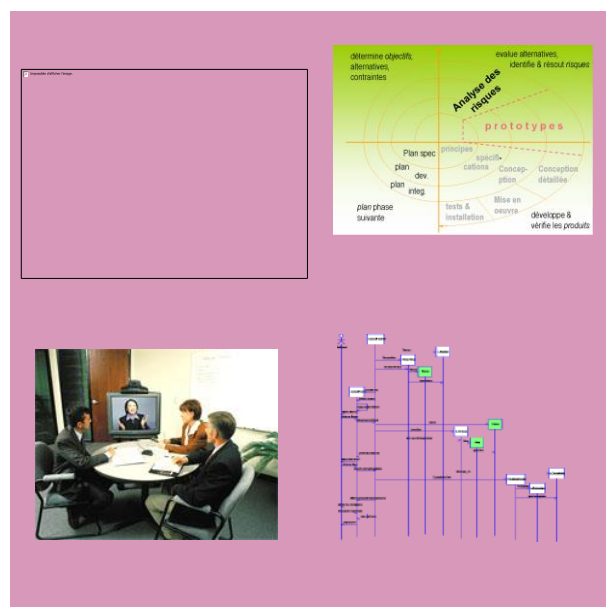
Elaborer des formulaires Windows

Accueil

Apprentissage

PAE

Evaluation



Localisation : U03-E03-S02

SOMMAIRE

1	Introduction	4
1.1	Application Windows	4
2	Les objets graphiques	5
2.1	Les propriétés.....	5
2.1.1	Les propriétés les plus fréquemment utilisées	6
2.2	Les méthodes.....	6
2.3	Les évènements	6
2.3.1	Les événements les plus fréquemment utilisés	7
2.3.2	Un événement peut en cacher un autre	8
3	Création d'une application Windows	9
3.1	Mode d'emploi	9
3.2	Caractéristiques du projet.....	13
3.3	La classe de démarrage	14
3.4	Le code modifiable d'un formulaire	15
3.4.1	Les espaces de noms.....	15
3.4.2	La classe Form	16
3.5	Le code généré par le concepteur (Designer)	17
3.6	Ajout d'un contrôle.....	18
3.7	Les événements	19
3.7.1	Traiter le click du bouton.....	19
3.8	Exemple	21
4	L'IDE Visual Studio	22
4.1	Commenter le code	22
4.2	Manipuler et mettre en forme le code	23
4.2.1	Copier / Coller du code	23
4.2.2	Suivre les modifications de code	23
4.2.3	Isoler des portions de code.....	23
4.2.4	Mise en forme du code	24
4.3	Refactoriser	25
4.4	L'environnement graphique	25
4.4.1	La liste des tâches (<i>smartag</i>).....	25
4.4.2	Les barres d'alignement (<i>snaplines</i>).....	26
4.4.3	Donner la même propriété à plusieurs composants	26
4.4.4	Placement des contrôles les uns par rapport aux autres	26
4.4.5	Positionner automatiquement le passage du focus.....	26
4.4.6	Ancrage des composants par rapport à la fenêtre mère	27
4.4.7	Accoler un contrôle à un bord de fenêtre	27
5	Les fenêtres	28
5.1	Les principales propriétés d'une fenêtre	28
5.2	Les principaux événements d'une fenêtre	30
6	Les différentes formes de dialogue	31
6.1	Application en mode SDI	31
6.2	Application en mode MDI.....	31
6.2.1	Créer une feuille mère	32
6.2.2	Créer une feuille enfant	32

6.2.3	Arranger la disposition des fenêtres filles	33
6.2.4	Fusion du menu des fenêtres filles	33
6.3	Dialogue Modal ou Non	34
6.3.1	Dialogue Modal	34
6.3.2	Dialogue non modal	35
6.4	La classe MessageBox.....	35
7	Généralités sur les contrôles	37
7.1	Création dynamique de contrôles	38
7.2	Gestion des évènements	39
7.2.1	Associer un gestionnaire par programme.....	39
7.3	Gérer le focus.....	40
8	Communiquer des informations entre feuilles	40
8.1	Surcharge du constructeur	41
8.2	Implémentation d'une propriété de formulaire.....	43
	

1 Introduction

Nous allons ici découvrir une nouvelle approche du développement basé sur les événements. Nous parlerons de **programmation événementielle** qui s'oppose à la programmation **séquentielle** découverte initialement.

En programmation séquentielle, les opérations exécutées sont toujours les mêmes et fonction des conditions programmées par le développeur. C'est le développeur qui pilote l'application et qui contrôle le déroulement des opérations.

En programmation événementielle, les actions sont exécutées par réaction aux événements qui surviennent et qui peuvent être émis par le système ou par l'utilisateur.

La programmation événementielle se base donc sur le principe de la détection et la gestion des événements. Elle est mise en œuvre dans le cadre du développement d'interfaces graphiques.

Nous allons ici la découvrir dans le cadre du développement d'applications Windows mais nous retrouverons ces mêmes principes dans le cadre du développement d'applications Web.

Le développement de **composants Windows** repose donc sur une architecture de **programmation événementielle**.

1.1 Application Windows

Une **application WINDOWS** peut se résumer en un **ensemble de feuilles**, initialement blanches.

Chaque feuille va ensuite être spécialisée et devenir une feuille de saisie, un document, une boîte de dialogue etc. Chaque feuille comportera un certain nombre d'objets : zones de données, propositions de choix d'options ou d'actions.

Ces objets s'appellent des **contrôles** (zones de texte, boutons, cases à cocher, options, listes déroulantes, barres d'outils etc.). Ces contrôles vous sont fournis sous la forme de type comme tout composant du système. Ils se distinguent toutefois des autres composants par le fait qu'ils se **dessinent** et disposent d'une **interface graphique**.

Les **actions** de l'opérateur (déplacement du curseur, de la souris, saisie ou modification ...) sur chacun de ces contrôles font partie du dialogue et sont à l'origine **d'événements** qui peuvent être gérés afin d'exécuter une **séquence d'instructions**.

A chaque événement correspond une séquence isolée de programme, un sous-programme, effectuant un traitement en fonction du contexte.

Une application de type formulaire Windows est donc composée d'une multitude de **couples événement/contrôle**.

2 Les objets graphiques

Nous distinguerons :

Les feuilles (Form) ou fenêtres :

- Feuille principale
- Feuilles filles (une par document)
- Feuilles modales ou boîtes de dialogue.

Des principaux contrôles :

- Etiquettes ou labels, pour dénommer des données
- Zones de texte, utiles pour contenir des données
- Boutons de commande
- Boutons d'option dits boutons radio
- Cases à cocher
- Conteneurs pour contenir d'autres contrôles
- Listes simples, déroulantes ou combinées (combobox)
- Barres de défilement

Tous les contrôles de l'espace de noms **System.Windows.Forms** héritent de la classe **System.Windows.Forms.Control**.

C'est la raison pour laquelle ils possèdent un grand nombre de caractéristiques communes.

Vous pouvez créer des contrôles dérivés à partir de ces classes de base pour étendre les fonctionnalités de ceux-ci. Nous retrouvons ici les concepts de la programmation orientée objet.

2.1 Les propriétés

Chaque objet possède des **propriétés** qui définissent son identification, sa position, ses couleurs, son état, sa valeur, les possibilités de le modifier, ses liens avec d'autres objets, etc.

La plupart des propriétés peuvent être définies ou modifiées lors de la conception (design-time) ou lors de l'exécution du programme (run-time).

Quelques propriétés identiques se retrouvent dans la plupart des contrôles comme les propriétés de forme, positionnement, couleurs et polices de caractères.

Le tableau ci-dessous en présente quelques-unes :

Couleurs	Police	Position	Tabulation
BackColor	Font.Bold	Height	TabIndex
ForeColor	Font.FontFamily	Width	TabStop
	Font.Height	Location.X	
	Font.Italic	Location.Y	
	Font.Name		
	Font.Size		
	Font.Underline		

2.1.1 Les propriétés les plus fréquemment utilisées

Name : C'est le nom de la variable ou structure associée à l'objet. C'est avec ce nom que vous désignerez un contrôle lors de l'exécution des traitements.

Text: C'est le nom qui apparaît sur la feuille :

- Libellé de la feuille dans le bandeau
- Libellé d'un label (ou étiquette)
- Libellé du Bouton radio ou de la case à cocher
- Texte d'une textbox
- Libellé de l'objet data qui représente une base de donnée

Il est souvent exprimé sous la forme d'un littéral de type chaîne constant mais il peut être modifié dynamiquement lors de l'exécution du programme.

Visible : Cette propriété permet de masquer un contrôle qui n'a momentanément pas de sens dans le contexte. Cette propriété vaut True ou False.

Enabled : Un contrôle, tout en étant visible, peut ne pas être utilisable. Il sera alors **disabled** et ne pourra obtenir le focus.

Par exemple un bouton "Ajout" ou "Suppression" alors qu'il n'y a aucun élément à ajouter ou supprimer.

Dans ce cas, le contrôle est inaccessible par l'opérateur et apparaît en grisé.

Cette propriété vaut True ou False.

Parent : Ce contrôle n'est accessible qu'en exécution et en lecture seulement. Il indique le nom du conteneur du contrôle courant. Nous reviendrons sur cette notion de conteneur.

TabIndex : Définit l'ordre de passage d'un contrôle à un autre (à condition que sa propriété TabStop soit à true) avec les touches de tabulation.

TabStop : Indique si le bouton peut recevoir le focus.

Focus : propriété du contrôle qui est actif. Nous disons que le contrôle actif a le focus. L'ordre de tabulation est l'ordre dans lequel un utilisateur déplace le focus d'un contrôle à l'autre en appuyant sur la touche TABULATION.

2.2 Les méthodes

Un objet possède également des **méthodes**.

Certaines méthodes sont communes à tous les contrôles, comme la méthode **Focus** qui permettra de rendre actif un contrôle particulier, ou spécifiques à une classe de contrôle, comme la méthode **Clear** de la **TextBox** qui efface tout le texte du contrôle zone de texte.

2.3 Les évènements

Un objet réagit à des **événements**. Si une procédure est abonnée à un événement, celle-ci sera exécutée lors de la survenance de l'événement.

Comme avec les propriétés, on retrouve souvent les mêmes possibilités d'événements pour les contrôles. Les événements sont les éléments essentiels à la programmation sous Windows dite aussi programmation événementielle.

2.3.1 Les événements les plus fréquemment utilisés

GotFocus

Se produit lorsque le contrôle prend le *focus*.

Enter

Se produit lorsque le contrôle devient contrôle actif du formulaire.

Pour effectuer un traitement préalable à une saisie ou modification : dès que l'on met un pied dans le contrôle (Avec la souris en cliquant, par tabulation, avec les flèches, ...)

TextChanged

Se produit lorsque la valeur de la propriété Text a été modifiée.

Permet de déclencher des contrôles très réactifs.

Beaucoup de traitements gagneront toutefois plutôt à être faits lors de l'événement Leave lorsque l'on quitte le contrôle.

KeyDown, KeyPress, KeyUp

Ces événements sont utiles à traiter particulièrement lorsque l'on veut gérer finement la saisie au clavier caractère par caractère.

KeyDown et KeyUp, utilisés pour toute touche, servent plus particulièrement à traiter les touches spéciales, touches de fonction ou combinaison avec *Ctrl*, *Maj* (ou Shift) et *Alt*.

Ces événements sont exploitables à tout moment dans une feuille pour filtrer les raccourcis claviers (ou shortcuts), actions de l'opérateur alternatives à l'appui sur un bouton.

MouseDown, MouseUp, MouseMove, MouseEnter, MouseHover, MouseLeave

Ces événements permettent de contrôler finement les actions de la souris.

- MouseDown : Un bouton de la souris est enfoncé ;
- MouseUp : Un bouton de la souris est relâché ;
- MouseMove : La souris se déplace au-dessus du contrôle
- MouseEnter : La souris entre dans la zone du contrôle
- MouseLeave : La souris quitte la zone du contrôle
- MouseHover : La souris marque un temps d'arrêt dans le contrôle

Leave

Se produit lorsque le contrôle cesse d'être le contrôle actif du formulaire.

Déclenche les traitements lorsque l'on quitte un contrôle.

Validating, Validated

Validating se produit lorsque l'utilisateur veut quitter le contrôle, sous réserve que le contrôle où il souhaite se diriger ait la valeur de sa propriété CauseValidation à true.

Validated se produit lorsque l'événement Validating s'est terminé avec succès.

Utilisés pour contrôler la validité d'un champ, il est possible d'interrompre la poursuite de la

validation d'un contrôle en cas d'erreur.

Click

Utilisé principalement avec les Boutons de commande et les listes d'option.

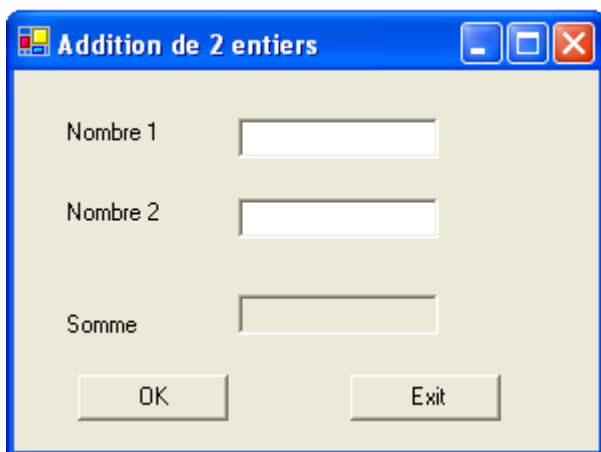
DoubleClick :

Utilisé pour sélectionner un élément d'une liste. Déclenche en général une vue de détail sur l'élément considéré.

