

MARTIN Pierre  
MONTE Nicolas  
OLAGNON Loïc

# Logiciel de Design d'interfaces graphiques « VisualDev<sup>©</sup> »



## Rapport final

**Responsable :** MONTE Nicolas

**Version :** 1.0 du 22/03/2006

**Statut du document :** Terminé

# Sommaire

<b>INTRODUCTION.....</b>	<b>3</b>
<b>1. EXPRESSION DU BESOIN.....</b>	<b>5</b>
1.1. DEFINITION DU PROBLEME.....	5
1.2. OBJECTIFS.....	6
<b>2. CHOIX TECHNOLOGIQUES.....</b>	<b>7</b>
2.1. LE C#.....	7
2.2. LE XML.....	7
<b>3. DIAGRAMMES UML.....</b>	<b>8</b>
3.1. CAS D'UTILISATIONS.....	8
3.2. CLASSES.....	10
3.3. SEQUENCES.....	11
3.4. ACTIVITES.....	13
<b>4. GESTION DU PROJET.....</b>	<b>15</b>
<b>5. NOTIONS UTILISEES.....</b>	<b>16</b>
5.1. REFLEXION.....	16
5.2. SERIALISATION.....	16
<b>6. INTERFACE GRAPHIQUE.....</b>	<b>17</b>
6.1. FORMULAIRE PRINCIPALE : FRMMAIN.....	17
6.2. FORMULAIRE DE GESTION DES CONTROLES : LISTCONTROLSDIALOG.....	22
<b>7. UN CONTROLE ACTIVEX : SELECTOR.....</b>	<b>23</b>
7.1. LES CONTRAINTES.....	23
7.2. LES SOLUTIONS.....	24
<b>8. NOTRE NORME XML.....</b>	<b>25</b>
8.1. NOTION SUR LE XML.....	25
8.2. LA NORME VISUALDEV.....	26
8.3. CONCLUSION.....	28

<b>9.</b>	<b>ENREGISTREMENT .....</b>	<b>29</b>
9.1.	TROUVER DYNAMIQUEMENT LES PROPRIETES D'UN CONTROLE .....	29
9.2.	ALGORITHME SIMPLIFIE DE LA FONCTION SAUVEGARDE .....	29
9.3.	RECUPERER LES PROPRIETES PRIVEES D'UN CONTROLE.....	30
9.4.	EST-IL NECESSAIRE DE TOUT SAUVEGARDER ?.....	30
9.5.	LES DIFFERENTS TYPES RENCONTRES .....	30
9.6.	ENREGISTRER LES IMAGES ET ICONES : .....	31
9.7.	ENREGISTRER LES DONNEES PAR LA SERIALISATION .....	31
9.8.	FONCTION DE SAUVEGARDE FINALE .....	31
<b>10.</b>	<b>CHARGEMENT .....</b>	<b>32</b>
10.1.	PARSER UN FICHIER XML :.....	32
10.2.	CREER DYNAMIQUEMENT DES CONTROLES AVEC LEUR NOM.....	32
10.3.	LEUR ASSIGNER DES PROPRIETES .....	33
<b>11.</b>	<b>BILAN.....</b>	<b>34</b>
11.1.	BILAN DU PROJET .....	34
11.2.	BILAN DES CONNAISSANCES.....	34
11.3.	BILAN HUMAIN.....	34
<b>12.</b>	<b>DIFFICULTES RENCONTREES .....</b>	<b>35</b>
<b>13.</b>	<b>BIBLIOGRAPHIE.....</b>	<b>36</b>
<b>14.</b>	<b>DOCUMENTATION.....</b>	<b>37</b>
14.1.	MODE D'EMPLOI DE L'INSTALLATION. ....	37
14.2.	GERER LES PROJETS .....	38
14.3.	MODIFIER LA LISTE DES CONTROLES .....	39
14.4.	GERER LES CONTROLES.....	40
14.5.	AUTRES.....	41
<b>15.</b>	<b>GLOSSAIRE .....</b>	<b>42</b>

## Introduction

Le but de ce projet est de réaliser un logiciel de design d'interface utilisateur dans l'environnement .Net<sup>1</sup>. Ce designer doit fonctionner d'une manière visuelle comme l'outil de design intégré dans Visual Studio (Boîte à outils, espace de travail et fenêtre de propriétés des objets en cours de modification). Mais à la grande différence de Visual Studio, il ne produira pas de code mais un fichier descriptif de l'IHM<sup>2</sup> en XML<sup>3</sup>.

## Remerciements

Nous tenons à remercier Mr Canitia, notre encadrant de projet tuteuré, pour ses conseils avisés. Nous remercions aussi toutes les personnes ayant apporté une aide à l'aboutissement de notre projet. Entre autres, del-dongo membre du forum de developpez.com.

---

<sup>1</sup> Voir glossaire

<sup>2</sup> Voir glossaire

<sup>3</sup> Voir chapitre 8.1 – Notion sur le XML

# 1. Expression du besoin

## 1.1. Définition du problème

Nous devons réaliser un logiciel qui permettra à un utilisateur quelconque de créer une interface graphique<sup>1</sup>. Il disposera d'une palette avec les différents contrôles<sup>2</sup> qu'il pourra insérer dans son interface. Cette interface sera sauvegardée en un fichier XML qu'il sera possible de modifier pour changer l'apparence visuelle de l'interface créée. Pour cela nous allons développer cette application avec Visual Studio.Net (VS.Net).

### 1.1.1. Besoins des utilisateurs

Tout d'abord, l'utilisateur doit pouvoir créer l'interface graphique de son application. Il faut que notre logiciel soit simple d'utilisation. Il pourra :

- Créer un projet
- Ajouter des formulaires<sup>3</sup>
- Ajouter/Modifier les différents éléments (Button, TextBox, PictureBox, etc.)
- Enregistrer/Charger son projet au format XML

### 1.1.2. Contexte

Le logiciel doit pouvoir être utilisé à partir de n'importe quelle machine disposant d'une version récente du système d'exploitation Windows ainsi que du Framework .Net<sup>4</sup>. La cible visée est l'ensemble des utilisateurs, expérimentés ou non.

---

<sup>1</sup> Voir glossaire

<sup>2</sup> Voir glossaire

<sup>3</sup> Voir glossaire

<sup>4</sup> Voir glossaire

## 1.2. Objectifs

### 1.2.1. Objectifs principaux

Notre application devra posséder au minimum les fonctionnalités suivantes :

- Une bibliothèque de composants visuels<sup>1</sup> disponibles sous forme de boîte à outils dans laquelle on pourra évidemment ajouter des composants ou en enlever.
- Un espace de travail représenté par une fenêtre dans lequel on pourra réaliser graphiquement l'interface utilisateur désirée. Cet espace peut posséder des outils d'alignement, une grille de positionnement,...
- Une fenêtre permettant de visualiser les valeurs des propriétés des objets sélectionnés. Cette fenêtre permettra la modification des valeurs.
- Une fenêtre affichant les différents fichiers du projet en cours.

### 1.2.2. Objectifs optionnels

D'autres fonctionnalités pourraient être très utiles à notre application :

- Faire des liens dynamiques entre les événements<sup>2</sup> des contrôles et les méthodes<sup>3</sup> désignées par l'utilisateur.
- La possibilité pour l'utilisateur de charger ses propres librairies<sup>4</sup> de contrôles, c'est-à-dire des contrôles utilisateurs.
- Créer un logiciel complémentaire permettant de charger le projet de l'utilisateur sans passer par VisualDev. Ainsi cette application devra lire le fichier XML et recréer l'interface de l'utilisateur, en lui fournissant les liens événements – méthodes.

---

<sup>1</sup> Voir glossaire

<sup>2</sup> Voir glossaire

<sup>3</sup> Voir glossaire

<sup>4</sup> Voir glossaire

## 2. Choix technologiques

### 2.1. Le C#

Le langage qui nous a été imposé pour réaliser notre application est le C#. Ce choix se justifie aisément :

- C'est un langage orienté visuel, grâce à VS.Net, ce qui correspond bien à notre type d'application.
- Beaucoup de mécanismes propres à C# ont été utiles dans la réalisation de notre projet.
- C'est un langage que nous ne connaissions pas, mais qui a été très intéressant à apprendre. Il est par ailleurs en pleine expansion et il pourra nous servir dans les années à venir.

L'implémentation se fera sur le Framework .Net. Ce Framework a été choisi pour ses fonctionnalités et leur facilité de mise en œuvre.

### 2.2. Le XML

L'interface créée sous VisualDev sera sauvée dans un fichier XML.

Dans un premier temps et suite à une piste fournie par notre encadrant nous avons dirigé nos recherches vers la norme XUL. XUL est un langage de création d'interface graphique au format XML, créé par Mozilla<sup>1</sup>. Le langage XUL est gratuit, fiable et reconnu contrairement à ses concurrents.

Par la suite, nous nous sommes rendu compte que le choix de cette norme allait limiter l'enregistrement des projets créés par VisualDev. En effet, elle ne nous permet pas d'enregistrer toutes les propriétés d'un contrôle telles que les propriétés FlatStyle, AnchorStyle, Dock, etc.... De plus il serait impossible d'enregistrer des types importés par l'utilisateur.

Nous avons donc fait le choix de rédiger notre propre norme XML.

---

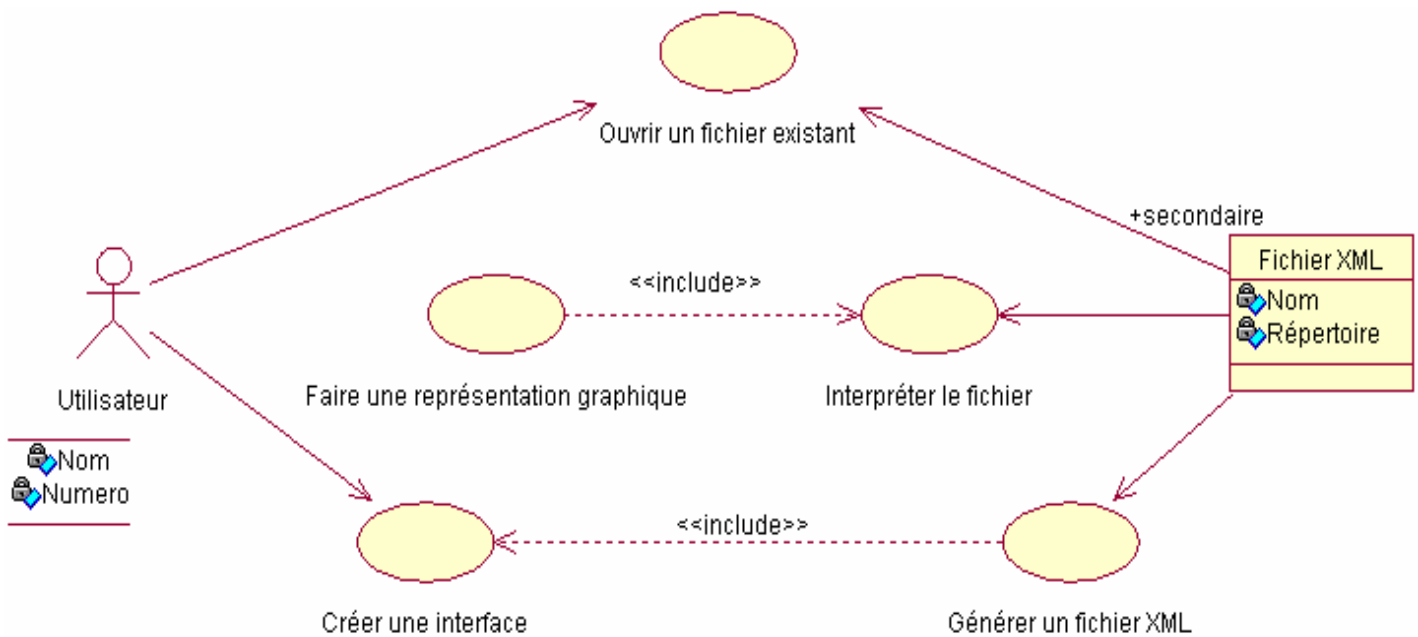
<sup>1</sup> Société créatrice du célèbre navigateur FireFox.