DragDrop, DragEnter, DragLeave, DragOver

Tous ces évènements sont relatifs à la technique de glisser déposer (Drag and Drop)

2.3.2 Un événement peut en cacher un autre



Le curseur étant dans la zone de saisie "Nombre1", nous pouvons aller dans Nombre2 de plusieurs manières :

Click dans Nombre2
Tabulation ou ↓

Nous devons donc envisager de traiter tous les cas possibles de passage de Nombre1 à Nombre2, en étant sûr de ne traiter qu'une seule fois l'action de l'opérateur.

Pour Nombre1 nous devons envisager l'occurrence possible des événements :

- Leave Abandon du focus du champ,
- ou KeyPress Filtre de la touche TAB

Pour Nombre2 :

- Enter Entrée dans le contrôle (quelle origine ?)
- MouseEnter Arrivée avec la souris
- Click Clic de souris.

Il n'est pas toujours évident de faire le bon choix pour associer gestionnaire et événement.

3 Création d'une application Windows

Visual Studio a été entièrement remanié de manière à standardiser les méthodes de développement à destination des deux environnements qui cohabitent de plus en plus étroitement aujourd'hui :

- Les applications Windows.
- Les applications Internet.

Dans les deux cas, une application se compose de fenêtres nommées **Forms**.

Pour les applications Windows, il s'agit de **WinForms** que l'on désignera sous le terme **feuilles** ou **formulaires**.

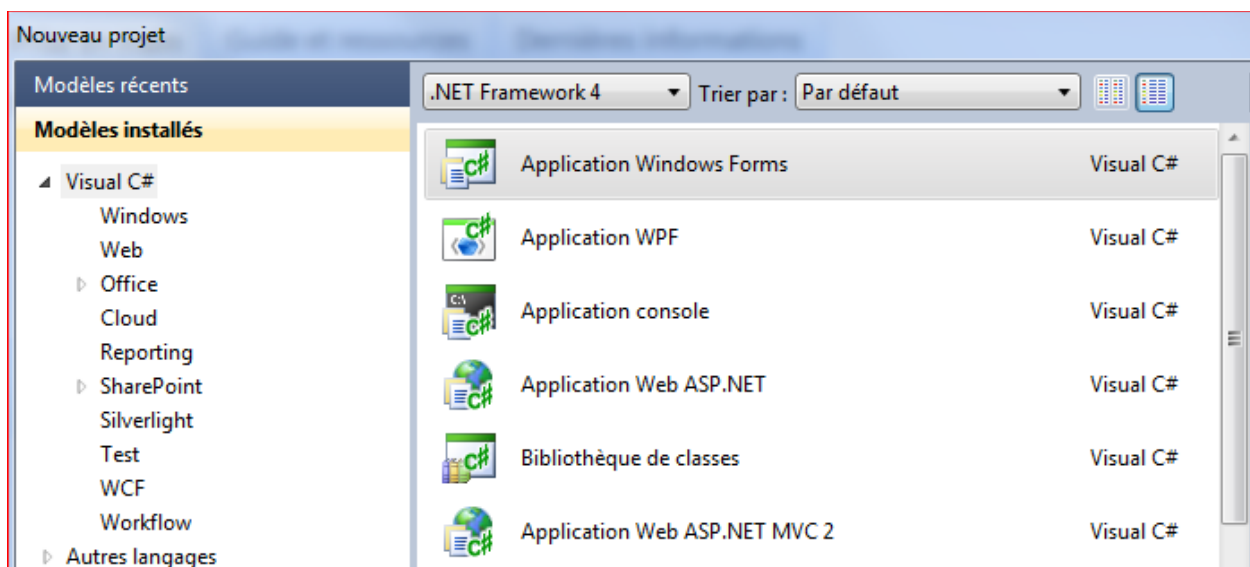
Pour les applications Web, il s'agit de **WebForms**, que l'on nommera **pages**, pour les applications dédiées au Web.

Mais le développement des interfaces présente des similitudes dans les deux cas.

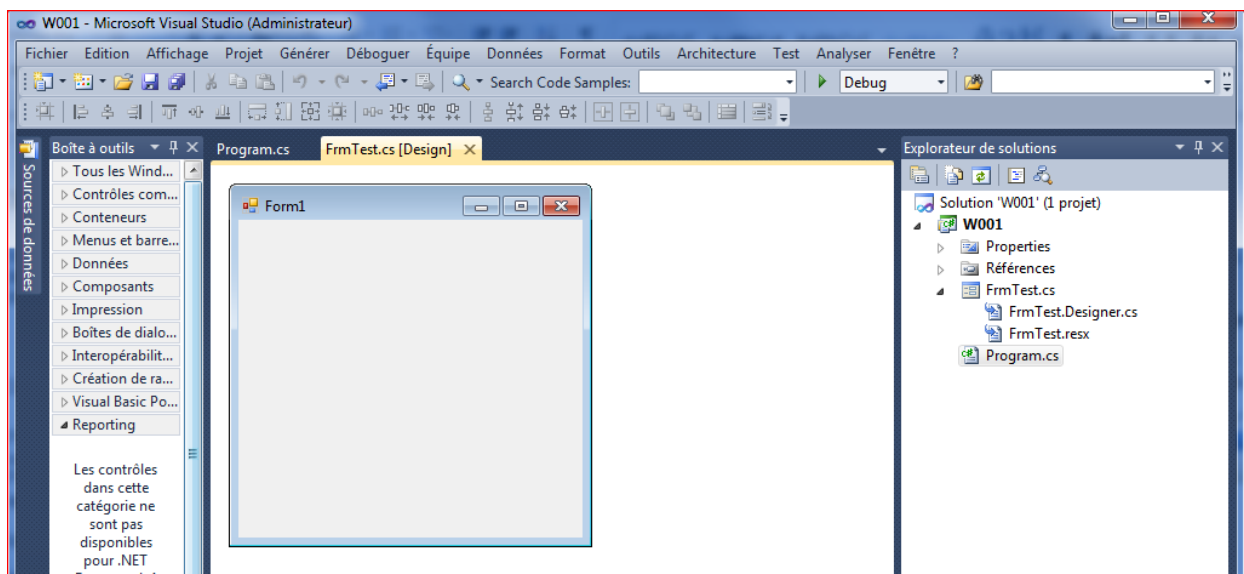
3.1 Mode d'emploi

Un projet est créé sous l'environnement Visual Studio .NET qui se charge de créer les répertoires.

Créer un nouveau projet par le menu fichier, en choisissant le type "**Application Windows Forms**".

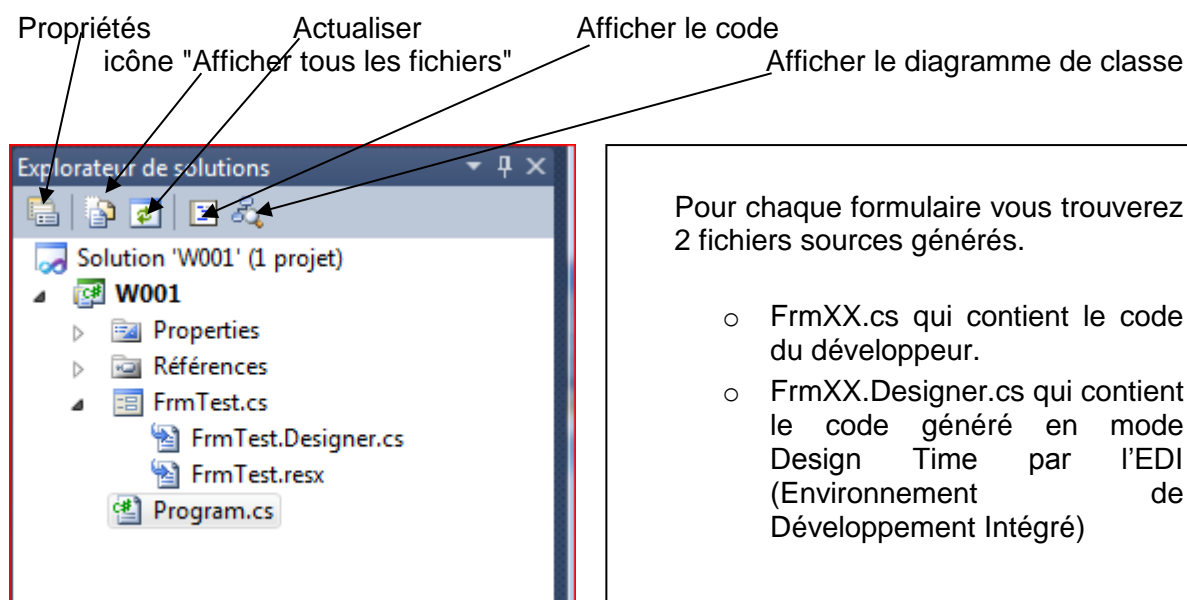


Choisir le nom de votre projet, son emplacement et les caractéristiques de la solution.



Renommer le fichier Form1.cs en un nom mnémorique caractérisant l'application, dans cet exemple FrmTest

Cette action sera à effectuer pour chaque feuille codée dans une application.



La propriété **Name** de la feuille, qui sert à adresser la feuille dans le code aura été modifiée à frmTest.

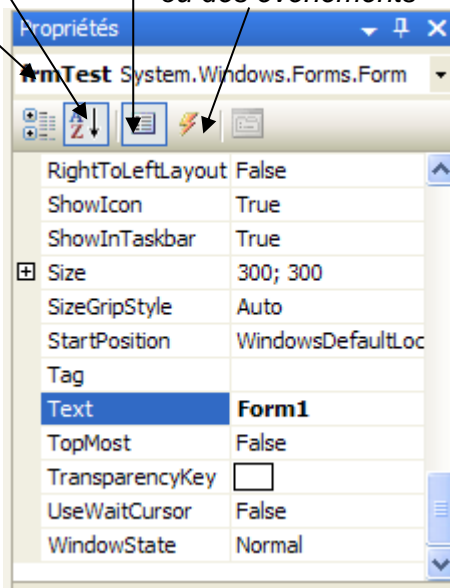
Donner un titre à la feuille en modifiant la propriété **Text** dans la fenêtre des propriétés.
Cette action devra être répétée pour chaque feuille traitée dans une application.

Cette fenêtre comporte la liste des propriétés de l'objet sélectionné (ou de la feuille si aucune sélection n'est faite sur un contrôle).

S'il n'est pas intéressant d'agir à ce stade sur les propriétés de taille ou position du contrôle, on y définit en revanche son nom, sa valeur et son état initial et, peut-être, quelques autres valeurs ...

Classement par thèmes ou
Classement alphabétique

.... des propriétés
ou des événements



La fenêtre des propriétés n'est renseignée que si la feuille est en mode design.

La création d'un projet d'Application Windows suscitent la création de plusieurs éléments :

- les fichiers qui décrivent le projet (.sln, .csproj). , une solution pouvant contenir plusieurs projets
- le fichier qui contiendra le code C# (FrmTest.cs) modifiable par le développeur
- et celui contenant le code C# généré par le concepteur Visual Studio (FrmTest.Designer.cs)
- le fichier XML (.resx) où sont stockées les ressources -données, images,...- locales au formulaire FrmTest.

Ces ressources *resx* peuvent être modifiées selon des critères de "culture" qui particularisent par exemple la langue, sous Visual Studio, sans nécessiter la recompilation de la page.

Toute nouvelle feuille créée donnera toujours lieu à ces deux mêmes types de fichiers (ouvrir par l'icône "Afficher tous les fichiers" de l'explorateur de projet)

La fenêtre de code modifiable par le développeur peut être atteinte :

- En cliquant sur l'icône Afficher le code dans l'explorateur de solution
- En cliquant droit puis Afficher le code sur le fichier frmTest.cs dans l'explorateur de solution

La fenêtre de code générée par Visual Studio peut être atteinte par le menu contextuel du fichier Form1.Designer.cs puis **Afficher le code** dans l'explorateur de solution

On pourra choisir un élément de code particulier dans la liste déroulante de droite.

Les icônes affichées représentent chacun un élément différent, quelquefois précédé d'une icône de signalisation, indiquant leur accessibilité :

Quelques exemples ...

<i>Icônes</i>	<i>Description</i>	<i>Icônes</i>	<i>Description</i>
	Espace de Nom		Champ ou variable
	Classe		Protégé
	Méthode ou fonction		Privé
	Propriétés		Publique

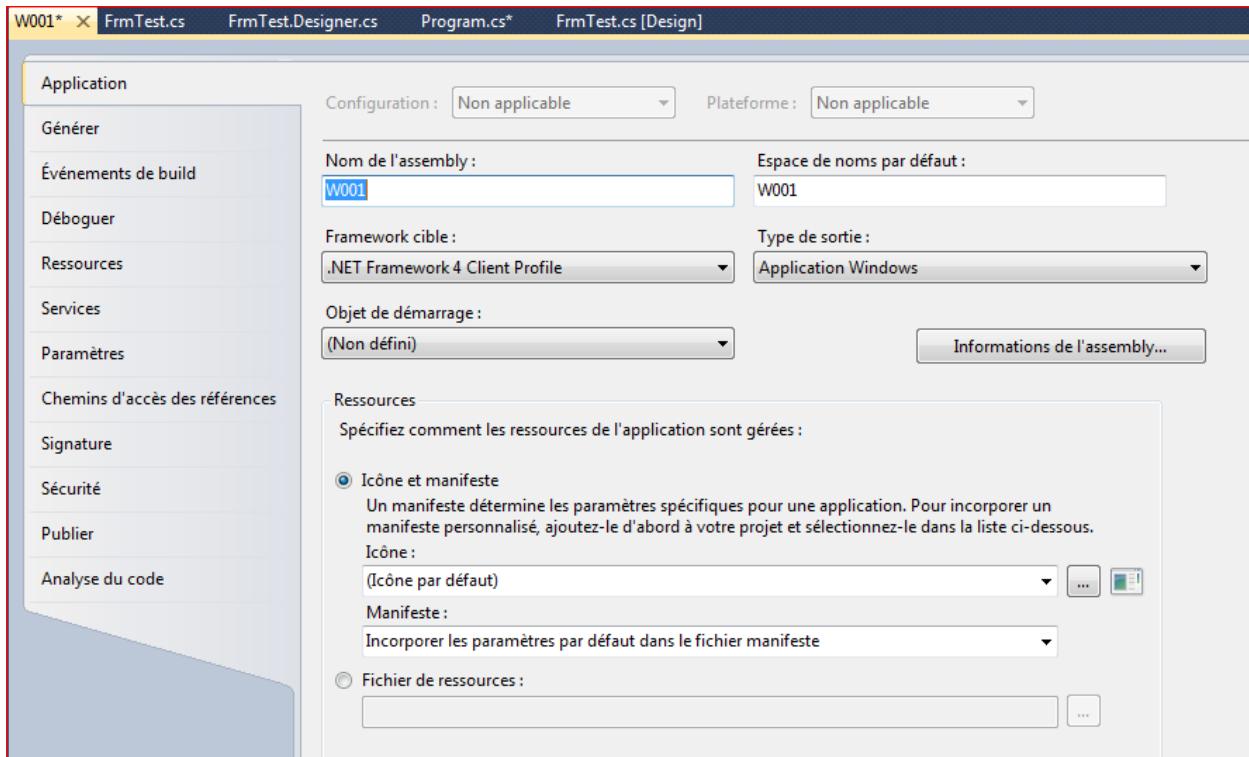
A noter :

- la génération du projet (= la compilation de tous les sources) entraîne la sauvegarde des sources par défaut (paramétrable dans Outils / Options / Projets et Solutions / Générer et Exécuter)
- le lancement de l'application depuis l'EDI entraîne une génération préalable si cela le nécessite (fichiers sources modifiés récemment)
- la génération crée un fichier .exe rangé dans le répertoire bin/release s'il n'y a pas l'option de DEBUG ou dans le répertoire bin/debug dans le cas contraire (défaut)
- ces options DEBUG ou RELEASE du projet et le chemin de sortie de la génération se modifient depuis la fenêtre des propriétés du projet
- La version Debug génère des informations de débogage sous la forme de fichiers .pdb. Elle est utilisée pendant la phase de développement ; la version Release est destinée à être déployée : elle est entièrement optimisée et ne contient aucune information de débogage

3.2 Caractéristiques du projet

Les ressources globales au projet sont stockées dans un fichier Resources.resx présent dans le répertoire Properties du projet, modifiable dans les propriétés du projet.

La fenêtre des propriétés du projet W001 permet d'ouvrir sa *page des propriétés* depuis sa dernière icône à droite.



A noter :

1. Il est possible de choisir l'objet de démarrage de l'application (ici W001.Program).
Il est aussi envisageable de modifier la feuille de démarrage. Mais le point d'entrée de l'application est la méthode statique Main.
2. Il est possible de modifier le nom du fichier exécutable résultant de la génération en modifiant le nom de l'assembly.
Il est aussi envisageable de modifier le nom de l'espace de noms par défaut.

3.3 La classe de démarrage

La classe Program et sa méthode static Main est le point d'entrée de l'application. Vous connaissez déjà ce principe.

```
static class Program
{
    /// <summary>
    /// Point d'entrée principal de l'application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new FrmTest());
    }
}
```

La méthode Main appelle la méthode **Run** de la classe Application.

Celle-ci permet de lancer le traitement des messages Windows envoyés à l'application.

Le système crée une boucle de messages sur l'application.

Le formulaire FrmTest est ensuite créé via son constructeur par défaut.

La méthode InitializeComponent() est alors invoquée. Celle-ci est générée par l'éditeur.

L'attribut **[STAThread]** (STA pour Single Threaded Apartment) signifie que seul le thread de la fenêtre pourra accéder aux composants de la fenêtre. Cet attribut est géré par la classe STAThreadAttribute qui dérive de la classe System.Attribute.

La méthode **EnableVisualStyles()** active les styles visuels pour l'application.

La méthode **SetCompatibleTextRenderingDefault()** active le rendu de texte par défaut pour les nouveaux contrôles.

3.4 Le code modifiable d'un formulaire

Une application Windows IHM est composée d'un ensemble de fenêtres qui se présentent à l'écran, soit directement au démarrage, soit à la demande de l'utilisateur.

Chaque feuille est fractionnée sur 2 fichiers sources (en C# -extension .cs pour CSharp- dans notre cas).

Chaque fichier source contient une section de la définition de classe, et toutes les parties sont combinées lorsque l'application est compilée. La définition de classe est fractionnée grâce au modificateur de mot clé **partial**.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace W001
{
    public partial class FrmTest : Form
    {
        public FrmTest()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
        }
    }
}
```

3.4.1 Les espaces de noms

Un **espace de noms** ("name space") agit comme un conteneur de types. Il permet de ranger nos classes, interfaces, structures, énumérations au sein d'un assemblage. Ces espaces sont organisés hiérarchiquement et fonctionnellement.

En fonction des modèles de projet, des directives **using** sont générées par défaut et font référence aux espaces de noms couramment utilisés dans ce type de projet. Les **directives using** du fichier source dispensent de qualifier l'utilisation d'un type dans le code source si ce type fait partie d'un de ces espaces de noms.

La directive **using System.ComponentModel;** permet ainsi d'utiliser sa classe *Container* en la nommant simplement **Container** et non pas "**System.ComponentModel.Container ()**".

Cependant la notation entièrement qualifiée reste utile dans le cas où deux espaces de noms présenteraient des classes de même nom.

Quelques exemples d'espaces de noms.

- **System** : Contient des classes fondamentales et des classes de base qui définissent les types de données référence et valeur (string, String, int). Il contient aussi de nombreux espaces de noms de deuxième niveau.
- **System.Collections.Generic** : Contient des interfaces et des classes qui définissent différentes collections d'objets, telles que des listes, des files d'attente, des tableaux de bits, des tables de hachage et des dictionnaires.
- **System.ComponentModel** : Fournit des classes qui sont utilisées pour implémenter le comportement au moment de l'exécution et au moment du design des composants et des contrôles.
- **System.Data** : Permet d'accéder aux classes qui représentent l'architecture ADO.NET
- **System.Drawing** : Permet d'accéder aux fonctionnalités graphiques de base de GDI+.
- **System.Text** : Contient des classes pour manipuler les caractères.
- **System.Windows.Forms** : Contient des classes permettant de créer des applications Windows qui profitent pleinement des fonctionnalités élaborées de l'interface utilisateur disponibles dans le système d'exploitation Microsoft Windows.

3.4.2 La classe Form

La classe **Form** est utilisée pour créer des fenêtres à document unique (SDI, Single Document Interface), des boîtes de dialogues (DialogBox) ou des fenêtres multi-documents (MDI, Multiple Document Interface).

Ici la classe est dérivée en une classe **FrmTest** qui permettra de personnaliser notre fenêtre.

La méthode **frmTest()** est le **constructeur** de la classe.

Ce constructeur invoque la méthode **InitializeComponent()** contenant le code d'initialisation du formulaire, généré par Visual Studio et généré dans le fichier frmTest.Designer.cs.

Cette méthode d'initialisation sera enrichie au fur et à mesure de l'ajout de contrôles et composants à la feuille. Initialement, elle ne comporte que les initialisations de la feuille.

C'est à partir de cette classe FrmTest que sera créé en mémoire un objet feuille lors du démarrage de l'application (ou par la suite pour d'autres feuilles de l'application, à la demande de l'utilisateur).

On utilisera pour ce faire la classe **Application** dont le rôle est de fournir les méthodes statiques permettant de démarrer et arrêter une application de type Windows, et des propriétés statiques permettant d'obtenir des informations telles que le chemin de l'exécutable, sa version, ...

Ainsi la méthode **Run** de **Application** permet de démarrer une boucle de messages Windows puis d'afficher une instance du type de formulaire passé en argument de la méthode.

```
Application.Run(new FrmTest());
```


3.5 Le code généré par le concepteur (Designer)

Le code généré par le concepteur se trouve dans le fichier FrmTest.Designer.cs.

```
partial class FrmTest
{
    /// <summary>
    /// Variable nécessaire au concepteur.
    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    /// Nettoyage des ressources utilisées.
    /// </summary>
    /// <param name="disposing">true si les ressources managées doivent être supprimées ; sinon, false.</param>
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

    #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.SuspendLayout();
        //
        // Form1
        //
        this.AutoScaleMode = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(284, 262);
        this.Name = "Form1";
        this.Text = "Form1";
        this.Load += new System.EventHandler(this.Form1_Load);
        this.ResumeLayout(false);

    }

    #endregion
}
```

Vous pouvez prêter une attention particulière à la déclaration de la variable components. Elle est de type **IContainer**. Il s'agit d'une interface.

Cela indique que la variable container pourra référencer tout type qui implémente l'interface IContainer.

Les composants qui contiennent d'autres composants (conteneurs) implémentent cette interface.

Les conteneurs sont des objets qui encapsulent et effectuent le suivi de zéro ou plusieurs composants. Vous pouvez utiliser des composants et des conteneurs dans divers scénarios, notamment des scénarios à la fois visuels et non visuels.

Les composants d'un conteneur sont suivis dans une liste FIFO (premier entré, premier sorti), qui définit également leur ordre dans le conteneur. Les composants ajoutés sont insérés à la fin de la liste.

La méthode **Dispose** sera exécutée à la fermeture du formulaire et permet de libérer les ressources.

Dans la méthode **Initializecomponent** sont codées toutes les descriptions des objets graphiques générées par le concepteur Visual studio.

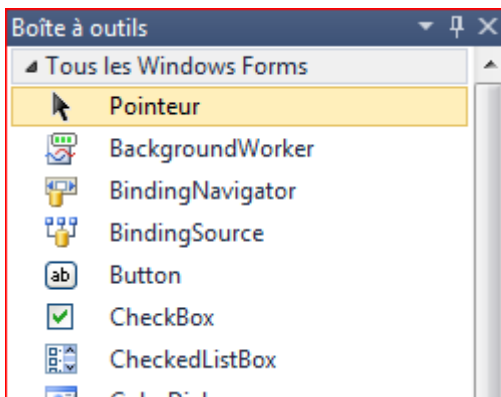
On retrouve le titre de la fenêtre donné précédemment dans la fenêtre des propriétés (propriété **Text**), et le nom de la feuille modifié par sa propriété **Name**

Les méthodes **SuspendLayout** et **ResumeLayout** sont utilisées en tandem pour supprimer les événements **Layout** multiples lorsque vous ajustez plusieurs attributs du contrôle (leur place, leur taille, leur couleur, leur fonte, leur contenu pour les contrôles conteneur, etc ...).

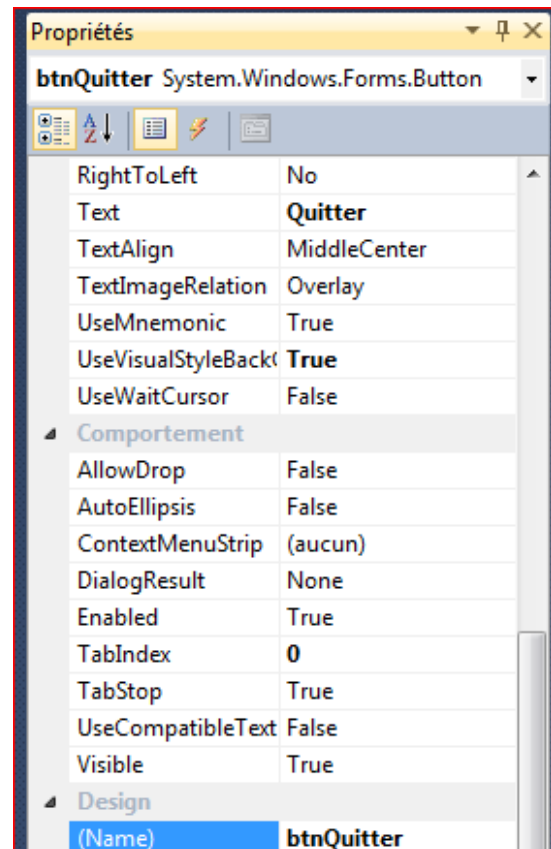
La surface ne doit pas être systématiquement rafraîchi à chaque modification de propriété qui affecte le rendu visuel.

3.6 Ajout d'un contrôle

Ici, nous allons ajouter un premier contrôle de type **Button** à notre formulaire.



Le choix du composant graphique se fait dans la boîte à outil de Visual Studio, en glissant et déposant le bouton de l'onglet sur la feuille.



Vous devez ensuite modifier les propriétés Name (btnQuitter) et Text (Quitter) du contrôle sur la feuille.

Observez le code généré au niveau du fichier source.

On s'aperçoit qu'un champ btnQuitter du type **Button** est déclaré dans la classe FrmTest.