# 3. Diagrammes UML

## 3.1. Cas d'utilisations

Le diagramme de cas d'utilisation décrit les possibilités d'interaction entre le système et les acteurs, c'est-à-dire toutes les fonctionnalités que doit fournir le système.

### 3.1.1. Schéma



MARTIN Pierre  
MONTE Nicolas  
OLAGNON Loïc  
Diagramme de cas d'utilisation  
Version : 1.0

### 3.1.2. Description

#### 3.1.2.1. Ouvrir un fichier existant :

Résumé : Nous entendons par ouvrir un fichier existant, le fait que l'utilisateur de notre logiciel puisse ouvrir un fichier de son projet afin de l'éditer.

Acteurs : Pour ce cas d'utilisation nous trouvons l'utilisateur et le fichier XML (qui joue un rôle secondaire).

Pré conditions : Pour que le fichier soit lisible il faut qu'il soit implémenté dans la norme définie c'est-à-dire la norme XML de VisualDev.

#### 3.1.2.2. Interpréter un fichier :

Résumé : L'application que nous allons développer doit être capable de traduire un fichier XML fourni afin de le convertir en interface.

Acteurs : Le fichier XML

#### 3.1.2.3. Faire une représentation graphique :

Résumé : Cette action vient à la suite de l'interprétation du fichier. Une fois que le fichier est interprété il faut que l'application le traduise en représentation graphique.

Acteurs : Il n'y en a pas car c'est à l'application de réaliser cette tâche.

#### 3.1.2.4. Créer une interface :

Résumé : L'utilisateur doit pouvoir créer sa propre interface. Il doit pouvoir ajouter, modifier des contrôles à son interface.

Acteurs : L'utilisateur.

#### 3.1.2.5. Générer un fichier XML :

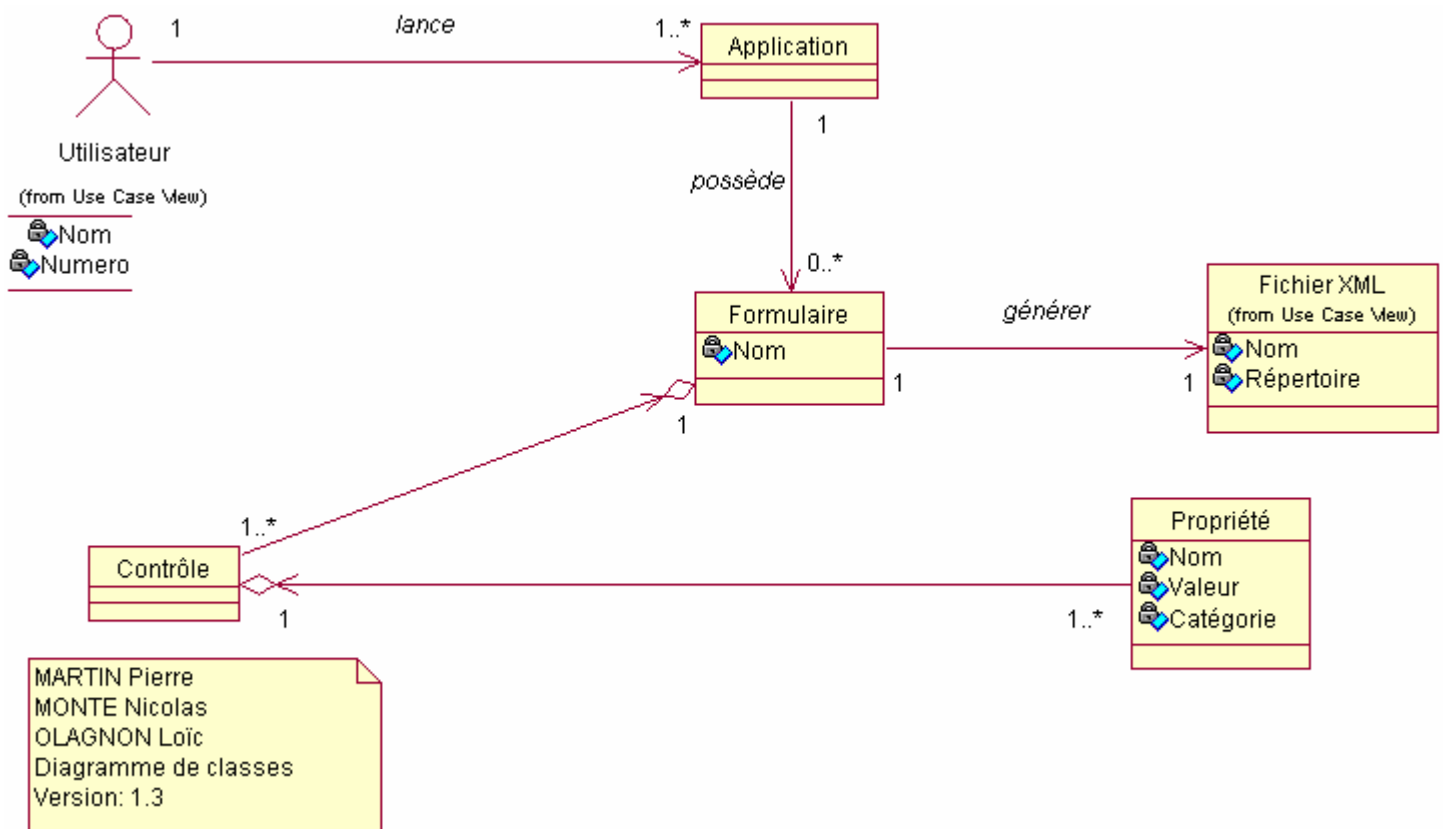
Résumé : Après la création d'une interface par l'utilisateur, l'application doit pouvoir générer un fichier XML qui contiendra toutes les informations nécessaires pour pouvoir re-modéliser l'interface qui aura été créée.

Acteurs : Le fichier XML sera le composant résultant de cette tâche.

## 3.2. Classes

Le diagramme de classes, utilisé en génie logiciel, représente les classes et les interfaces d'un système ainsi que les différentes relations entre celles-ci. Une classe est un ensemble de méthodes et d'attributs (données) qui sont liés ensemble par un champ sémantique. Les classes permettent de modulariser un programme pour découper une tâche complexe en plusieurs petits travaux simples.

### 3.2.1. Schéma



### 3.2.2. Description

L'utilisateur lance une ou plusieurs applications VisualDev. Chaque application permet de créer des formulaires.

Un formulaire peut contenir des contrôles visuels ajoutés et modifiés par l'utilisateur.

Chaque contrôle contient une liste de propriétés (Taille, position, image, texte, etc.). La plupart de ces propriétés sont modifiables par l'utilisateur.

Un formulaire peut être généré sous forme d'un fichier au format XML. Ce fichier est enregistré dans un répertoire sous un nom choisi par l'utilisateur.

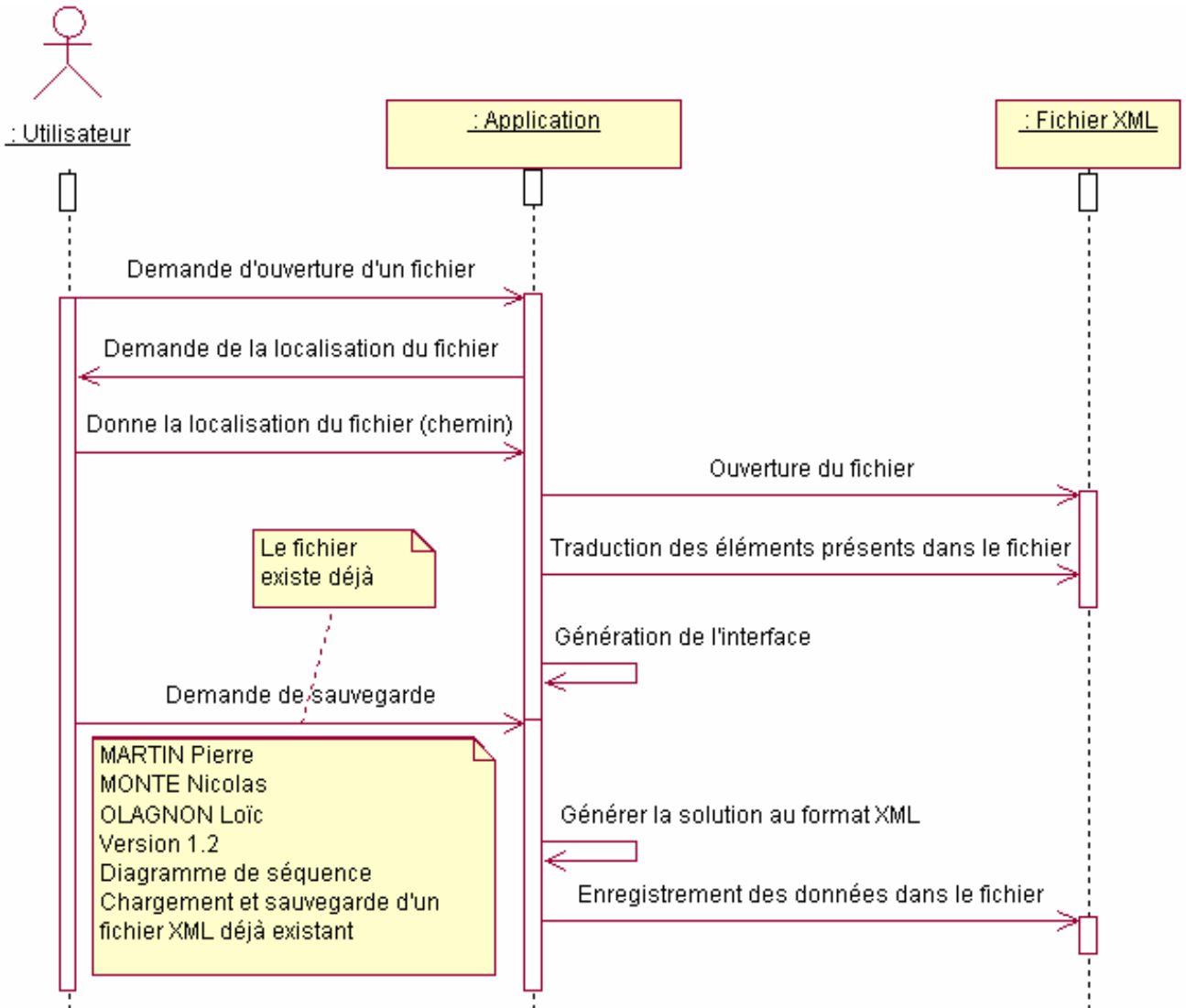
Le fichier XML contient la description du formulaire, à savoir ses contrôles et leurs propriétés.

### 3.3. Séquences

Le diagramme de séquences est une représentation séquentielle du déroulement des traitements et des interactions entre les éléments du système et/ou des acteurs.