```
private System.Windows.Forms.Button btnQuitter;
```

```
this.btnQuitter = new System.Windows.Forms.Button();
this.btnQuitter.Location = new System.Drawing.Point(12, 210);
this.btnQuitter.Name = "btnQuitter";
this.btnQuitter.Size = new System.Drawing.Size(75, 23);
this.btnQuitter.TabIndex = 0;
this.btnQuitter.Text = "Quitter";
this.btnQuitter.UseVisualStyleBackColor = true;
```

Une instance du type **Button** est créée, stockée dans le champ btnQuitter dans la méthode d'initialisation et ses propriétés modifiées en fonction des valeurs définies en mode Conception.

3.7 Les événements

Les événements peuvent être déclenchés par l'action d'un utilisateur via son clavier ou sa souris, par le programme lui-même ou par le système (timers par exemple).

Le programmeur peut décider de gérer ou non un événement en lui associant du code spécifique, appelé gestionnaire de l'événement.

Il doit donc dans un premier temps écrire la méthode C# qui va traiter cet événement, puis dans un deuxième temps associer cette méthode à l'événement.

L'événement peut d'ailleurs être associé à plusieurs méthodes qui seront toutes exécutées lorsque celui-ci surviendra.

Plusieurs événements peuvent être aussi associés à un même gestionnaire.

Il faut donc mettre en place un mécanisme qui va permettre de relier un objet « **abonné** » à un objet « **annonceur** » qui va déclencher un événement.

Lorsque l'événement survient chez l'annonceur, tous les abonnés sont prévenus et exécutent le traitement prévu dans le gestionnaire. Si l'événement n'a pas d'abonné, aucune opération particulière n'est exécutée. L'événement n'est pas géré.

Le .NET Framework utilise la technique du **délégué d'événement** pour connecter un événement à son gestionnaire. Un délégué est un type particulier de variable qui contient des appels à des méthodes.

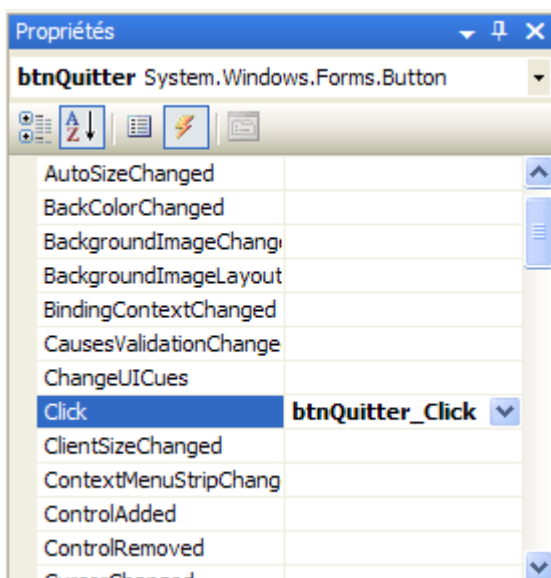
On crée ce délégué de type **System.EventHandler** en lui donnant la référence à la méthode de traitement de l'abonné. Ensuite ce délégué est ajouté à l'événement de l'annonceur.

Les délégués permettent de préciser la méthode à invoquer sur la survenu d'un événement.

Les événements sont définis au niveau de la classe. Il s'agit d'un type de membre de la classe au même titre que les propriétés ou les méthodes. Par contre le concepteur de la classe ne sait évidemment pas quelles seront les opérations à exécuter lorsque l'événement sera déclenché. D'où le recours aux délégués.

Nous aborderons la définition des événements lors d'une prochaine séance de formation consacrée à la programmation Objet.

3.7.1 Traiter le click du bouton



Le clic étant l'événement le plus courant du bouton, il suffit de double cliquer sur le bouton depuis la fenêtre de conception pour que le code soit généré **automatiquement** par Visual Studio, ou alors par le biais de la fenêtre de propriétés, double cliquer sur l'évènement choisi.

Nous aurons donc deux points à observer :

- La création du gestionnaire
- La création du délégué

Le squelette de la méthode de traitement de l'événement est généré au niveau de l'annonceur. Cette méthode est privée et propre à l'annonceur, le formulaire. L'usage est de normaliser, autant que faire se peut, les noms de gestionnaires ainsi *annonceur_événement*.

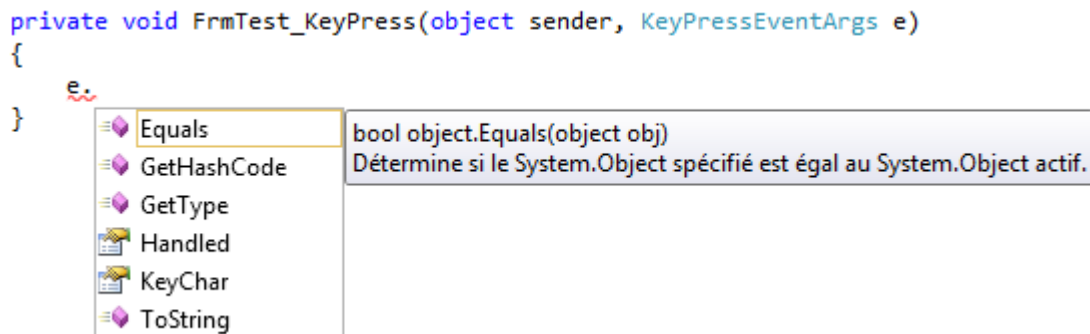
Vous pouvez observer la **signature du gestionnaire** : il reçoit deux paramètres en arguments :

- **Object sender** : sender fait référence à l'annonceur qui est à l'origine de l'événement
- **EventArgs e** : e représente les données de l'événement. Ce sont les informations transmises avec un événement. Toujours de type **EventArgs** ou d'un type dérivé de **EventArgs**.

```
private void btnQuitter_Click(object sender, EventArgs e)
{
    |
}
```

Observons maintenant un gestionnaire dont le squelette a été généré à partir d'un événement KeyPress.

```
private void FrmTest_KeyPress(object sender, KeyPressEventArgs e)
{
    e.
}
```



Les informations qui accompagnent l'événement sont transmises dans e qui est de type **KeyPressEventArgs**, type dérivée de **EventArgs**. La documentation de Microsoft ne dit pas autre chose.

▲ Hiérarchie d'héritage

```
System.Object
System.EventArgs
System.Windows.Forms.KeyPressEventArgs
```

Génération du code qui crée un **délégué** d'événement **new** un délégué **EventHandler** et l'ajoute (+=) au membre événement **Click** de l'annonceur **btnQuitter**.

(codé dans la méthode **InitializeComponent** dans **frmTest.Designer.cs**)

```
this.btnQuitter.Click += new System.EventHandler(this.btnQuitter_Click);
```

Il ne reste au développeur qu'à coder le traitement à exécuter lorsque l'utilisateur clique sur ce bouton Quitter.

Dans cet exemple, nous allons donc quitter la feuille.

Pour quitter la feuille, nous appellerons la méthode Close() de l'instance : **this.Close()**.

Comme cette feuille est celle sur laquelle démarre l'application, nous allons sortir de la boucle de messages Windows. L'application va donc s'arrêter.

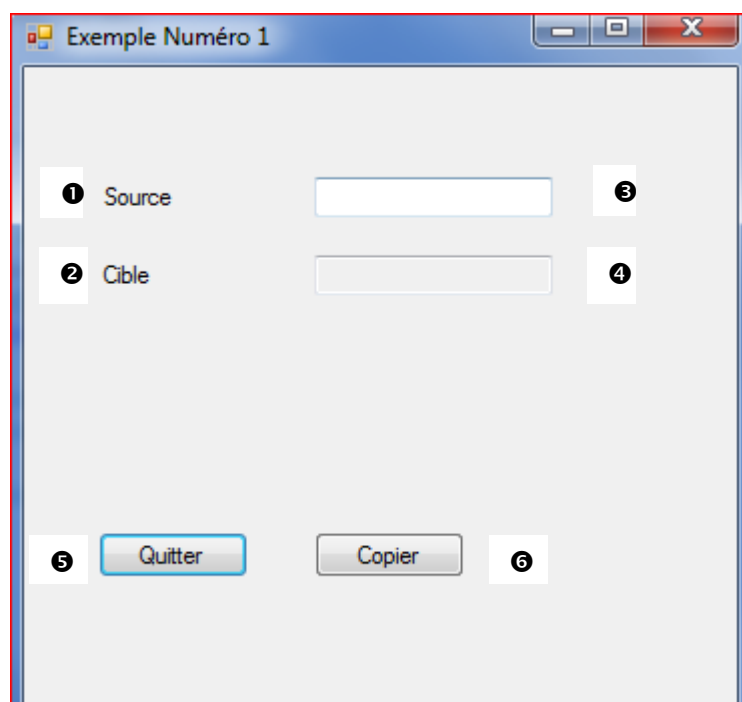
3.8 Exemple

Après avoir saisi du texte dans le contrôle source, en cliquant sur le bouton Copier, le texte saisi est ajouté au contrôle cible inaccessible en saisie.

Le contrôle source est alors automatiquement effacé.

Le focus se positionne sur le contrôle source.

Le bouton Quitter permet de fermer la feuille.



Les contrôles graphiques et leurs propriétés modifiées

- ❶ Un contrôle Label : Propriété **Text** a la valeur **Source**
- ❷ Un contrôle Label : Propriété **Text** a la valeur **Cible**
- ❸ Un contrôle TextBox : Propriété **Name** a la valeur **txtSource**
- ❹ Un contrôle TextBox : Propriété **Name** a la valeur **txtCible**
Propriété **ReadOnly** a la valeur **true**
- ❺ Un contrôle Button : Propriété **Name** a la valeur **btnQuitter**
Propriété **Text** a la valeur **Quitter**
- ❻ Un contrôle Button : Propriété **Name** a la valeur **btnCopier**
Propriété **Text** a la valeur **Copier**

Le code associé :

L'évènement **Click** sur le bouton **Copier** provoque l'exécution de la fonction qui copie le texte saisi dans la boîte de texte cible, efface le contenu de la boîte source et lui redonne le focus.

```
private void btnCopier_Click(object sender, EventArgs e)
{
    txtCible.Text += txtSource.Text + " " |;
    txtSource.Clear();
    txtSource.Focus();
}
```

L'évènement **Click** sur le bouton **Quitter** provoque la fermeture du formulaire courant.

```
private void button1_Click(object sender, EventArgs e)
{
    this.Close();
}
```

4 L'IDE Visual Studio

4.1 Commenter le code

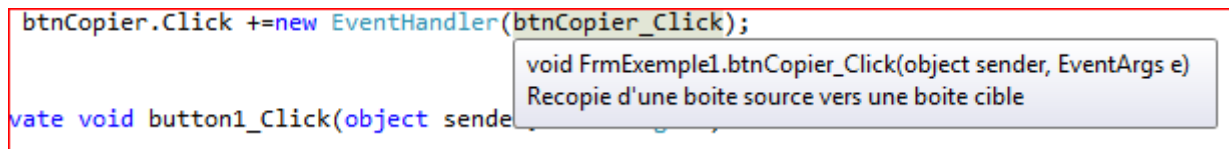
L'environnement de développement permet de fournir des commentaires repris par l'intellisense et pour la documentation, des commentaires XML en interprétant les caractères `///`. Des balises XML spécifiques permettent alors de préciser des commentaires particuliers que Visual Studio utilisera :

- `<summary>` : description de l'objet commenté
- `<param>` : description d'un paramètre
- `<returns>` : valeur de retour
- `<remarks>` : remarques particulières

A l'insertion des premiers caractères `///` Visual Studio insère automatique un bloc complet de commentaires standards, alimentés avec les paramètres déjà existants de la méthode. Le même résultat peut être obtenu en sélectionnant l'option Ajouter un commentaire dans le menu contextuel d'une méthode.

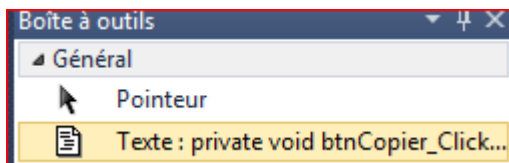
```
/// <summary>
/// Recopie d'une boîte
/// source vers une boîte cible
/// </summary>
/// <param name="sender">Boîte source</param>
/// <param name="e">Aucune info</param>
private void btnCopier_Click(object sender, EventArgs e)
{
    txtCible.Text += txtSource.Text + " " ;
    txtSource.Clear();
    txtSource.Focus();
}
```

L'Intellisense affichera cette aide lors du survol de la méthode.



4.2 Manipuler et mettre en forme le code

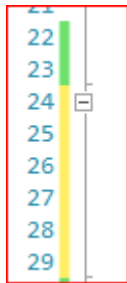
4.2.1 Copier / Coller du code



La boîte à outils offre la possibilité de stocker du code source afin de pouvoir le dupliquer autant de fois que désiré. Il suffit juste de glisser déposer du code sélectionné dans l'onglet Général de la boîte à outils.

L'opération inverse servira à insérer le bout de code dans l'éditeur.

4.2.2 Suivre les modifications de code



Visual Studio suit les modifications apportées dans la fenêtre de code.

Une barre verte à gauche mentionne les lignes modifiées et sauvegardées.

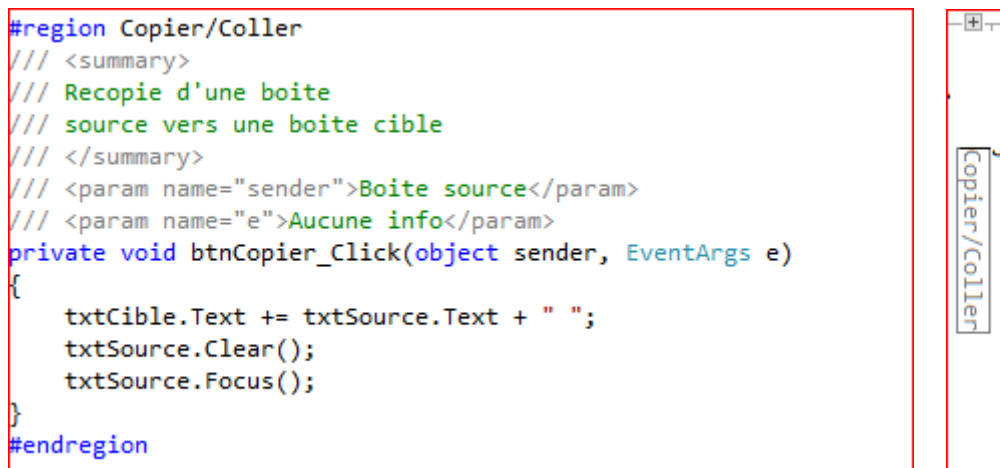
Une barre jaune les lignes modifiées non sauvegardées

La compilation sauvegarde les données.

4.2.3 Isoler des portions de code.

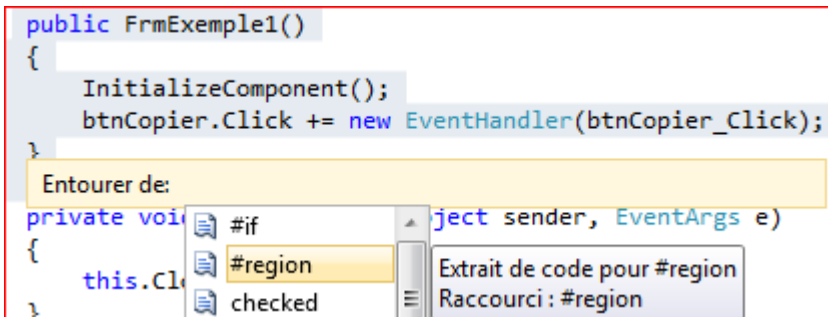
Des portions entières de code peuvent être regroupées en régions ; ce regroupement peut être très utile pour réduire le code affiché et donc faciliter la navigation puis la maintenance du code.

Une région est une zone de code qui possède un début, une description et une fin, le début étant marqué par #region, et la fin par #endregion.



L'intérêt de créer des régions est de pouvoir masquer une région entière pour la réduire à une seule ligne contenant la description de la région, en cliquant sur le signe – apparaissant à gauche de la fenêtre de code devant chaque début de région. Pour déplier une région, il faudra alors cliquer sur le signe +.

Un moyen encore plus simple de créer une région est d'utiliser le menu contextuel de l'éditeur de code : après avoir sélectionné le code correspondant à la région, choisir **Entourer de ...**, dans le menu contextuel.



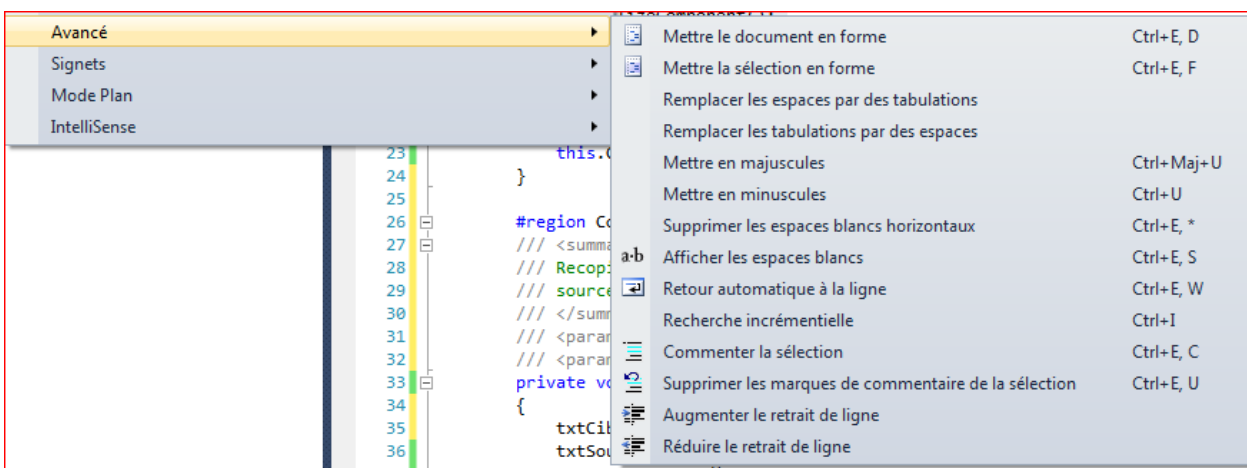
4.2.4 Mise en forme du code

Visual Studio permet d'appliquer facilement des modifications de formatage de texte à une sélection.

- Ajout ou suppression de tabulations
- Majuscules / Minuscules
- Suppression des espaces
- Commenter / Décommenter
-

Ces options sont accessibles par le menu **Edition / Avancé**.

L'option particulièrement intéressante de ce sous-menu est l'option **Mettre la sélection en forme**, qui permet, en fonction du paramétrage de l'éditeur et du langage utilisé, de mettre en forme un bloc de texte selon les normes.



4.3 Refactoriser

Le refactoring est une opération qui consiste à retravailler le code source, pour améliorer la lisibilité et simplifier la maintenance.

Visual Studio fournit un ensemble d'outils accessible, depuis la fenêtre de code ou ...

Depuis le menu **Refactoriser**.

- **Renommer** permet de renommer une classe, une méthode une variable en proposant certaines options : prévisualisation des modifications, étendre le changement de nom aux commentaires ...
- **Extraire une méthode** permet après sélection d'une portion de code, d'en constituer une méthode.
- **Encapsuler le champ** permet d'associer une propriété à ce dernier et ses accesseurs get/set.
- **Transformer la variable locale** en paramètre permet d'extraire et de passer en paramètre une variable déclarée localement dans une méthode.
- **Extraire l'interface** permet, à partir d'une méthode, d'en extraire la signature, de créer l'interface contenant la définition de l'interface et de l'implémenter dans la classe initiale.
- **Réorganiser les paramètres** permet après avoir sélectionné les paramètres d'une méthode, de pouvoir en redéfinir l'ordre en pré visualisant le code obtenu.

Depuis le menu **Edition / Intellisense**.

- **Générer un stub de méthode** permet après avoir utilisé un nom de méthode non codée d'en extraire une signature possible.
- **Insérer un extrait**

Les extraits de code (ou snippets) sont des portions de code personnalisables qui correspondent à des tâches souvent répétitives : ouvrir un fichier, écrire une boucle foreach, écrire un bloc Try ...Catch.

Visual Studio affiche la liste des extraits qu'il connaît : après insertion du code choisi, (les paramètres sont encadrés en couleur), le passage d'un paramètre à l'autre se fait par la touche Tab.

Il suffit de remplacer le paramètre par sa valeur et valider par Entrer

4.4 L'environnement graphique

On remarquera les possibilités de l'environnement graphique de Visual Studio.

4.4.1 La liste des tâches (*smarttag*)



Le clic sur la petite flèche donne des raccourcis rapides et simples vers des fonctions essentielles du contrôle, comme l'affectation de la valeur d'une propriété ou le lancement d'un assistant ou d'un éditeur personnalisé

4.4.2 Les barres d'alignement (*snaplines*)

Permettent d'aligner horizontalement ou verticalement deux contrôles graphiques.

Une ligne bleue indique que le contrôle est correctement aligné, une ligne rouge indique que le texte est aligné, une ligne pointillée indique que l'espacement est correct par rapport aux autres contrôles ou aux bords de la feuille ;

Utilisées avec la touche *Ctrl*, les touches directionnelles du clavier permettent de poser le contrôle sur la prochaine *snapline*.

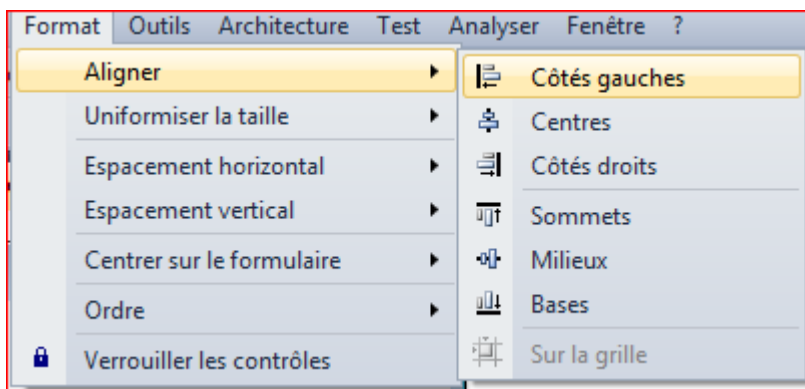
4.4.3 Donner la même propriété à plusieurs composants

Il est possible, en une seule opération, d'affecter des propriétés identiques à plusieurs contrôles :

Sélectionnez l'ensemble des contrôles (par la touche MAJ ou CTRL), la fenêtre des propriétés affiche alors uniquement les propriétés communes qu'il est possible de modifier.

4.4.4 Placement des contrôles les uns par rapport aux autres

Après avoir sélectionné les contrôles à traiter :



Choisissez le menu **Format / Aligner**, on alignera l'ensemble des contrôles sur le dernier composant sélectionné.

Choisissez le menu **Format / Uniformiser la taille**, on donnera une taille identique au dernier contrôle sélectionné.

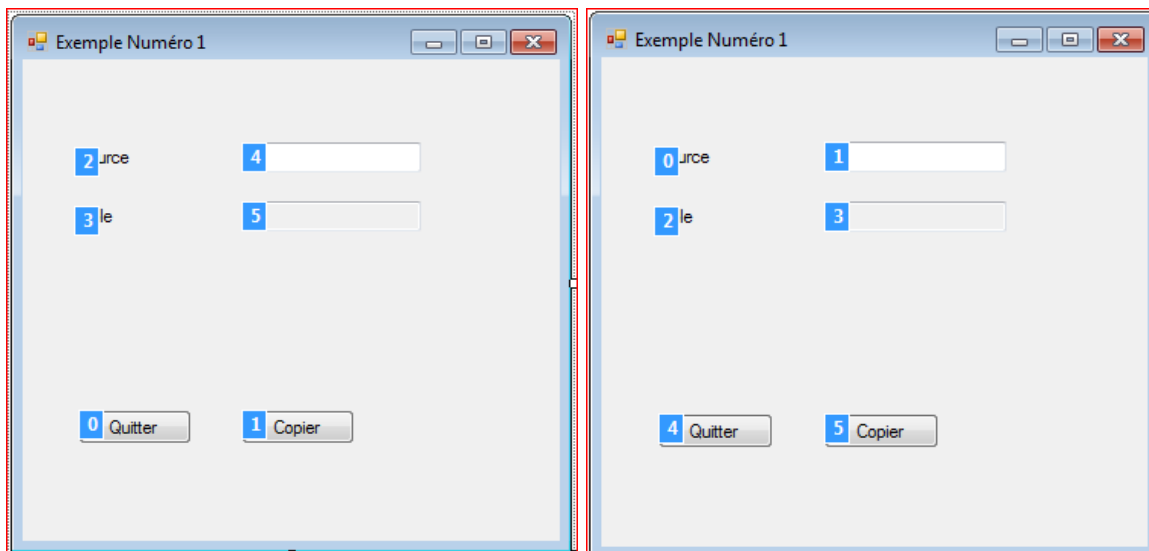
Vous pouvez aussi uniformiser les espacements entre les contrôles ou répartir ceux-ci sur la formulaire.

4.4.5 Positionner automatiquement le passage du focus

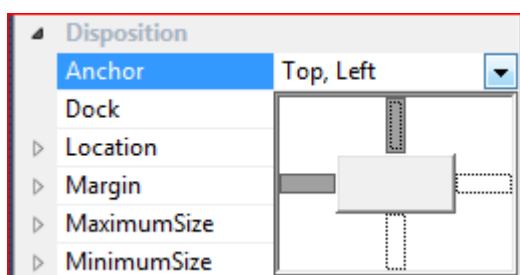
Vous devez absolument programmer correctement l'ordre de passage du focus en fonction de la tabulation.

Choisissez **Affichage / Ordre de tabulation**, l'ordre de tabulation des contrôles de la fenêtre ayant leur propriété **TabStop** à **true** est affiché. Pour modifier l'ordre de passage du focus, cliquez sur chaque composant dans l'ordre désiré et terminer par la touche Echap.

L'exemple ci-dessous illustre à gauche une fenêtre avec un ordre de tabulation non conforme, à droite la version corrigée.



4.4.6 Ancrage des composants par rapport à la fenêtre mère

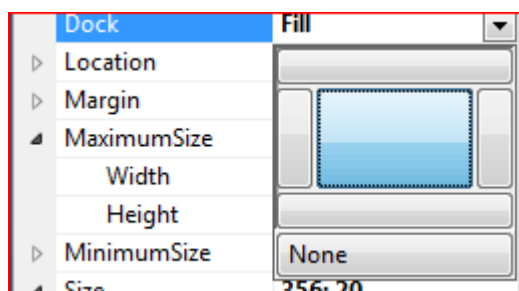


La propriété **Anchor** permet de redimensionner automatiquement la taille d'un composant, mais aussi de le repositionner automatiquement quand la fenêtre mère change de taille.

Par défaut, le composant s'ancre par rapport au bord supérieur et au bord gauche.

Cette propriété est positionnée graphiquement.

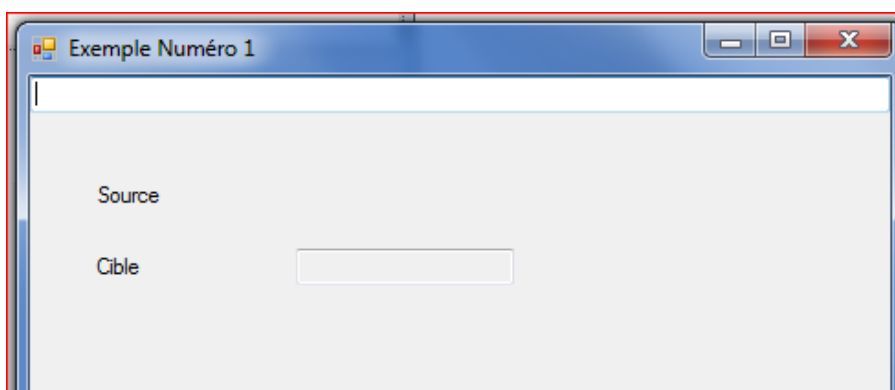
4.4.7 Accoler un contrôle à un bord de fenêtre



La propriété **Dock** permet de fixer la position d'un contrôle par rapport à un bord de sa fenêtre mère : il peut alors être forcé à rester coller contre un bord (Left, top, Bottom, Right), ou s'étendre automatiquement de manière à occuper toute la largeur ou toute la hauteur de sa fenêtre mère (Fill).

Cette propriété est positionnée graphiquement.

Dans l'exemple suivant la boîte de texte utilise toute la largeur de la feuille. Sans intérêt ici.



5 Les fenêtres

Une fenêtre n'est pas seulement une zone rectangulaire de l'écran dans laquelle une application effectue des affichages : Une fenêtre est en effet définie par :

- Des attributs, ou caractéristiques, ou propriétés. Par exemple, son icône affichée dans la barre des tâches ou son titre.
- Des méthodes pour agir directement sur la fenêtre, par exemple la fermer.
- Des méthodes pour traiter des événements signalés par Windows comme le chargement, le premier affichage ou la demande de fermeture.