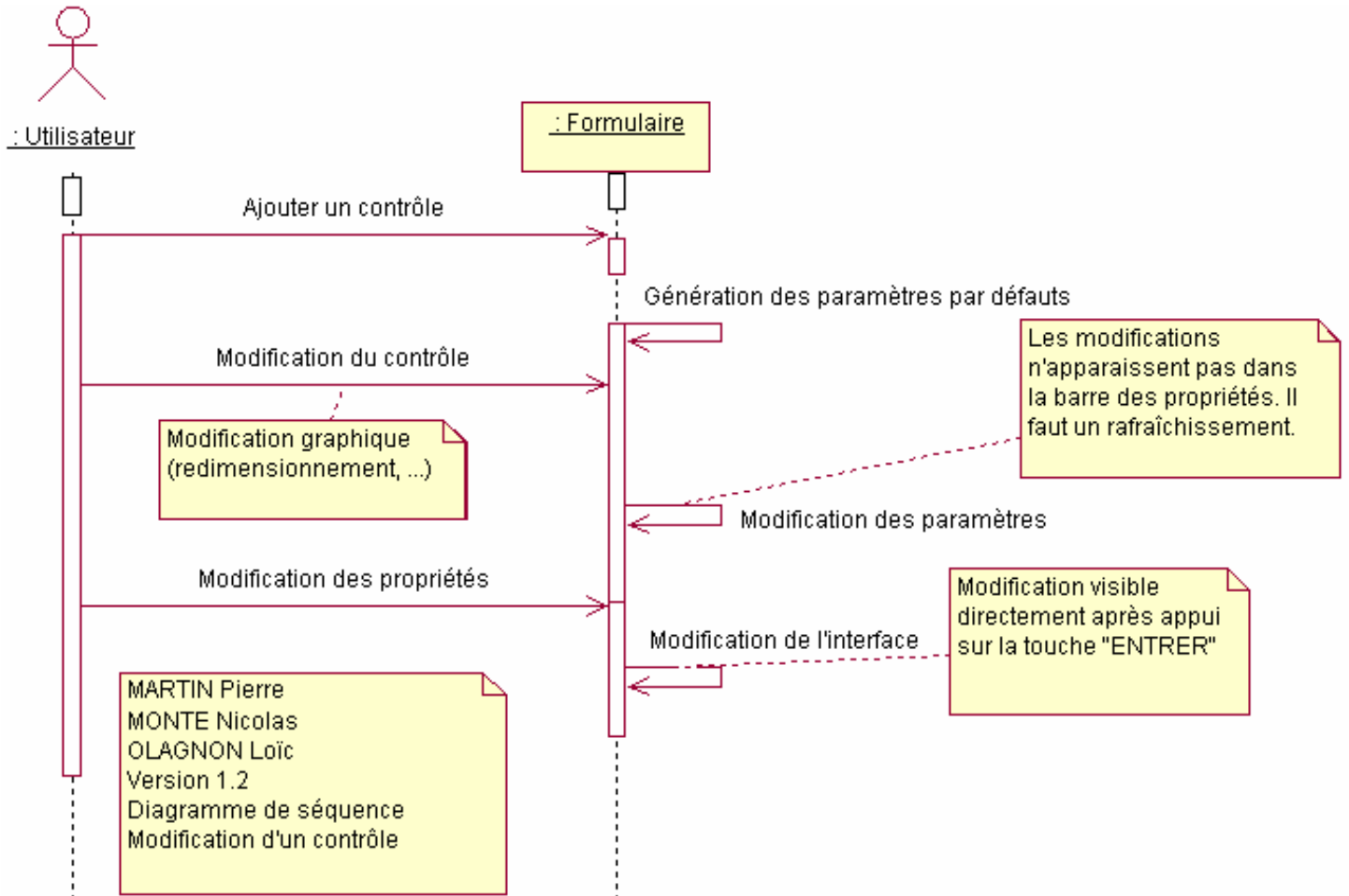
#### 3.3.1. Chargement et sauvegarde de fichiers

Ce diagramme de séquence décrit les processus de sauvegarde et de chargement d'un fichier projet au format XML. L'utilisateur choisit de charger un fichier, l'application ouvre alors une fenêtre lui permettant de choisir le fichier à ouvrir. Ensuite, l'application utilise les données présentes dans le fichier pour générer l'interface. Lorsque l'utilisateur souhaite sauvegarder son interface, l'application génère un code XML dans un fichier. Si le fichier n'existe pas, l'application demande à l'utilisateur de choisir la destination.



### 3.3.2. Modification d'une interface

L'utilisateur peut modifier son interface, en modifiant les formulaires et les propriétés. S'il ajoute un contrôle, l'application va l'afficher et générer les propriétés relatives à celui-ci. Ensuite l'utilisateur peut choisir de les modifier. Si l'utilisateur modifie une valeur directement dans le champ de propriété alors les modifications seront appliquées immédiatement au contrôle.



## 3.4. Activités

Le diagramme d'activités montre la façon dont l'état du système est modifié en fonction des événements du système. Il permet de représenter le déclenchement d'événements en fonction des états du système.

### 3.4.1. Description

Ce diagramme représente les différentes utilisations qu'un utilisateur peut faire avec le logiciel. Tout d'abord il peut créer un nouveau fichier. Cette action va entraîner la création d'un formulaire avec des propriétés prédéfinies. Un message d'erreur lui indique si la création a échoué.

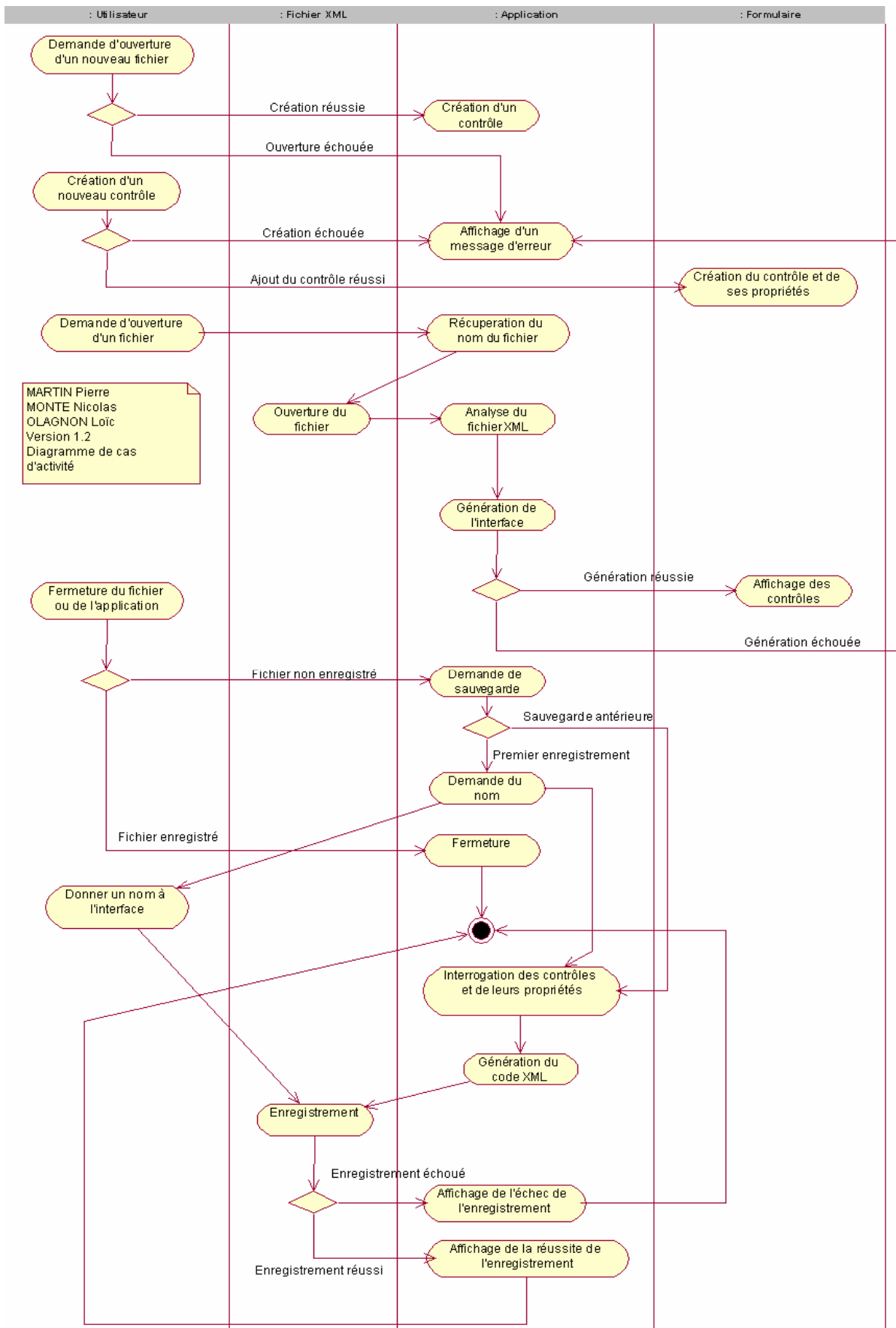
L'utilisateur peut ensuite insérer de nouveaux contrôles (Button, Label, etc.)

Dans le cas où l'utilisateur souhaite ouvrir une interface existante, le logiciel va lire le fichier XML sélectionné et générer l'interface correspondante. Si la génération de l'interface réussie alors celle-ci s'affiche à l'écran, sinon un message d'erreur prévient l'utilisateur.

L'utilisateur peut également choisir de sauvegarder le fichier. S'il s'agit du premier enregistrement, le logiciel va demander le nom sous lequel l'interface devra être sauvegardée. La sauvegarde peut alors avoir lieu. Mais si elle échoue un message d'erreur prévient l'utilisateur.

Pour la fermeture de l'interface ou de l'application, si l'interface a été enregistrée avant la demande de fermeture, celle-ci s'effectue. Sinon une demande est faite à l'utilisateur afin de s'assurer qu'il ne ferme pas le logiciel par erreur.

### 3.4.2. Schéma



## 4. Gestion du projet

Pour commencer, nous avons désigné Nicolas comme chef de projet dont le rôle a été de répartir les différentes tâches entre les membres et de faire régulièrement des bilans de l'agencement pour recentrer les priorités et redéfinir les objectifs en fonction des éléments déjà accomplis.

Pour mener à bien ce projet, nous avons débuté par une phase d'analyse durant laquelle nous avons défini ce que notre logiciel sera capable de réaliser. Loïc s'est occupé de toute la partie étude conceptuelle en utilisant la méthode d'analyse UML<sup>1</sup>. De cette étape est ressortie une liste de fonctionnalités que nous allions devoir réaliser.

Après avoir posé les limites et contraintes, Pierre a pu débiter l'implémentation de l'interface. A la suite d'un premier jet, il s'est lancé dans la programmation des différentes fonctionnalités (ajouts de contrôles, redimensionnement, fonctions d'alignements, etc....). Loïc quant à lui, est intervenu dans l'intégration des fonctions créées par Pierre dans le logiciel ainsi qu'à l'élaboration de l'ergonomie suivant les conseils de Nicolas et Pierre.

Pendant ce temps, Nicolas s'est occupé de la définition de la norme XML à utiliser pour l'enregistrement. Une fois la norme établie, il s'est occupé de la réalisation des fonctions d'enregistrement puis de chargement. Fonctions qui ont été beaucoup plus longues à mettre en place que ce qui était initialement prévu.

Enfin, Loïc s'est occupé de réaliser un programme d'installation pour le logiciel et de rédiger une documentation technique pour aider l'utilisateur à utiliser notre logiciel.

---

<sup>1</sup> Voir glossaire

# 5. Notions utilisées

## 5.1. Réflexion

La réflexion est l'art de découvrir des types et d'invoquer leurs membres à l'exécution. La réflexion permet d'inspecter dynamiquement le contenu d'*assemblages*, d'en lire ses types, de créer des instances de ces types durant l'exécution du programme et d'appeler leurs méthodes ou champs dynamiquement. La réflexion est également appelée **introspection** suivant les auteurs, mais, avec l'arrivée de .NET, le terme **réflexion** s'est généralisé. On utilise également le terme de **liaison tardive** (*late binding*) pour décrire une instanciation à la volée au moment de l'exécution. Il existe d'autres mécanismes s'approchant de la réflexion utilisés avant .NET, comme RTTI (*Run-Time Type Identification*) en C/C++ ou, dans une certaine mesure, l'interface *IDispatch* pour les composants COM. On peut définir la réflexion comme le fait de récupérer les attributs d'un objet ou de sa classe de manière dynamique.

*Source : <http://emericadeveloppez.com/dotnet/reflection/introduction/csharp/>*

Dans notre application, cette notion nous a été extrêmement utile, même indispensable. En effet nous nous sommes servis de la réflexion pour tout ce qui est création de liste de contrôles, accès aux propriétés lors de l'enregistrement/chargement, lecture de méthodes, d'événements ou encore de contrôles utilisateurs dans des bibliothèques dynamiques.

De plus l'instanciation à la volée a été utilisée à outrance. Particulièrement grâce à la fonction *Activator.CreateInstance(System.Type)*, qui crée une instance d'un objet dont on connaît le Type.

## 5.2. Sérialisation

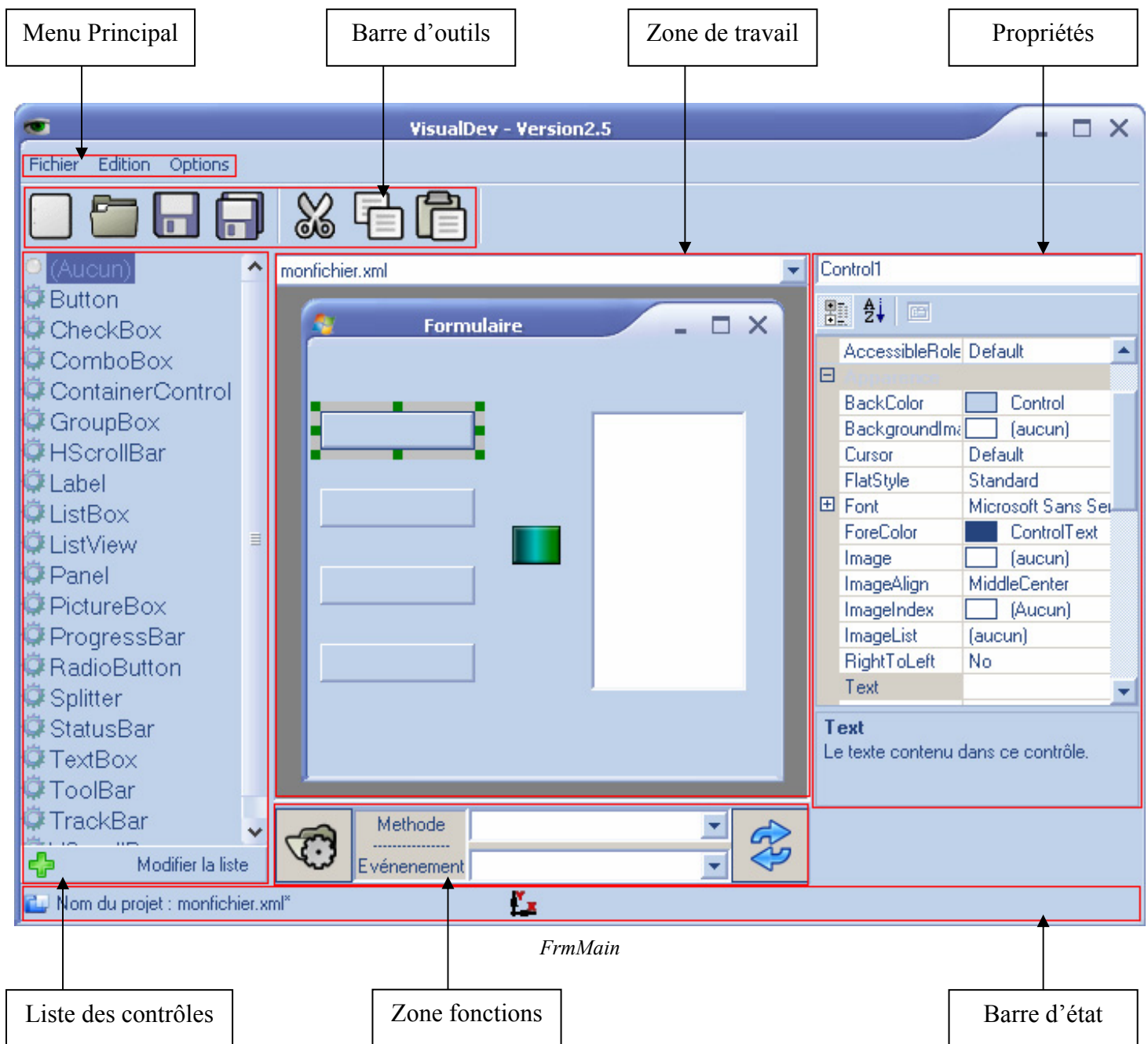
La sérialisation consiste à transformer un objet en mémoire en un flux<sup>1</sup> de données séquentiel. Cela permet de transférer un objet vers un flux de données (sauvegarder sur fichier, envoi par le réseau, ...). La désérialisation correspond à l'opération inverse : à partir d'un flux de données séquentiel, il faut recréer l'objet correspondant.

---

<sup>1</sup> Voir glossaire

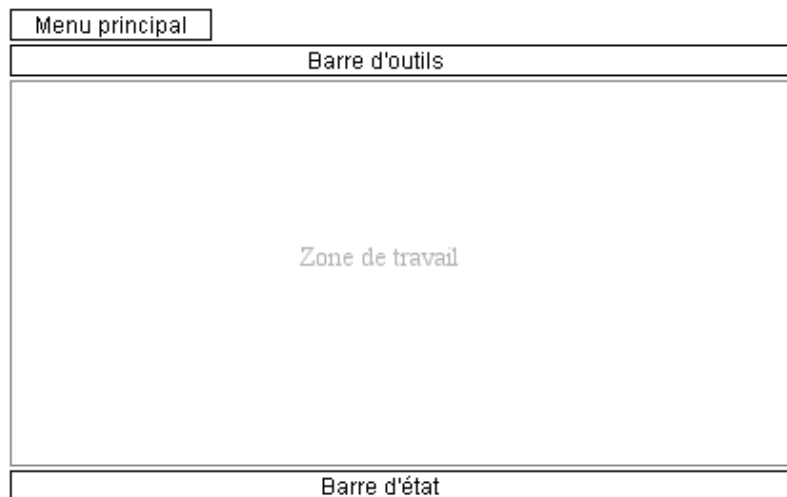
# 6. Interface graphique

## 6.1. Formulaire principale : FrmMain



Nous avons commencé par définir une interface classique pour une application Windows :

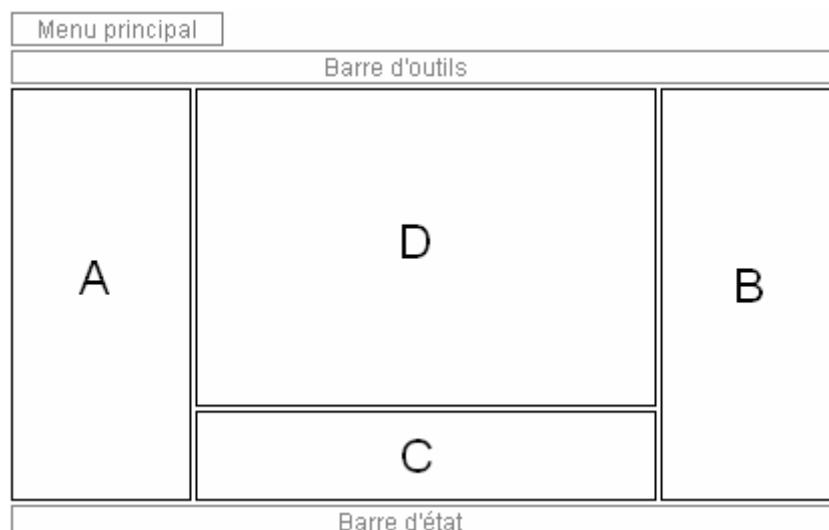
- Un **menu principal** (Fichier, Edition, Options...)
- Une **barre d'outils** (Nouveau, Ouvrir, Copier, Coller...)
- Une **barre d'état** (Etat du projet, Coordonnées d'un point...)



*Une interface classique d'application Windows*

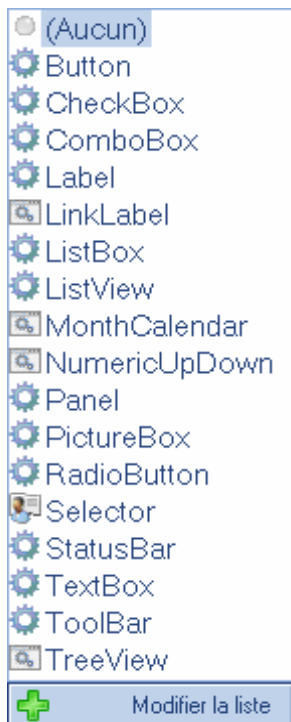
Compte tenu du fait que notre application est orientée programmation visuelle, nous avons décidé de reprendre certains concepts de VS.Net, en y apportant quelques modifications, puis nous avons ajouté de nouvelles fonctionnalités qui nous semblaient intéressantes.

Nous avons découpé l'interface principale en 4 zones comme le montre ce schéma :



*Les 4 zones de l'interface principale*

### 6.1.1. Liste des contrôles (A)



Liste des contrôles

C'est une ListView contenant les contrôles que l'utilisateur peut ajouter à son projet.

Comme dans VS.Net, c'est une liste d'éléments sélectionnables. Mais dans notre version, il n'y a pas de drag&drop<sup>1</sup> vers la zone de travail (D), ainsi nous laissons à l'utilisateur la possibilité de dimensionner son contrôle en créant un cadre lors de l'insertion.

L'utilisateur a aussi la possibilité de compléter cette liste en cliquant sur le bouton « Modifier la liste ».

La création de la liste a été réalisée grâce à l'utilisation de la réflexion<sup>2</sup>. Ce système nous a permis d'obtenir tous les types dérivés de la classe Control<sup>3</sup>. Nous avons ensuite épuré cette liste pour ne garder que les composants visuels.

C'est là qu'intervient la classe **TypesVisualDev** dont tous les membres sont statiques. Cette classe possède 4 ArrayList<sup>4</sup> contenant des types de contrôles :

- **TypesBase** : Types fournis à l'initialisation de VisualDev.
- **TypesInvalides** : Types à bannir, c'est-à-dire ceux que nous estimons inutiles ou techniquement inutilisables.
- **AllTypes** : Tous les types disponibles, c'est-à-dire les types issus de la classe Control moins les TypesInvalides.
- **UserTypes** : Types chargés à partir de DLL par l'utilisateur.

La classe TypesVisualDev peut également initialiser ces listes de types, grâce aux méthodes *InitTypesBase()*, *InitTypesInvalides()*, *InitAllTypes()*, *InitUserTypes()*.

L'intérêt de cette classe est que nous pouvons tester un type à n'importe quel endroit du programme. Cette classe possède aussi la fonction *Type getTypeFromName(String nom)* qui retourne le Type dont on passe le nom en paramètre.

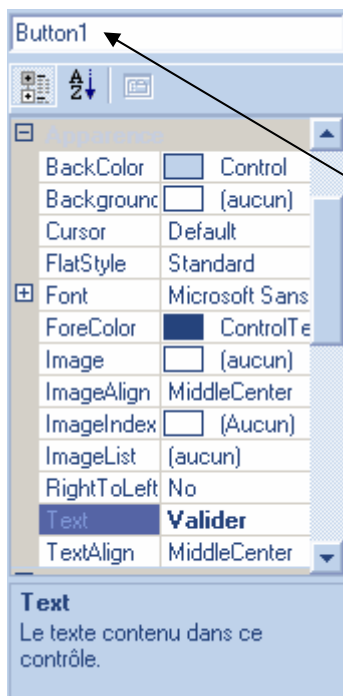
<sup>1</sup> Voir glossaire

<sup>2</sup> Voir chapitre 5.1 – Réflexion

<sup>3</sup> Voir glossaire : Héritage

<sup>4</sup> Nous avons préféré travailler sur des ArrayList plutôt que sur des tableaux de Types car elles possèdent beaucoup plus de fonctionnalités (Add, Remove, etc.)

### 6.1.2. Propriétés (B)



Cette zone est réservée à l'affichage des propriétés des contrôles sélectionnés.

L'utilisateur peut accéder à certaines propriétés des contrôles et les modifier.

Le nom du contrôle sélectionné peut être modifié dans ce TextBox.

Le composant visuel utilisé pour visionner et modifier les propriétés est une PropertyGrid. Ce composant est très utile dans notre application, et très simple d'utilisation.

En effet, il suffit de lui indiquer le ou les objets dont on veut afficher les propriétés, le reste est géré automatiquement. La modification des propriétés agit directement sur le ou les contrôles sélectionnés.

En fait, ce composant est le même que celui utilisé dans VS.Net, donc les utilisateurs de ce dernier seront déjà habitués.