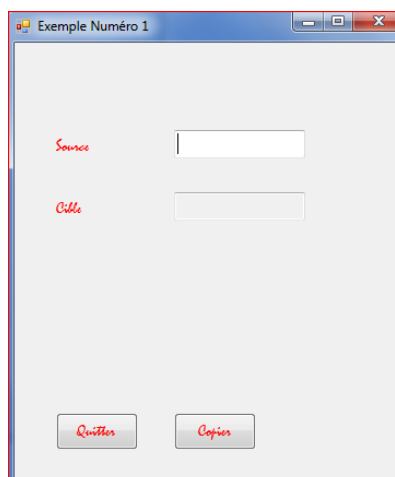
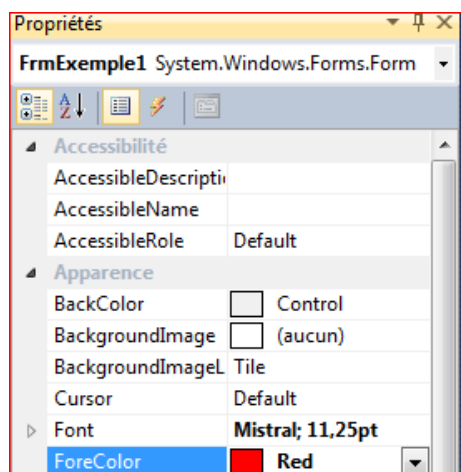
5.1 Les principales propriétés d'une fenêtre

Le tableau suivant décrit les propriétés les plus courantes, qui peuvent être modifiées au moment du design.

Il existe aussi des propriétés qui n'apparaissent pas dans la fenêtre **Propriétés** et qui ne peuvent être utilisées par programme qu'au moment de l'exécution. La propriété **ActiveControl** du formulaire et qui indique le contrôle actif en fait partie car les contrôles doivent au préalable avoir été initialisés.

Propriétés	Description
Name	Nom du formulaire. Deux formulaires d'un même projet ne peuvent pas avoir le même nom.
BackColor	Couleur d'arrière-plan par défaut des textes et graphiques d'un formulaire
BackgroundImage	Bitmap, icône ou autre fichier graphique à utiliser comme image de fond du formulaire. Si l'image est plus petite que le formulaire, elle s'affiche en mosaïque pour remplir la totalité du formulaire
ControlBox	Indique si le menu système est affiché ou non
Font	Police par défaut utilisée par les contrôles incorporés au formulaire et affichant du texte
ForeColor	Couleur de premier plan par défaut des textes et images du formulaire

A noter : Il ne faut pas redéfinir les couleurs et les polices de chaque contrôle. Si aucune propriété spécifique n'est définie pour un contrôle, il hérite des propriétés du conteneur. Ainsi, si vous souhaitez modifier la police par défaut pour l'ensemble des contrôles du formulaire, modifiez la propriété de l'objet formulaire. Le résultat en image.



Propriétés	Description
FormBorderStyle	Contrôle l'aspect et le type de la bordure du formulaire. (par défaut : <i>Sizable</i>). D'autres options spécifient que les bordures ne sont pas redimensionnables ou qu'elles ne comportent pas les boutons du menu Système
Icon	Indique l'icône apparaissant dans le menu Système du formulaire et sur la barre des tâches Windows
Location	Précise les coordonnées du coin supérieur gauche du formulaire par rapport à son conteneur (l'écran lui-même ou un autre formulaire)
MaximizeBox	Indique si la commande Agrandir du menu Système et la barre de légende sont activées ou non. (activée par défaut)
MaximumSize	Indique la taille maximale du formulaire. La valeur par défaut (0, 0) signifie qu'il n'existe pas de taille maximale et que l'utilisateur peut le redimensionner à sa convenance
Menu	Indique le menu apparaissant dans la barre de menus du formulaire. (par défaut -aucun- indique que le formulaire n'a pas de menu)
MinimizeBox	Indique si la commande Réduire du menu Système et la barre de titre sont activées ou désactivées (activée par défaut)
MinimumSize	Indique la taille minimale du formulaire
StartPosition	Définit la position initiale du formulaire, par exemple <i>CenterScreen</i>
ShowInTaskBar	Détermine si le formulaire s'affiche dans la barre des tâches
Size	Taille par défaut du formulaire quand il est affiché pour la première fois.
Text	Contient le texte figurant sur la barre de titre du formulaire.
WindowState	Définit l'état initial du formulaire lorsqu'il est affiché pour la première fois. L'option par défaut (<i>Normal</i>) positionne le formulaire conformément aux propriétés <i>Location</i> Autres options : <i>Minimized</i> et <i>Maximized</i>

Consultez l'aide de la classe Windows Form pour plus d'infos.

5.2 Les principaux événements d'une fenêtre

Les événements liés à un composant sont repris dans la partie Événements de la fenêtre des propriétés relative à ce composant.

Pour traiter un événement, il faut compléter la méthode qui lui est liée en « double-cliquant » sur le nom de l'événement dans la fenêtre des propriétés, de la même façon que nous avons traité le click du bouton, dans le chapitre précédent.

Le cycle de vie d'une fenêtre passe par de nombreux événements :

Lorsque la méthode Show () est appelée, des événements se produisent généralement dans l'ordre suivant :

Evènements	Description
Load	Se produit chaque fois qu'une feuille est chargée en mémoire. Utilisé pour effectuer des traitements avant que l'affichage ne se produise.
Activated	La fenêtre devient active (au démarrage) ou le redevient (après une réduction en icône ou au changement de fenêtre)..
Deactivate	La fenêtre a perdu son état de fenêtre active (l'utilisateur réduit la fenêtre en icône ou a terminé l'exécution)
FormClosing	L'utilisateur a marqué son intention de fermer la fenêtre et le programme peut encore refuser cette fermeture (c'est l'occasion de demander confirmation de l'opération). Le second argument de la fonction de traitement est de type FormClosingEventArgs.(e.Cancel= true annulera la fermeture de la fenêtre)
FormClosed	La fenêtre a été fermée par l'utilisateur (par ALT+F4, ou un clic sur la case de fermeture ou par arrêt de Windows).

6 Les différentes formes de dialogue

Windows propose une interface graphique multifenêtres. C'est à dire que l'écran est fractionné en plusieurs parties où s'exécutent les applications. Ces fenêtres peuvent se chevaucher, se recouvrir mutuellement ou être disposées côte à côte permettant ainsi de partager l'espace de travail.

Dans ce contexte, vous devez différencier différents modes de fonctionnement, qui précisent si l'application peut ouvrir un ou plusieurs documents et si l'utilisateur est libre de passer d'un document à un autre à sa guise où s'il doit clore le document pour passer à un autre.

Ces différents modes de fonctionnement nous permettent de distinguer :

- Les applications qui fonctionnent en mode SDI
- Les applications qui fonctionnent en mode MDI
- Les dialogues de type modal ou non

6.1 Application en mode SDI

Les applications qui fonctionnent en mode SDI (Single Document Interface) n'autorisent le chargement d'un seul document à la fois. Pour travailler sur un nouveau document, l'application doit au préalable fermer celui actuellement ouvert.

Le principe des applications SDI est exploité dans le Bloc-notes de Windows.

Le mode SDI est à proscrire.

6.2 Application en mode MDI

Le MDI (Multiple Document Interface) autorise l'ouverture de plusieurs documents à l'intérieur d'une même application. Il définit le comportement des fenêtres documents à l'intérieur d'une fenêtre mère ou fenêtre de l'application.

Les meilleurs exemples de l'utilisation de fenêtres MDI se trouvent dans les outils bureautiques de la suite Office.

L'intérêt du principe MDI est suffisamment manifeste pour l'utiliser également dans des applications de gestion. Il se justifie alors dans l'ouverture simultanée de plusieurs vues du système d'information de l'entreprise.

Pour le développeur, la gestion des fenêtres devient dès lors plus complexe.

Il lui faudra notamment :

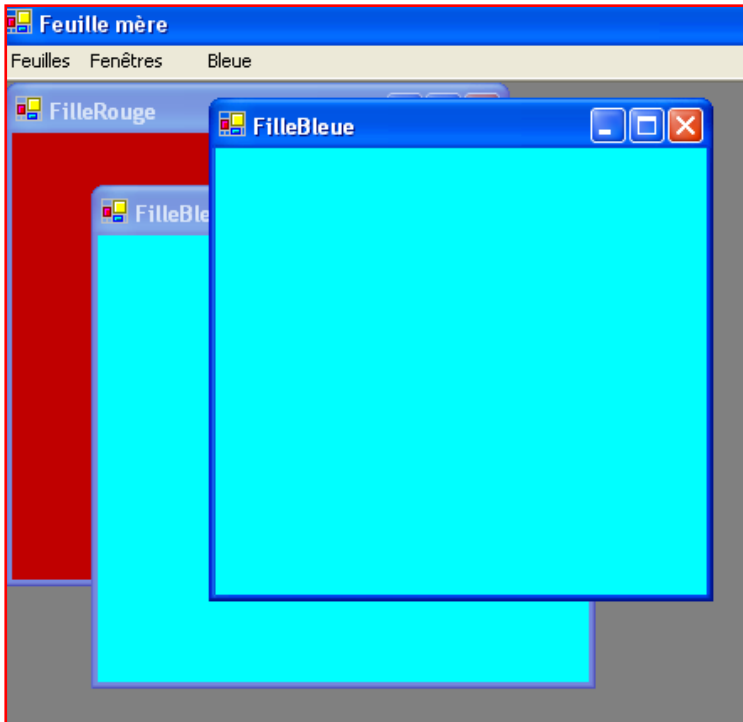
- Gérer les éventuels conflits d'accès qui peuvent survenir lorsque deux formulaires souhaitent modifier les mêmes informations de manière concurrente
- Eviter à l'utilisateur de se perdre dans le dédale de fenêtres ouvertes (dédales : du grec Δαίδαλος / Daídalos qui tire son origine de l'architecte concepteur du labyrinthe destiné à enfermer le minotaure).

La mise en place de dialogues modaux peut être une approche avec pour objectif de prévenir les conflits d'accès concurrentiels et la perte de contrôle de l'utilisateur.

Les applications MDI sont composées d'un formulaire principal appelée formulaire MDI. Il ne peut exister qu'un formulaire MDI par application. Le formulaire MDI agit comme un conteneur

de formulaires enfants qui seront contenus dans son contexte. Les feuilles filles d'une feuille MDI sont, comme la feuille MDI, non modales.

La fermeture du formulaire MDI a pour incidence la fermeture de l'ensemble de ses fenêtres filles.



Les fenêtres MDI présentent les principales caractéristiques :

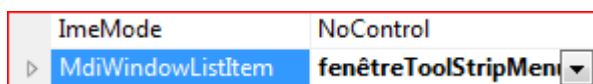
- Toutes les fenêtres enfant sont affichées à l'intérieur de la fenêtre parent et ne peuvent être déplacées qu'à l'intérieur de cette fenêtre.
- Les fenêtres enfant peuvent être réduites en icône, mais les icônes sont affichées dans la fenêtre parent et pas dans la barre des tâches.
- Si une fenêtre enfant est maximisée, son titre est ajouté au titre de la fenêtre parent
- Le menu de la fenêtre enfant active s'insère dans celui de la fenêtre parent.

6.2.1 Créer une feuille mère

Dans la fenêtre des propriétés de la feuille, positionner la propriété **IsMdiContainer** à **True**. Il est conseillé de positionner également la propriété **WindowState** à **Maximized**, ce qui permettra de manipuler les feuilles enfant plus facilement.

Le conteneur MDI embarque un menu, un composant **MenuStrip**, afin de permettre l'instanciation des fenêtres filles par sélection dans le menu et de disposer de mécanismes d'arrangement des fenêtres enfant dans le conteneur.

Positionner dans la propriété **MdiWindowListItem** du composant **MenuStrip** de la feuille mère l'élément du menu dans lequel Windows se chargera de gérer la liste des fenêtres actives, ici le menu Fenêtres.



Vous pouvez choisir cet élément dans la liste proposée en mode Design.

6.2.2 Créer une feuille enfant

Ajouter un élément de menu puis cliquer sur cet élément pour ouvrir le gestionnaire d'événement associé. Vous allez pouvoir maintenant créer une instance de la fenêtre fille et l'associer au conteneur parent par la propriété MDIParent.