### 6.1.3. Zone fonctions (C)

Notre application permet à l'utilisateur de créer des interfaces graphiques, sans qu'il ait besoin d'implémenter de code. Toutefois dans cette zone il pourra sélectionner les méthodes chargées à partir de DLL<sup>1</sup>, puis les associer aux événements des contrôles, afin de rendre le projet plus dynamique.

Pour mieux comprendre cette fonctionnalité, regardons l'exemple ci-dessous :



En cliquant sur le bouton « Charger » situé à gauche, l'utilisateur a chargé une liste de méthodes statiques présentes dans une librairie dynamique. Ici il a choisit la méthode *Void test()*.

Après cela il sélectionne un contrôle, ce qui a pour effet de remplir la liste du bas avec les événements de ce contrôle. Ensuite il choisit un événement particulier, ici *Click*.

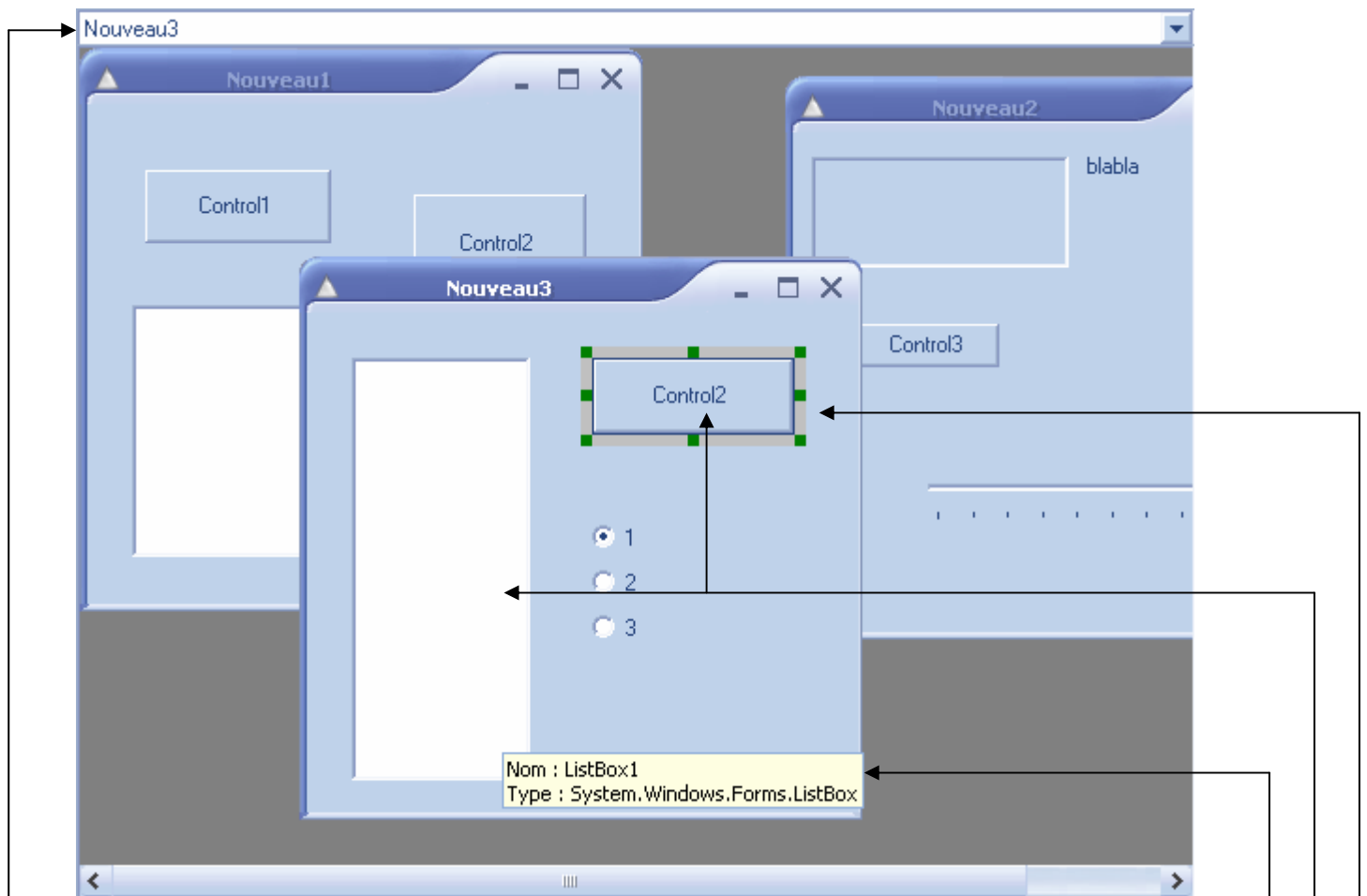
Il ne reste plus qu'à cliquer sur le bouton « Lier », à droite, pour lier la méthode sélectionnée à l'événement.



Malheureusement à cause du manque de temps cette fonctionnalité n'a pas pu être implémentée.

<sup>1</sup> Voir glossaire

#### 6.1.4. Zone de travail (D)



C'est la zone principale sur laquelle on opère. On y trouve les projets créés par l'utilisateur. Ce sont des formulaires MDI<sup>1</sup>, ce qui permet à l'utilisateur de naviguer facilement entre les projets.

De plus notre application possède un ComboBox qui contient les références aux projets ouverts, au cas où l'utilisateur se perde dans ses formulaires.

Dans la première version de notre projet, nous avons utilisé un système d'onglets similaire à VS.net. Mais le système actuel est plus pratique.

Les formulaires sont composés de contrôles visuels.

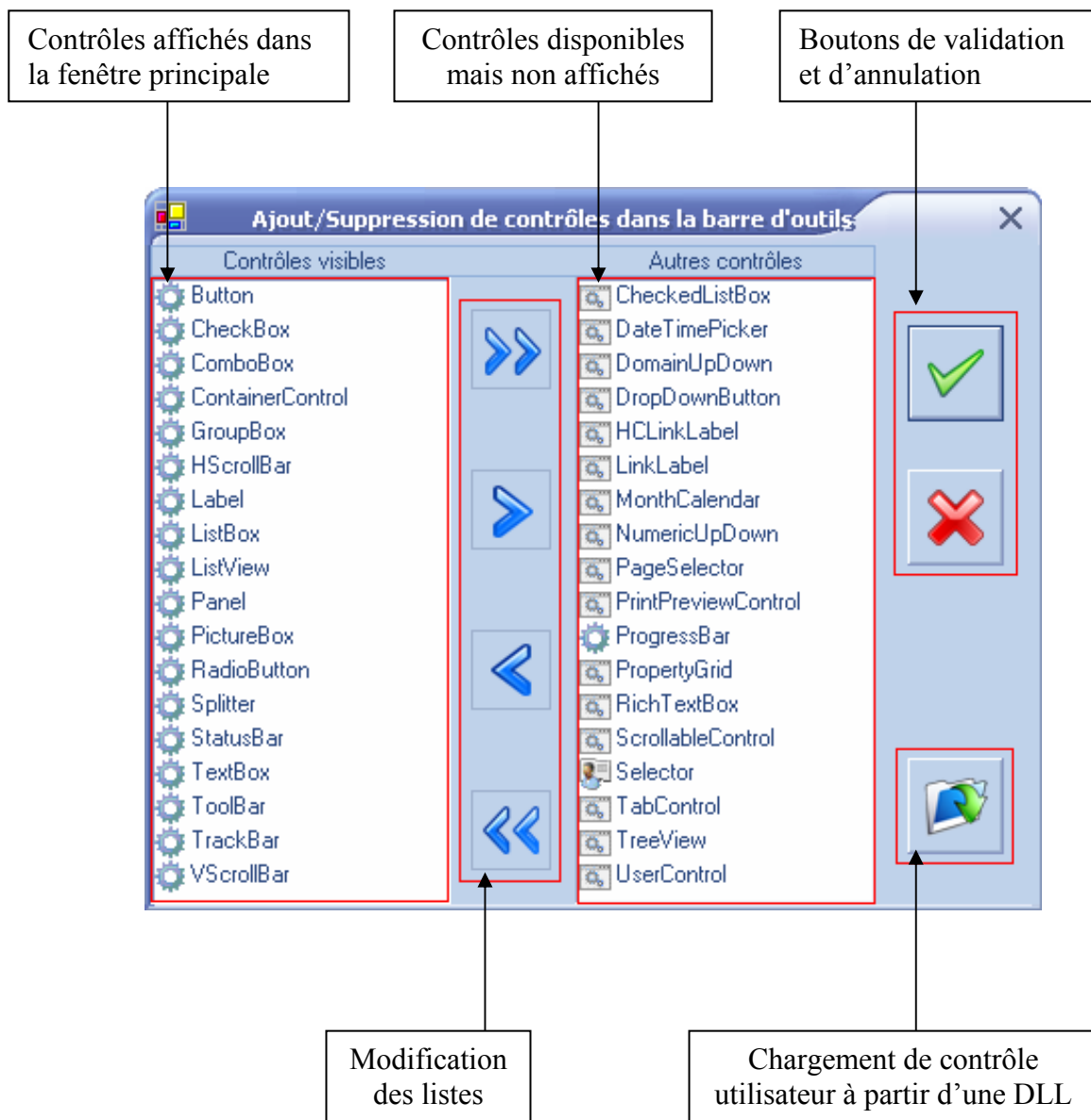
Le nom du contrôle ainsi que son type peuvent être facilement vus par l'utilisation d'un Tooltip.

On peut également sélectionner un contrôle afin de le déplacer ou de le redimensionner. Pour cela on utilise un Selector<sup>2</sup>.

<sup>1</sup> Voir glossaire

<sup>2</sup> Voir chapitre 7 – Un contrôle ActiveX : Selector

## 6.2. Formulaire de gestion des contrôles : ListControlsDialog



Ce formulaire est appelé lorsque l'utilisateur clique sur le bouton « Modifier la liste » disponible sur le formulaire principal.

En effet, ici l'utilisateur sera en mesure de créer sa liste personnalisée de contrôles. Pour cela nous mettons à sa disposition deux listes.

- « Contrôles visibles » : C'est la liste personnalisée que l'utilisateur peut voir dans la fenêtre principale.

- « Autres contrôles » : C'est une liste contenant des contrôles en 'réserve'. Ils sont à la disposition de l'utilisateur pour qu'il puisse les ajouter dans sa liste personnalisée.

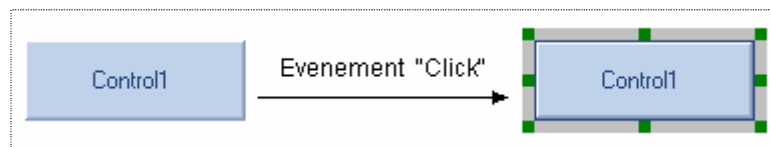
Ce formulaire est appelé en tant que fenêtre modale. C'est-à-dire qu'une fois qu'elle est appelée, l'utilisateur ne peut plus agir sur les autres formulaires. Il doit donc valider ou annuler le formulaire.

# 7. Un contrôle ActiveX : Selector

Un des objectifs de notre projet était de permettre à l'utilisateur de modifier simplement une interface graphique. Ceci inclus donc le déplacement et le dimensionnement des contrôles.

L'outil utilisé sous VS.Net pour effectuer cette tâche est très pratique, c'est pourquoi nous avons cherché un équivalent, simple d'utilisation. Après des recherches sans succès, nous avons finalement pris la décision de créer un contrôle capable de réaliser cette tâche.

Voici une représentation de la sélection d'un contrôle : A l'événement « Click » sur un contrôle, celui-ci est sélectionné.



## 7.1. Les contraintes

Dès lors, nous nous sommes fixés plusieurs objectifs en ce qui concernait ce contrôle. Il devait répondre à certaines contraintes :

- Simplicité : cet outil devait être facilement manipulable.
- Performance : il devait fournir toutes les fonctionnalités requises, et demander un minimum de ressources.
- Indépendance : ce composant devait effectuer le maximum de tâches sans avoir besoin d'intervenir. Ce point était crucial, car cette contrainte détermine la simplicité et la réutilisabilité du contrôle.
- Réutilisabilité : ce composant doit théoriquement être utilisable dans d'autres applications que VisualDev.

## 7.2. Les solutions

Pour répondre à la contrainte de simplicité, la classe Selector possède trois membres principaux capables de « discuter » avec le contrôle sélectionné :

- Une propriété SelectedControl de type Control. Cet attribut est défini à null au chargement. Il peut être assigné d'une référence sur un contrôle, grâce au mutateur<sup>1</sup> *SetControl(Control ctrl)*. On peut également obtenir cette référence avec l'accessor<sup>2</sup> *Control GetControl()*.
- Une méthode *Refresh()* qui permet de rafraîchir la position/taille du Selector si on modifie la position/taille du contrôle.
- Une propriété Selection de type Selector[]. C'est grâce à cet attribut que l'utilisateur peut passer un tableau de Selector, car dans le cas d'une multi sélection, un Selector doit mettre à jour tous les autres, notamment pour le déplacement.

Le Selector se fixe sur un contrôle, c'est-à-dire qu'il prend la taille et la position de celui-ci. A partir de cet instant si le Selector est modifié, il modifie automatiquement le contrôle sélectionné. Ceci résout la contrainte d'indépendance du composant.

Le Selector est donc aussi réutilisable dans d'autres applications. De plus il est « performant » au niveau ressource car il n'utilise pas de fonction trop gourmande.

→ Nous avons donc trouvé des solutions adaptées à chaque contrainte.

---

<sup>1</sup> Voir glossaire

<sup>2</sup> Voir glossaire

# 8. Notre norme XML

## 8.1. Notion sur le XML

Pour enregistrer une interface nous avons commencé par définir une norme XML.

Le XML abrégé de Extensible Markup Language (« langage de balisage extensible »), est un standard du W3C (World Wide Web Consortium) qui sert de base pour créer des langages de balisage : c'est un « métalangage<sup>1</sup> ». Le XML permet de définir un vocabulaire et une grammaire associée à une base de règles formalisées.

Un fichier XML est un fichier texte. Le fichier XML est structuré en « éléments » (ou nœuds) à l'aide de « balises » qui marquent le début et la fin de chaque élément. Les éléments peuvent contenir du texte et éventuellement d'autres éléments. L'ensemble des données du document XML est contenu dans un élément unique appelé « racine ». Ce dernier contient tous les autres éléments.

Pour établir notre propre norme et obtenir des fichiers XML bien formés nous avons donc dû suivre quelques règles :

- Un fichier XML doit commencer par un *prologue*. Le prologue identifie le document comme étant un document XML, déclare le codage (**encoding**) et indique si le document est lié à d'autres fichiers ou s'il est autonome (**standalone**).
- Un document XML ne doit avoir qu'un seul élément *racine*. Tous les autres éléments sont contenus dans cet élément.
- Chaque élément XML doit commencer par une balise ouvrante et se terminer par une balise fermante. Un élément vide peut être représenté par une balise d'élément vide qui ressemble à ceci : `<exemple/>`, cette balise est considérée comme étant une balise ouvrante suivie d'une balise fermante. Ceci est utilisé pour éviter de devoir écrire `<exemple></exemple>` tout en conservant le bon formatage. Le XML tient compte de la case, ainsi `<Exemple></exemple>` est une balise non valide.

---

<sup>1</sup> Voir glossaire

## 8.2. La norme VisualDev

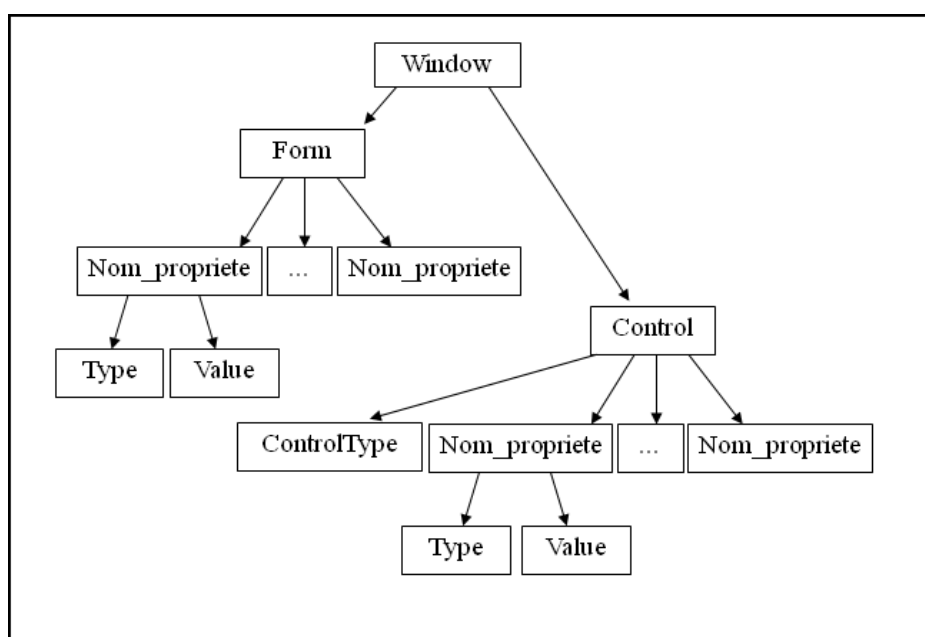
```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<Window>
  <Form>
    <AllowTransparency>
      <Type>Boolean</Type>
      <Value>True</Value>
    </AllowTransparency>
    <AutoScale>
      <Type>Boolean</Type>
      <Value>False</Value>
    </AutoScale>
    <AutoScaleBaseSize>
      <Type>System.Drawing.Size</Type>
      <Value>
        <Width>
          <Type>System.Int32</Type>
          <Value>1032</Value>
        </Width>
        <Height>
          <Type>System.Int32</Type>
          <Value>752</Value>
        </Height>
      </Value>
    </AutoScaleBaseSize>
  </Form>
</Window>
```

*Aperçu d'un fichier projet VisualDev*

La hiérarchie générale de notre norme XML est la suivante. Le nœud racine qui englobe tous les autres est le nœud **Window**. Nous avons choisie de le nommer ainsi car un projet VisualDev est composé d'un seul formulaire, c'est-à-dire d'une fenêtre.

Le premier sous-nœud est donc **Form**. Ce nœud désigne le formulaire de l'interface. Il est unique dans notre document XML car une interface représente une seule fenêtre.

Après le nœud **Form** on trouve les nœuds **Control**. Ces nœuds représentent les différents contrôles composant l'interface créée sous VisualDev (Bouton, TextBox, TreeView,...).

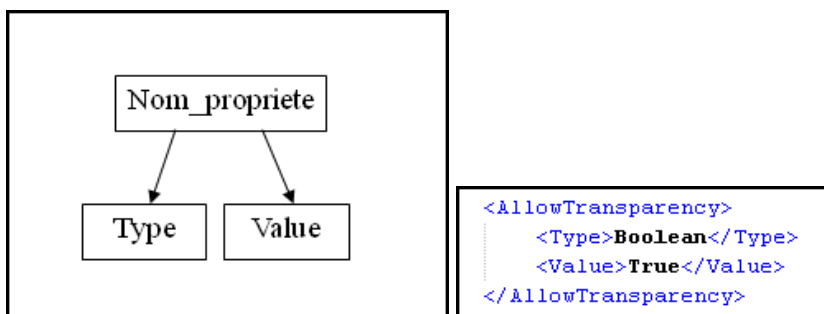


*Hiérarchie générale d'un fichier projet VisualDev*

A l'intérieur des nœuds **Form** et **Control** on trouve les nœuds **Nom\_propriete**. Ces nœuds prennent les noms des différents champs de propriété du formulaire ou du contrôle désigné par le nœud (Cursor, Name, BackColor, FormBorderStyle ...).

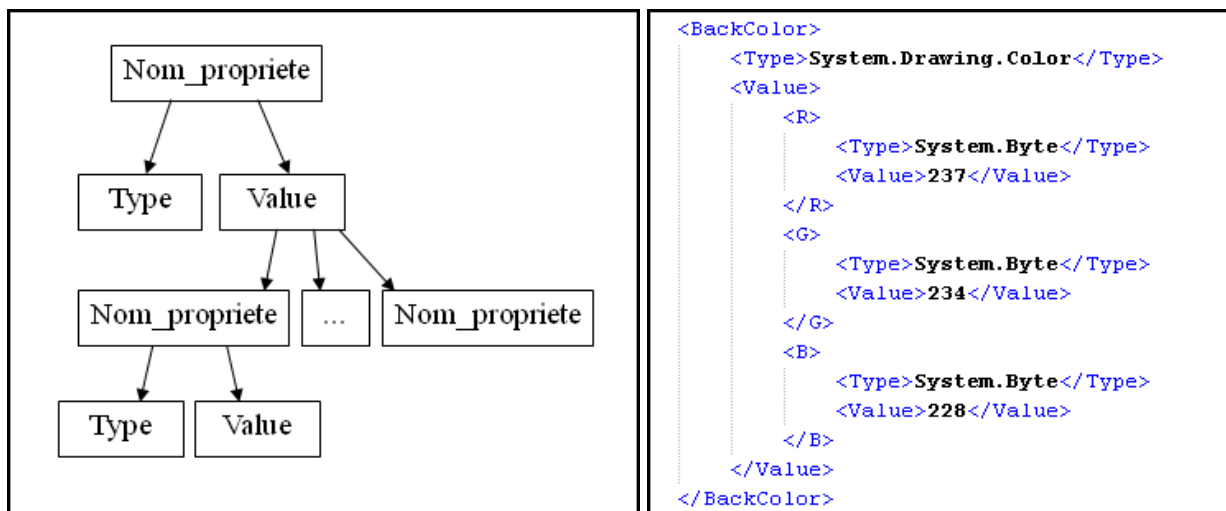
Il existe plusieurs types de propriété à enregistrer :

- Les propriétés simples qui désignent les propriétés de type simple<sup>1</sup> c'est-à-dire les types string, boolean, int, byte,... Ces types sont enregistrés suivant ce schéma :



Cas d'un type simple (exemple un Boolean)

- Les propriétés de type complexe<sup>2</sup>, représentées essentiellement par les types appartenant à la classe System.Drawing. Ces types désignent les couleurs, les polices, les rectangles... Pour pouvoir sauvegarder toutes les informations de ces types nous avons choisi de les décomposer selon le schéma suivant :



Cas d'un type complexe (exemple un System.Drawing.Color)

- Cas particulier de System.Drawing, les icônes et les images. Pour enregistrer les icônes et les images insérées dans notre application, nous avons choisi de les enregistrer en les identifiant par une URL. Cette URL pointe sur une copie du fichier image que le logiciel effectue lors de la sauvegarde du projet.

<sup>1</sup> Voir glossaire

<sup>2</sup> Voir glossaire

```
<Icon>
  <Type>System.Drawing.Icon</Type>
  <IconeURL>./Test_datas/icone.ico</IconeURL>
</Icon>
```

Cas d'un type icône

```
<BackgroundImage>
  <Type>System.Drawing.Bitmap</Type>
  <ImageURL>./Test_datas/image_0</ImageURL>
</BackgroundImage>
```

Cas d'un type image

- Les propriétés désignant des Collections. Lorsque nous rencontrons un type de contrôle qui contient une collection, nous enregistrons le contrôle selon le schéma ci-dessous, c'est-à-dire le schéma de base pour un contrôle avec un nœud supplémentaire intitulé **Donnees**. Ce nœud contient l'URL du fichier binaire contenant les données du contrôle.