```
private void filleToolStripMenuItem_Click(object sender, EventArgs e)
{
    FrmExemple1 frmExemple1 = new FrmExemple1();
    frmExemple1.MdiParent = this;
    frmExemple1.Show();
}
```


6.2.3 Arranger la disposition des fenêtres filles

Vous pouvez créer des options de menus permettant de réarranger l'espace du conteneur et des formulaires enfant qu'il contient.

Pour arranger la disposition des feuilles utilisez la méthode `LayoutMdi` avec l'une des quatre valeurs de l'énumération `MdiLayout`.

```
private void cascadeToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.Cascade);
}

private void répartiesToolStripMenuItem_Click(object sender, EventArgs e)
{
    this.LayoutMdi(MdiLayout.ArrangeIcons);
}
```

Valeurs de `MdiLayout` :

- **ArrangeIcons** :
Toutes les icônes enfants MDI sont disposées dans la zone cliente du formulaire parent MDI.
- **Cascade** :
Toutes les fenêtres enfants MDI sont disposées en cascade dans la zone cliente du formulaire parent MDI.
- **TileHorizontal**
Toutes les fenêtres enfants MDI sont disposées en mosaïque horizontalement dans la zone cliente du formulaire parent
- **MDI.TileVertical**
Toutes les fenêtres enfants MDI sont disposées en mosaïque verticalement dans la zone cliente du formulaire parent MDI.

6.2.4 Fusion du menu des fenêtres filles

Les propriétés **AllowMerge** des composants **MainStrip** de la feuille mère et de la feuille fille doivent être positionnées à `true` (valeur par défaut).

Le menu de la fenêtre enfant active **se fond** dans celui de la feuille mère, suivant la valeur de la propriété **MergeAction** des objets **ToolStripMenuItem**.

Le type de fusion d'un élément de menu indique la façon dont l'élément se comporte quand il a le même ordre de fusion qu'un autre élément de menu fusionné.

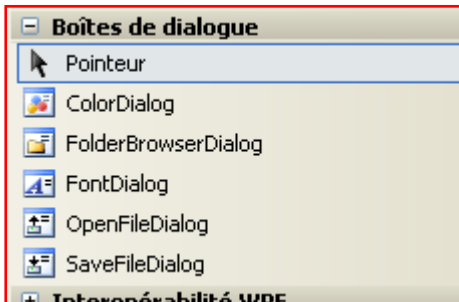
Vous pouvez fusionner des menus pour créer un menu consolidé basé sur deux ou plusieurs menus existants.

La propriété **MergeIndex** définit la position d'un élément fusionné.

La propriété **MergeAction** indique ce qui doit se passer lors de la fusion. Voir documentation sur les menus.

6.3 Dialogue Modal ou Non

La modalité est une des notions primordiales de la conception d'une interface graphique. Elle décrit un mode de fonctionnement particulier où l'application reprend le contrôle. Tant que l'application n'a pas reçu de réponse de l'utilisateur toutes les autres actions au sein de l'application sont suspendues. Par contre l'utilisateur peut accéder aux autres applications de Windows.



Un dialogue modal est mis en œuvre via une fenêtre particulière désignée sous le terme de boîte de dialogue.

Plusieurs boîtes de dialogue standard sont livrées avec l'EDI : Couleurs, Polices, Ouverture ou Enregistrement d'un fichier, ...que vous trouvez regroupées dans la boîte à outils.

Pour simplifier, l'utilisation d'une fenêtre modale est nécessaire dès lors que l'utilisateur doit terminer une action avant d'en commencer une autre.

Il est souhaitable d'éviter les situations modales au sein d'une interface graphique. Ce mode de fonctionnement allant à l'encontre même du principe de l'environnement graphique qui place l'utilisateur en position de contrôle de l'application.

Malgré tout, dans certains cas, l'utilisation de ce contexte est inévitable :

- Les dialogues de continuation qui demandent à l'utilisateur de fournir une information afin de continuer l'action en cours. Le principe de l'assistant d'installation par exemple.
- Les dialogues transactionnels, très fréquemment utilisés en gestion, qui attendent de la part de l'utilisateur la saisie d'informations suivi d'une validation ou d'une annulation des modifications.

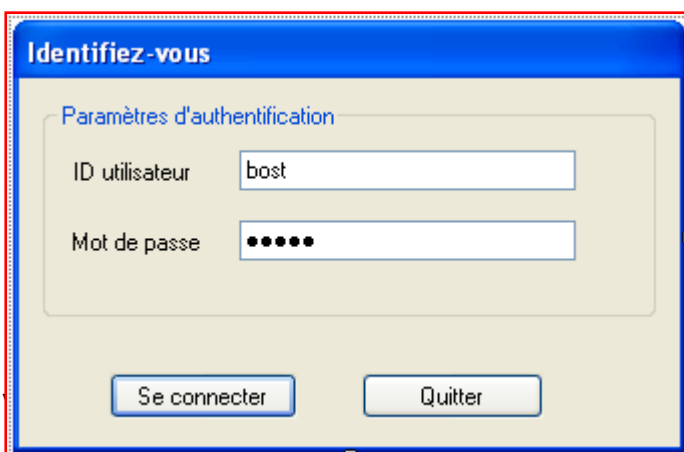
Les formulaires seront donc affichés selon deux modes :

- Le mode modal : L'utilisateur devra quitter le formulaire avant de pouvoir entreprendre une autre action dans l'application.
- Le mode non modal : L'utilisateur pourra activer un autre formulaire sans fermer le formulaire en cours.

Quel que soit le mode retenu, il n'existe, au sein d'une application, qu'un seul formulaire actif. Ce formulaire détient le focus.

6.3.1 Dialogue Modal

Exemple de dialogue modal personnalisé :



Une boîte de dialogue est une fenêtre qui se doit d'obéir à certaines règles :

- Elle n'est pas redimensionnable
- Elle ne peut être mise en icône.
- Elle est indépendante de l'espace de travail et ne peut être une fenêtre fille d'une fenêtre conteneur MDI

Une fenêtre est affichée en mode modal avec la méthode en invoquant la méthode **ShowDialog()** de l'objet formulaire.

Il est nécessaire de tester au point de retour de la méthode la réponse de l'utilisateur au travers de la propriété **DialogResult** qui pourra prendre l'une des valeurs de l'énumération **System.Windows.Forms.DialogResult**.

Illustration ci-dessous dans le cadre de la fenêtre de connexion.

```
FrmConnexion oConnexion = new FrmConnexion();  
DialogResult oResult = oConnexion.ShowDialog();  
  
enum System.Windows.Forms.DialogResult  
Specifies identifiers to indicate the return value of a dialog box.
```

6.3.2 Dialogue non modal

Un formulaire est affiché dans un mode non modal en invoquant la méthode **Show()** de l'objet formulaire.

Il pourra être :

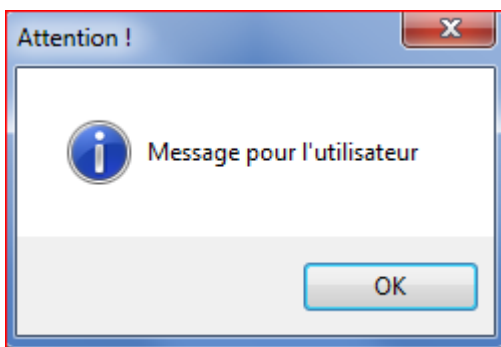
- Fermer avec la méthode **Close()**
- Minimiser dans la barre d'état
- Cacher avec la méthode **Hide()** masqué

6.4 La classe MessageBox

Elle permet d'afficher une boîte de dialogue à l'utilisateur dont la réponse peut être ou non ignorée. L'affichage de la boîte de dialogue est modal : l'utilisateur doit terminer ce dialogue avant d'envisager une autre action.

Si la réponse de l'utilisateur est sans intérêt, comme dans le cas suivant, nous ne traitons par le résultat du dialogue.

```
MessageBox.Show("Message pour l'utilisateur", "Attention !",  
MessageBoxButtons.OK,  
MessageBoxIcon.Asterisk);
```



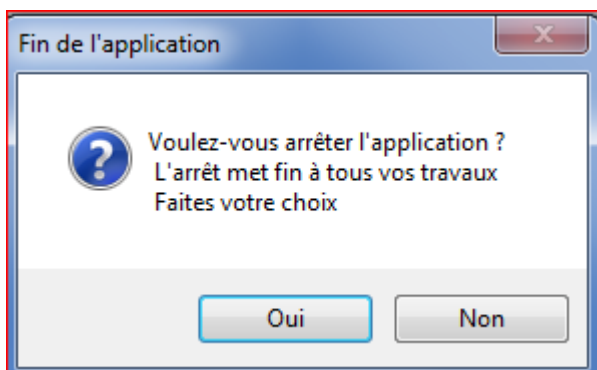
L'affichage de la boîte de dialogue est modal : l'utilisateur doit terminer ce dialogue avant d'envisager une autre action.

Dans l'exemple suivant, nous exigeons une réponse de l'utilisateur.

Comme dans le cadre d'un dialogue modal personnalisé, l'utilisateur doit sortir par l'un des boutons affichés et ne peut utiliser la case fermeture.

```
string texteDialogue=string.Format("Voulez-vous arrêter l'application ? \n "
+"|L'arrêt met fin à tous vos travaux \n Faites votre choix");
string titreDialogue = "Fin de l'application";

DialogResult dr = MessageBox.Show(texteDialogue, titreDialogue,
MessageBoxButtons.YesNo,
MessageBoxIcon.Question,
MessageBoxDefaultButton.Button1);
```



MessageBoxButtons peut prendre l'une des valeurs suivantes de l'énumération :

- AbortRetryIgnore Les boutons Abandonner, réessayer et Ignorer sont affichés
- OK Seul le bouton OK est affiché
- OKCancel Les boutons OK et Annuler sont affichés
- RetryCancel Les boutons Réessayer et Annuler sont affichés
- YesNo Les boutons Oui et Non sont affichés
- YesNoCancel Les boutons Oui, Non et Annuler sont affichés

MessageBoxIcon peut être l'une des valeurs de l'énumération suivantes :

Asterisk, Error, Exclamation, Hand, Information, None, Question, Stop, Warning

MessageBoxDefaultButton peut prendre la valeur Button1, Button2 ou Button3 indiquant lequel des boutons est le bouton par défaut (son contour étant souligné en gras)

La valeur renvoyée par Show() indique le bouton utilisé pour quitter la boîte de message, soit :

- Abort Bouton Abandonner
- Cancel Bouton Annuler ou touche ECHAP
- Ignore Bouton Ignorer
- No Bouton Non
- OK Bouton OK
- RetryBouton Essayer de nouveau
- Yes Bouton Oui

7 Généralités sur les contrôles

Les contrôles sont des objets contenus dans des composants graphiques qui sont disposés sur un formulaire via un conteneur. Le formulaire est un conteneur qui peut lui-même contenir d'autres conteneurs.

Chaque contrôle présent sur le formulaire doit être instanciés à partir d'un Type, personnalisé, positionné et enfin ajouté au conteneur.

Chaque type de contrôle possède son propre ensemble de propriétés, de méthodes et d'événements définis dans une classe System.Windows.Forms.

Ils sont **tous dérivés** de la classe de base **Control**.

▲ Hiérarchie d'héritage

```
System.Object
  System.MarshalByRefObject
    System.ComponentModel.Component
      System.Windows.Forms.Control
        System.Windows.Forms.ScrollableControl
          System.Windows.Forms.ToolStrip
            System.Windows.Forms.BindingNavigator
            System.Windows.Forms.MenuStrip
            System.Windows.Forms.StatusStrip
            System.Windows.Forms.ToolStripDropDown
```

Ici, affichage de la hiérarchie d'héritage du contrôle ToolStrip.

Les contrôles peuvent être insérés sur la feuille lors de sa conception, depuis la boîte à outils, comme vu dans les chapitres précédents.

Vous pouvez observer dans le code généré par le Designer les différentes opérations effectuées. Dans l'exemple qui suit, il s'agit de l'ajout d'un contrôle de type GroupBox qui sera amené à contenir d'autres contrôles.