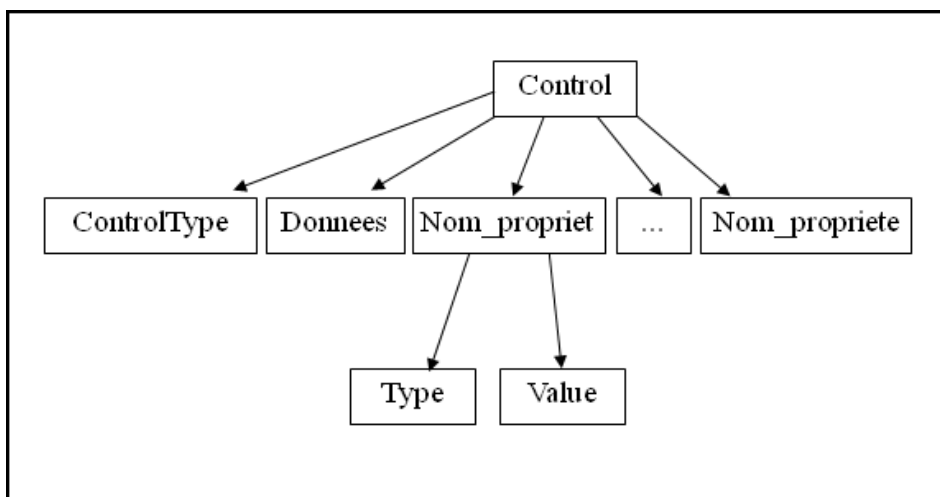


Schéma d'un contrôle contenant des données

```
<Control>
  <ControlType>System.Windows.Forms.TreeView</ControlType>
  <Donnees>./Test_datas/donnee0</Donnees>
  <BackColor>
    <Type>System.Drawing.Color</Type>
  <Value>
```

Cas d'un contrôle contenant des données (par exemple le TreeView)

### 8.3. Conclusion

Nous avons ainsi établi une norme simple qui possède comme intérêt majeur d'être clair et assez explicite. Ainsi si l'utilisateur désire changer le nom (ou toute autre propriété) d'un contrôle (ou du formulaire), il peut le faire directement dans le fichier XML en modifiant le sous-nœud correspondant à la propriété. Lorsque l'interface sera rechargée sous VisualDev la propriété du contrôle sera ainsi modifiée.

# 9. Enregistrement

Une fois l'interface créée sous VisualDev l'utilisateur souhaite généralement la sauvegarder pour une utilisation ultérieure. Pouvoir sauvegarder son interface est assez simple au premier abord, et c'est l'une des fonctionnalités majeure de notre logiciel. L'utilisateur clique sur le bouton sauvegarder. Une fois le chemin renseigné VisualDev va générer un fichier XML décrivant tous les contrôles de l'interface. Un dossier «datas» associé à l'interface va également être créé lors de l'enregistrement. Ce dossier va contenir les images utilisées dans le projet et les données des différents contrôles (données des TreeViews, ComboBox...).

## 9.1. Trouver dynamiquement les propriétés d'un contrôle

La norme XML établi, nous avons cherché un moyen d'enregistrer toutes les informations nous permettant de recréer une interface à partir du fichier XML enregistré. Une première piste fut la sérialisation. Seulement, les objets de type contrôles héritent<sup>1</sup> tous de la classe System.Windows.Form qui est une classe non sérialisable. Autres inconvénients de cette technique, la sérialisation XML comporte énormément de limitation quant aux types que l'on peut sérialiser, du moins avec les fonctions présentes dans le Framework .Net.

Notre seconde piste fut d'utiliser la puissance d'une des notions clés apparus dans le C#, la notion de réflexion. En d'autres termes, la difficulté que nous avons rencontrée pour sauvegarder un objet fut de connaître dynamiquement toutes ses propriétés. Il nous fallait un outil capable de nous donner à partir d'un objet quelconque son type, ses propriétés et ses méthodes (dans le cas des types complexes, par exemple les images).

Ainsi la réflexion est à la source de notre fonction de sauvegarde car elle permet d'obtenir dynamiquement les informations que nous voulions sauvegarder.

## 9.2. Algorithme simplifié de la fonction sauvegarde

- On crée le fichier et le flux XML de destination.
- On crée le dossier datas qui va contenir les données de notre projet (images, icône, ...).
- On charge l'Assembly<sup>2</sup> System.Windows.Form.
- On liste l'ensemble des types présents dans cette Assembly et on ne garde que les types héritant de System.Windows.Form.Control car notre interface n'est composée que d'objets de type contrôles.
- Ensuite, pour chaque propriété du contrôle on appelle la fonction *EnregistrementXmlMember()*. Cette fonction va déterminer s'il faut enregistrer un type simple ou un type complexe et va ajouter les informations nécessaires dans le flux XML.

---

<sup>1</sup> Voir glossaire

<sup>2</sup> Voir glossaire

- Une fois le formulaire et tous les contrôles parcourus on finalise le fichier XML et on ferme le fichier.

### 9.3. Récupérer les propriétés privées d'un contrôle

Lorsque nous avons mis en place cet algorithme, le premier souci que nous avons rencontré fut que nous ne pouvions pas accéder aux membres privés de nos objets. Pour y parvenir, nous avons une nouvelle fois usé de la réflexion en créant une fonction *GetPropertyValue()* qui utilise la méthode *GetProperty()* de la classe *System.Reflection.PropertyInfo*. Cette fonction nous permet de récupérer n'importe quelle donnée, qu'elle soit public, private, static ou autre...

### 9.4. Est-il nécessaire de tout sauvegarder ?

Suite à mise en place de cet algorithme et après quelques tests d'enregistrement, nous nous sommes rendu compte que beaucoup de propriétés ne nécessitaient pas d'être enregistrées. En effet, toutes les propriétés non visuelles ainsi que celles qui ne peuvent pas être modifiées à l'aide de *PropertyGrid* de notre logiciel ne nous intéressaient pas dans la mesure où l'utilisateur ne modifiera jamais ces données non visibles. Nous avons également remarqué que certaines propriétés avaient plusieurs champs ayant les mêmes fonctionnalités. Par exemple : la taille est défini par un champ *height* et *width* ainsi que par un *System.Drawing.Rectangle* possédant les même champs *height* et *width*. Nous avons donc créé une *ArrayList* contenant tous les types qui ne seront pas enregistrés.

### 9.5. Les différents types rencontrés

Notre fonction de sauvegarde doit pouvoir enregistrer tous types d'objets, ce qui signifie qu'elle doit être la plus générale possible. En effet, l'un des objectifs de notre logiciel est de pouvoir insérer des contrôles autres que ceux fournis avec celui-ci. L'utilisateur pourra donc créer des contrôles à l'aide de *VS.Net* puis les insérer dans *VisualDev*.

Néanmoins nous avons été contraints de limiter cette généralisation et de considérer chaque type complexe un par un car pour pouvoir recréer un type *Font* (une police d'écriture) par exemple, nous avons besoin de beaucoup plus d'informations que pour recréer un type simple, par exemple un *Int*. Nous avons donc choisi de décomposer les types complexes. Par exemple une couleur possède trois composantes de type *Byte* représentant les valeurs Rouge, Bleu et Vert de la couleur. Cependant, vu que nous ne travaillons qu'avec des contrôles, les types complexes sont pour la majorité des structures de types simples, ce qui nous a contraint à coder chaque structure une par une.

## 9.6. Enregistrer les images et icônes :

Une fois les types complexes enregistrés nous avons souhaité sauvegarder les images insérées dans l'interface ainsi que l'icône associée au formulaire. Pour y parvenir, nous avons décidé de faire une copie de ces images dans un dossier datas associé au projet en cours. Ainsi, lorsque l'on rencontre un type image nous utilisons la méthode *Save()* présente dans les types *System.Drawing.Image* et *System.Drawing.Icon*. De cette façon, une fois les images insérées dans l'interface, l'utilisateur n'aura plus à s'occuper de celle-ci car elles seront sauvegardées lors de l'enregistrement et resteront ensuite disponibles.

## 9.7. Enregistrer les données par la sérialisation

Dans cette étape nous enregistrons toutes les propriétés des contrôles ainsi que les images et icônes. Il ne restait plus qu'à enregistrer les données contenues dans certains contrôles. Les propriétés de ces contrôles sont des collections qui peuvent être de plusieurs types : collection de *String*, de *Node* (pour les *TreeView*s), de *Dates*, de *Buttons*, de *Panels*...

Bien sûr ces collections ne pouvaient pas rentrer dans la norme XML que nous avons définies et notre algorithme ne fonctionnait plus avec ce type de données. De plus ces données ne représentant pas des éléments graphiques, nous ne savions pas comment procéder pour les sauvegarder. Nous avons donc décidé de traiter au cas par cas et nous nous sommes donné comme objectif d'arriver à enregistrer dans un premier temps les données de types collections de *String* puis ensuite les types plus complexes de type collections de *Node* présentent dans les *TreeView*.

Pour enregistrer ces données de types *String* nous avons utilisé un mécanisme de sérialisation. Pour parvenir à nos fins, nous avons dû ruser car les collections ne sont pas des types sérialisables. Nous avons donc copié ces *Strings* dans une *ArrayList* de *Strings* que nous avons sérialisé à l'aide des fonctions de sérialisation du Framework. Nous avons fait le choix de sérialiser ces données en format binaire pour des raisons de simplicité car les fonctions de sérialisations en format XML possèdent de multiples contraintes qu'ils nous étaient impossibles de dépasser sans l'écriture de dizaines de lignes de codes (à comparer aux 4-5 lignes à écrire dans le cas d'une sérialisation binaire). Pour enregistrer les collections de *Node* nous avons appliqué le même mécanisme que pour les collections de *String* sauf que nous avons créé une *ArrayList* de *Nodes* à la place d'un *ArrayList* de *Strings*.

## 9.8. Fonction de sauvegarde finale

Notre fonction de sauvegarde enregistre donc tous les contrôles et pratiquement toutes leurs propriétés.

Il reste néanmoins quelques collections exotiques comme celles des collections de *Buttons* contenues dans les *Toolbars* que nous n'avons pas réussies à enregistrer car les *Buttons* comme nous l'avons vu précédemment ne sont pas des objets sérialisables, du moins nous n'avons pas encore réussies à le faire.

# 10. Chargement

L'utilisateur, après avoir enregistré son projet va vouloir le recharger. Il va donc falloir réussir à recréer son projet à l'aide de la description de son interface enregistrée dans le fichier XML. Tout comme l'enregistrement, le concept est assez simple. On demande à l'utilisateur le fichier qu'il veut charger, puis on le parcourt pour recréer l'interface en commençant par le formulaire puis chaque contrôle l'un après l'autre.

## 10.1. Parser un fichier XML :

La première difficulté que nous avons rencontrée fut de trouver comment parser<sup>1</sup> notre fichier XML. Cette difficulté fut très vite résolue grâce à notre norme XML assez simple et bien hiérarchisée. De plus, il existe de multiples fonctions dans le Framework qui nous ont aidées à réaliser cette étape.

## 10.2. Créer dynamiquement des contrôles avec leur nom

Le plus compliqué fut de recréer des objets à partir des Strings récupérés dans notre fichier XML. En effet, les informations que nous récupérons dans notre fichier sont des Strings représentant un Button, une Combobox, un TreeView, etc.... Pour parvenir à créer dynamiquement des objets nous avons une nouvelle fois utilisé le procédé très puissant de la réflexion. Nous avons aussi écrit une fonction nous permettant de connaître un type à partir d'une String. Il ne nous restait plus qu'à instancier nos objets avec la fonction *Activator.CreateInstance(Type\_désiré)* contenue dans la classe System.Reflection.

---

<sup>1</sup> Voir glossaire

## 10.3. Leur assigner des propriétés

Nous savions désormais recréer notre interface avec l'ensemble de ses contrôles mais il fallait encore leurs assigner leurs propriétés. Pour ce qui est des types simples, des fonctions de conversions sont disponibles dans la classe `System.Convert` permettant de créer des types `Byte`, `Boolean`, `Int...` à partir de `Strings`. Ces fonctions nous ont été d'une grande utilité.

### 10.3.1. Les propriétés de type simple

Pour assigner les propriétés au contrôle, tout comme lors de l'enregistrement, nous avons créé une fonction `SetPropertyValue()` s'occupant via la réflexion de mettre à jour une propriété donnée d'un contrôle.

### 10.3.2. Les propriétés de type complexe

Pour recharger les propriétés de type complexe nous avons été contraints de traiter chaque type séparément. Ainsi pour recharger un type `System.Cursor` (curseur de la souris) par exemple, nous avons dû traiter tous les types de curseur existant, c'est-à-dire pas moins d'une vingtaine de types différents. Cette méthode est appliquée à l'ensemble des types complexes rencontrés dans les contrôles lors du chargement.