```
this.gBBoutons = new System.Windows.Forms.GroupBox();
...
this.gBBoutons.Location = new System.Drawing.Point(90, 107);
this.gBBoutons.Name = "gBBoutons";
this.gBBoutons.Size = new System.Drawing.Size(470, 86);
this.gBBoutons.TabIndex = 2;
this.gBBoutons.TabStop = false;
this.gBBoutons.Text = "Mot ";
//
// FrmAjouterControles
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(613, 315);
this.Controls.Add(this.gBBoutons);
```

Ils peuvent également être créés dynamiquement en cours d'exécution du programme.

7.1 Création dynamique de contrôles

Nous pouvons par exemple créer un ensemble de contrôle dans un cycle.

Ici, je souhaite afficher un bouton par lettre qui compose un mot.

Le mot étant communiqué par l'utilisateur, ce mécanisme ne peut être réalisé que lors de l'exécution du programme et non en conception.

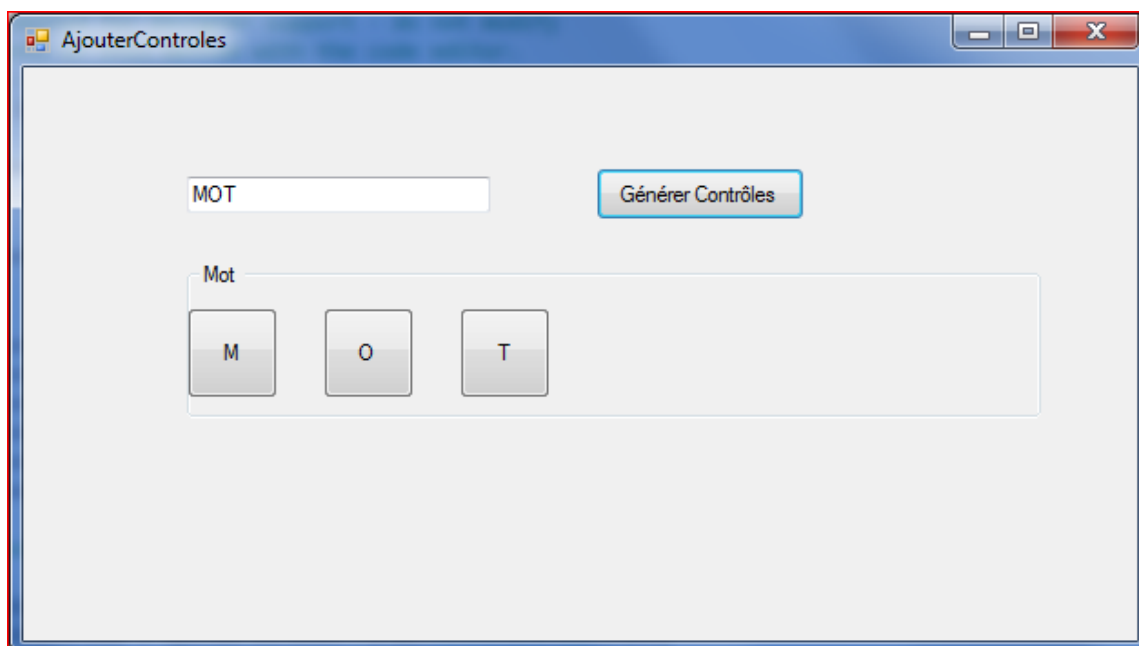
Pour simplifier la manipulation de ces boutons et leur positionnement, je les ai ajoutés dans un conteneur de type GroupBox.

Ce **conteneur**, tout comme tout autre conteneur, dispose d'une propriété **Controls** qui représente une collection de contrôles.

Il en est de même du formulaire : Vous pouvez observer dans le code généré par le Designer l'ajout du contrôle gBBoutons à This.Controls.

```
|
gBBoutons.Controls.Clear();

for (int i = 0; i < txtMot.Text.Length; i++)
{
    Button bouton = new Button();
    bouton.Size = new System.Drawing.Size(50,50);
    bouton.Left = i*(50 + (bouton.Width/2));
    bouton.Top = 25;
    bouton.Text = txtMot.Text[i].ToString();
    this.gBBoutons.Controls.Add(bouton);
}
```



A noter : la variable bouton ne fait référence qu'au dernier bouton ajouté à la collection. Pour accéder aux éléments de la liste, il faut itérer sur la collection Controls par le biais d'un cycle for ou foreach.

7.2 Gestion des événements

Chaque contrôle expose des événements génériques issus de la classe Control (clavier, souris, validation..) et aussi des événements propres à chaque classe de contrôle.

Le gestionnaire pour cet événement peut être créé directement en double-cliquant sur le contrôle dans la fenêtre des propriétés (affichage événement). Il ne reste ensuite qu'à coder la réaction de l'applicatif à l'événement ainsi généré.

Tous les contrôles lèvent des événements suite à une action de l'utilisateur qui peuvent être ou non gérés, i.e. associés par le biais d'un délégué d'événement à une procédure événementielle dite Gestionnaire d'Événement.

Nous avons introduit ce principe au chapitre 3. Je n'y reviens donc pas. Mais nous pouvons aussi les associer par programme.

7.2.1 Associer un gestionnaire par programme

Je vais amender le cas précédent pour que lorsque l'utilisateur clique sur un bouton, une boîte de message affiche le texte du bouton, change sa couleur de fond et le désactive pour qu'il ne soit plus sélectionnable.

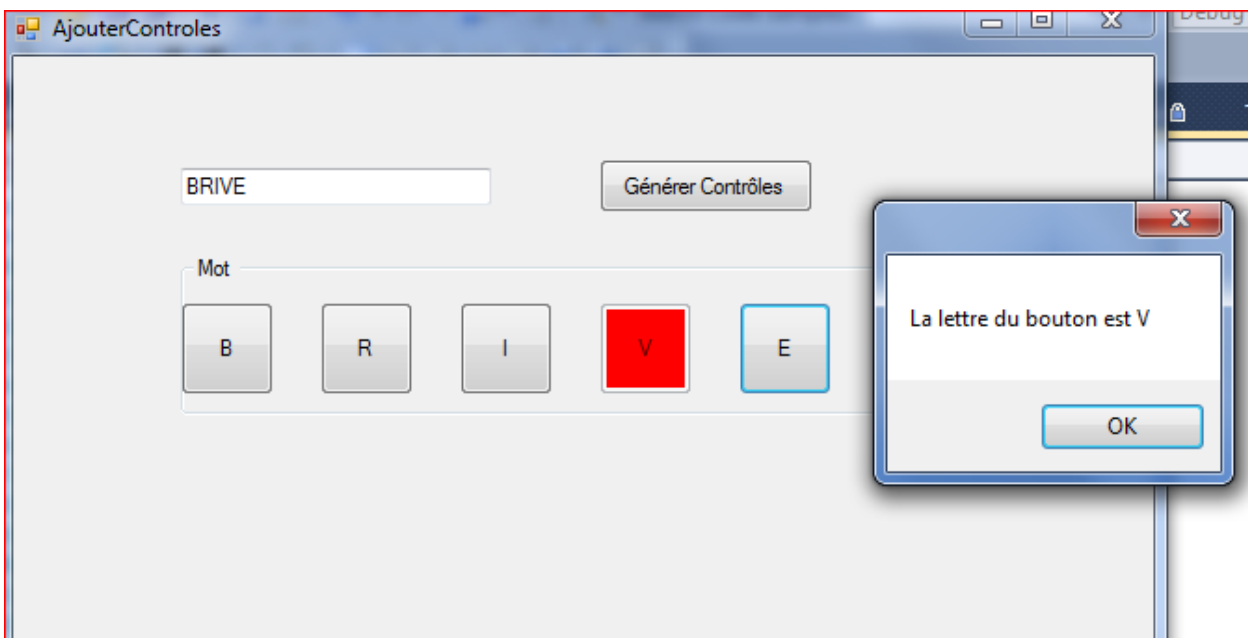
Au sein de la boucle for précédente, association du délégué au gestionnaire :

```
bouton.Click += new EventHandler(bouton_Click);
```

Définition du gestionnaire :

```
void bouton_Click(object sender, EventArgs e)
{
    Button boutonSelectionne = sender as Button;
    boutonSelectionne.Enabled = false;
    boutonSelectionne.BackColor = Color.Red;
    MessageBox.Show(string.Format("La lettre du bouton est {0}", boutonSelectionne.Text));
}
```

Résultat en image :



7.3 Gérer le focus

Pour donner le focus à un contrôle spécifique, il convient d'employer la méthode **Select** plutôt que la méthode **Focus** exposée par la classe **Control**.

En effet, la méthode **Focus()** semble ne pas toujours donner le résultat escompté...

Ainsi, pour que la boîte de texte txtMot de l'exemple précédent devienne le contrôle courant et que le curseur soit positionné dans la boîte de texte.

```
txtMot.Select();
```

Un contrôle ne peut obtenir le focus que lorsque le formulaire qui le contient est actif ;

Il est nécessaire pas ailleurs qu'il puisse être sélectionnable (enabled). Dans l'exemple précédent, nous ne pouvons plus sélectionner un bouton qui serait enabled false.

Les contrôles Windows Forms dans la liste suivante ne peuvent pas être sélectionnés pas plus que les contrôles qui en sont dérivés.

- **Panel**
- **GroupBox**
- **PictureBox**
- **ProgressBar**
- **Splitter**
- **Label**

8 Communiquer des informations entre feuilles

Lorsque vous avez besoin de fournir ou extraire des informations pour les passer d'un formulaire à une autre, vous devez respecter les canons de notre art !

Et vous rappelez qu'un formulaire est une instance d'un Type : les règles de la programmation objet doit s'appliquer la aussi !

Appliquez les règles suivantes pour respecter les normes de programmation :

Si l'objet qui crée le formulaire a besoin de communiquer des informations en entrée à celui-ci, il les lui communique :

- En recourant à un constructeur prenant en argument les informations nécessaires.
- En implémentant une propriété définie avec une visibilité publique ou interne au niveau du formulaire. Il fournira alors les informations par le biais de la propriété et de son accesseur **set**.

L'une ou l'autre de ces techniques permettent à la feuille de contrôler la qualité des données qu'elle reçoit.

De même si vous devez obtenir une information d'une feuille, passez par le biais d'un accesseur **get**.

Vous n'utiliserez pas d'autres méthodes.

8.1 Surcharge du constructeur

Pour exemple ce dialogue très courant mettant en œuvre une fenêtre de type Liste et une boîte de dialogue personnalisée.

La fenêtre liste contient une liste des éléments et le cas échéant des critères de filtre et de recherche.

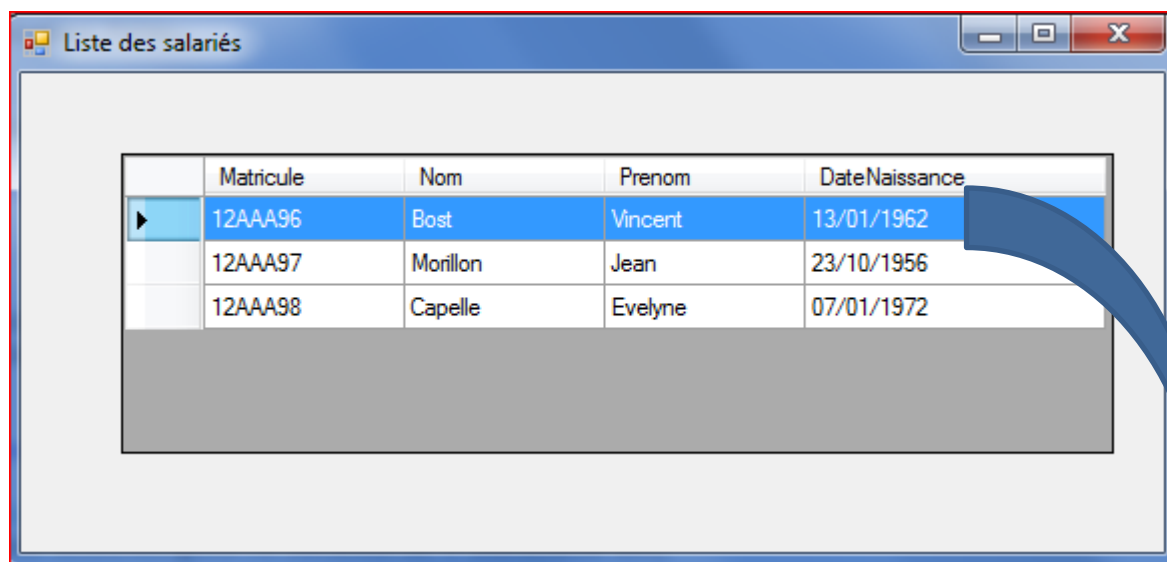
Elle permet de sélectionner un élément pour modification.

Elle permet aussi de demander l'ajout d'un nouvel élément.

Elle permet éventuellement de demander la suppression d'un élément existant.

Certaines actions nécessitent le recours à un dialogue modal qui affiche un formulaire détail qui permettra d'introduire les données ou de modifier celles-ci.

La fenêtre liste va créer l'instance de la fenêtre détail en lui communiquant les références de l'objet à modifier.



Nom du salarié :

Prénom du salarié :

Date naissance :

La référence de l'objet de type Salarie est passée via le constructeur d'initialisation.

Ne vous souciez pas pour le moment de l'objet `SalariesBindingSource` qui me permet de récupérer l'instance du Salarie sélectionné.

Une instance du formulaire de détail est créée par appel du constructeur surchargé qui attend en paramètre la référence d'un objet de type `Salarie`.

Dans le formulaire de gestion de la liste :

```
private void dataGridView1_DoubleClick(object sender, EventArgs e)
{
    FrmDetailSalarie frmDetailSalarie = new FrmDetailSalarie(salariesBindingSource.Current as Salarie);
    DialogResult dg = frmDetailSalarie.ShowDialog();
}
```

Dans le formulaire détail :

```
public partial class FrmDetailSalarie : Form
{
    Salarie _salarie = null;

    public FrmDetailSalarie()
    {
        InitializeComponent();
    }
    |
    public FrmDetailSalarie(Salarie salarie):this()
    {
        this._salarie = salarie;
        ChargerDetail();
    }
}
```

Nous sommes de nouveau confrontés aux mécanismes vus précédemment lors de l'initiation à la programmation objet.

8.2 Implémentation d'une propriété de formulaire

Là encore nous retrouvons les principes de l'objet.

Si nous souhaitons passer le texte affiché dans le titre de la fenêtre de détail à partir d'éléments de la fenêtre de liste, nous pouvons implémenter une nouvelle propriété Titre au niveau du formulaire de détail

Dans la fenêtre de détail :

```
public string Titre
{
    get { return _titre; }
    set { _titre = value; }
}
```

```
private void FrmDetailSalarie_Load(object sender, EventArgs e)
{
    this.Text += this._titre;
}
```

Dans la fenêtre Liste

```
frmDetailSalarie.Titre = "Modification de salarié faite par " + this.Text;
```

Le résultat en image :