### 10.3.3. Les images et icônes

Recharger les images n'est pas très compliqué car les objets de types `System.Drawing.Image` possèdent des méthodes de chargement d'image. Lorsque l'on charge une image celle-ci est interdite en écriture c'est-à-dire que l'on ne peut pas enregistrer une autre image par-dessus avant la fermeture du programme. Pour pallier ce problème nous créons au démarrage de l'application un dossier temporaire dans le dossier `c:/Windows/Temp/vdev`. Lors du chargement d'une interface, nous faisons une copie des images dans ce dossier temporaire, puis nous chargeons ces copies. Ainsi, lorsque l'utilisateur souhaite sauvegarder de nouveau son application, tout fonctionne parfaitement. Enfin, le dossier temporaire est supprimé à la fermeture de `VisualDev`.

### 10.3.4. Les données

Les dernières informations à recharger sont les données enregistrées en format binaire. Pour les charger il suffit d'appliquer la méthode inverse de la sérialisation soit la désérialisation et d'appliquer l'algorithme inverse que nous avons mis en place pour les sauvegarder.

# 11. Bilan

## 11.1. Bilan du projet

Au regard des objectifs principaux que nous avons définis en début de projet, nous constatons que le résultat est plutôt positif. En effet, ceux-ci ont tous été menés à terme. En ce qui concerne les objectifs optionnels ceux-ci n'ont pas tous été terminés mais sont sur le point de le devenir.

## 11.2. Bilan des connaissances

Tout d'abord ce projet nous a permis de mettre en application les méthodes d'analyses et de conceptions acquises au cours de ces deux années d'IUT. L'UML et la gestion de projet nous ont aidés lors de la phase d'étude ainsi qu'à la définition des tâches à réaliser. L'algorithmie a été utile pour la structuration du code. La rédaction des différents rapports a été facilitée par notre expérience obtenue en cours de préparation au projet tuteuré. La programmation orientée objets était prépondérante car nous avons dû nous servir d'un langage purement objet.

Ce projet a également été bénéfique dans la mesure où il nous a permis d'acquérir de nouvelles connaissances. Nous citerons entre autre l'environnement .Net avec le C#, ainsi que les notions sur le XML.

## 11.3. Bilan humain

Nous avons su démontrer qu'un travail en étroite collaboration à l'intérieur d'un groupe était plus efficace que l'individualisation des tâches. Le point le plus important était sans doute le partage des tâches, tout en gardant une bonne communication au sein du groupe. Nous avons pu entrevoir le travail en équipe, avec tout ce qu'il implique. Nous avons appris à imposer nos idées, mais aussi à nous plier à celles des autres. Toutefois il était nécessaire qu'une personne, en l'occurrence le chef de projet, prenne les décisions importantes. Cette personne permettait de recentrer les objectifs et de garder une certaine cohésion entre les membres du groupe.

## 12. Difficultés rencontrées

Les premières difficultés sont apparues lors de la phase d'analyse du sujet. En effet, nous avons du mal à le cerner. Une fois cette étape passée, nous devions trouver des solutions pour répondre à la problématique. La difficulté fut de trouver des pistes vers lesquelles nous diriger car nous n'avions que peu de documentation et d'exemples permettant de nous aider.

L'optique de notre projet nous a amené à utiliser des notions complexes, difficiles à appréhender et pauvres en aides.

Pour finir, notre méconnaissance du C# au début du projet a ralenti notre immersion dans le développement.

# 13. Bibliographie

## Sites :

<http://www.developpez.com/>

<http://www.csharpfr.com/>

<http://www.wikipedia.org/>

## Documentations :

La documentation MSDN.

## Ouvrage :

Le langage C# de Yannick Lejeune, collection e-Poche, aux éditions Micro-Application.

# 14. Documentation

VisualDev est un logiciel de création graphique. Il permet à tout utilisateur de créer une interface graphique pour ses applications. Il permet également d'enregistrer et d'ouvrir des fichiers possédant une extension .xml qui correspond à la description de l'interface.

Le logiciel a été créé sur le principe de Visual Basic et Visual Studio, se qui ne change pas trop pour un utilisateur confirmé. Pour les autres, l'utilisation est très intuitive. Nous retrouvons des actions classiques : Nouveau, Copier, etc., disponible dans un menu et dans une barre d'outils.

Nous avons décidé pour la finition de notre projet de créer un logiciel d'installation qui permettra de faciliter l'installation de notre application pour les utilisateurs.

## 14.1. Mode d'emploi de l'installation.



Tout d'abord cliquez sur l'icône d'installation  
Une application va se lancer.

- 1.) Sélectionnez la langue d'installation
- 2.) Cliquez sur « suivant » jusqu'à arrivé à « dossier de destination ». Choisissez alors la destination où le logiciel sera installé. Par défaut il sera installé dans C:\Program Files\VisualDev
- 3.) Cliquez sur « suivant ». Vous arrivez alors sur « Sélection du dossier du menu démarrer » choisissez le nom qui sera donné au dossier dans le menu démarré et où vous souhaitez qu'il soit créé. Par défaut il sera créé dans « Programmes » et son nom sera « VisualDev »
- 4.) Cliquez sur « suivant ». Une fois dans « Taches supplémentaires » choisissez si vous souhaitez installer un raccourci sur votre bureau. Par défaut aucun raccourci ne sera créé.
- 5.) Cliquez sur « suivant ». Un récapitulatif vous est proposé.
- 6.) Cliquez sur « suivant ». L'installation démarre.
- 7.) Choisissez ou non d'installer le Framework .Net, il est indispensable à l'utilisation du logiciel.


Attention si vous sélectionné les deux options, l'exécution de VisualDev se fera en même temps que l'installation du Framework .Net. Cela risque de poser des problèmes si le Framework .Net n'est pas encore installé.




Notre application fonctionne au minimum avec la version 1.1 du Framework .Net.

## 14.2. Gérer les projets

### 14.2.1. Créer un nouveau projet

La création d'une nouvelle interface est très simple. Il suffit dans un premier temps de créer un nouveau fichier, pour cela cliquez sur le bouton  (Nouveau) dans la barre d'outils. Vous pouvez également aller dans le menu « Fichier » puis sélectionner « Nouveau ». De plus il existe un raccourci clavier : CTRL + N. Ces trois actions ont pour effet d'ouvrir une nouvelle fenêtre dans la zone de travail.

### 14.2.2. Ouvrir un projet existant



Cliquez sur le bouton  (Ouvrir) dans la barre d'outils, ou allez dans le menu « Fichier » puis sélectionnez « Ouvrir », ou bien encore utilisez le raccourci clavier : CTRL + O. Ces actions ouvrent une fenêtre d'exploration où vous pourrez sélectionner l'emplacement du fichier à ouvrir. Le fichier sera alors chargé et le projet contenu dans celui-ci sera affiché dans la zone de travail.

### 14.2.3. Sélection du projet actif

Vous pouvez sélectionner un projet ouvert grâce à la barre qui se situe en haut de la zone de travail. En cliquant dessus, la liste des projets s'affiche. Cliquez alors sur le nom du projet que vous désirez ramener au premier plan.

### 14.2.4. Enregistrer un projet

Vous pouvez enregistrer un projet en allant dans le menu « Fichier » puis « Enregistrer sous ». Une fenêtre d'exploration s'ouvrira et il ne vous restera qu'à sélectionner l'emplacement de la sauvegarde ainsi que le nom du fichier.

Si le projet a déjà été enregistré, cliquez sur le bouton  (Enregistrer) ou  (Enregistrer tout), qui se situent dans la barre d'outil ou bien allez dans le menu « Fichier » puis « Enregistrer » ou encore avec le raccourci clavier : CTRL + S. Un message s'affichera à l'écran pour informer l'utilisateur du résultat de l'enregistrement.


### 14.2.5. Fermer un projet

Il y a plusieurs solutions pour fermer le projet en cours de création, en utilisant la croix en haut à droite de la fenêtre du projet, ou bien en allant dans le menu « Fichier » puis en sélectionnant « Fermer ». Si vous fermez un projet qui a été modifié ou qui n'a jamais été enregistré, alors l'application vous demandera si vous désirez l'enregistrer. C'est aussi le cas lorsque vous quittez l'application, dans ce cas chaque projet en cours sera traité.

## 14.3. Modifier la liste des contrôles


VisualDev possède une liste de contrôles par défaut. Mais il est possible d'ajouter ou de retirer des contrôles. Pour cela cliquez sur le bouton en bas de la liste des contrôles appelé « Modifier la liste ». Une fenêtre s'ouvre alors. A gauche, il y a la liste des contrôles déjà présents et à droite la liste des contrôles disponibles.

Vous pouvez ajouter un, plusieurs ou tous les contrôles.


Validez les modifications en cliquant sur 

Annulez les modifications en cliquant sur 


### 14.3.1. Ajouter un contrôle

Dans la liste de droite, cliquez sur le contrôle voulu puis sur le bouton .

### 14.3.2. Ajouter plusieurs contrôles



Sélectionnez dans la liste de droite tous les contrôles voulus et cliquez sur le bouton .

### 14.3.3. Ajouter tous les contrôles

Il suffit de cliquer sur le bouton , ainsi tous les contrôles seront ajoutés dans la liste de gauche.

→ Les contrôles déplacés vont alors apparaître dans la liste de gauche.

### 14.3.4. Retirer un ou plusieurs contrôles

C'est la même démarche que l'ajout, mais contrairement à lui il faut cliquer sur les boutons  et  pour déplacer les éléments de la liste de gauche dans la liste de droite.

### 14.3.5. Ajouter un contrôle utilisateur

Vous pouvez ajouter d'autres contrôles, pour cela cliquez sur le bouton en bas à droite. Une fenêtre s'ouvre alors permettant de sélectionner la librairie (.DLL) qui contient les contrôles à ajouter. Le ou les nouveaux contrôles vont alors apparaître dans la liste de droite il ne restera plus qu'à les déplacer dans la liste de gauche.

## 14.4. Gérer les contrôles


### 14.4.1. Insérer un contrôle dans le projet

Dans la liste des contrôles, sélectionnez celui que vous voulez insérer, puis cliquez à l'endroit où vous voulez l'insérer. Un clic simple donne au contrôle sa taille par défaut, mais en restant cliqué vous pouvez lui donner la taille que vous voulez. Une fois la taille désirée atteinte, relâchez le bouton de la souris.

### 14.4.2. Modifier les propriétés des contrôles

Cliquez sur le ou les contrôles à modifier. Leurs propriétés apparaissent sur la droite de l'écran. Vous pouvez désormais les modifier à volonté.




### 14.4.3. Déplacer un contrôle

Pour déplacer un contrôle, cliquez dessus. Il est alors encadré. Le contrôle est dit « sélectionné ». Déplacez alors le curseur de votre souris sur le cadre. Le curseur de la souris affiche . Vous pouvez alors cliquer sur le cadre et déplacer le contrôle. Relâchez le bouton de la souris pour terminer le déplacement.

### 14.4.4. Redimensionner un contrôle

Pour redimensionner un contrôle, sélectionnez-le. Déplacez alors le curseur de votre souris sur les points de couleur verte sur le cadre. Le curseur de la souris indique les directions dans lesquelles vous pouvez redimensionner. Cliquez sur le point approprié et déplacez la souris. Relâchez le bouton de la souris pour terminer le redimensionnement.

### 14.4.5. Copier, Coller, Couper

Il est possible de copier ou couper un contrôle. Sélectionnez le contrôle que vous cherchez à dupliquer, cliquez ensuite sur  (Copier) pour copier le contrôle ou sur  (Couper) pour le couper (c'est-à-dire que le contrôle sera supprimé après avoir été copié). Cliquez alors sur  (Coller) pour ajouter le contrôle précédemment copié.

Ces actions sont également disponibles dans le menu « Edition ».

Des raccourcis clavier existent pour ces différentes actions :

- Copier           CTRL+C
- Coller           CTRL+V
- Couper           CTRL+X

### 14.4.6. Supprimer un contrôle

Faites un clic droit sur le contrôle à supprimer puis sélectionnez « Supprimer », ou alors une fois le contrôle sélectionné appuyez sur la touche SUPPR ou DEL de votre clavier.

### 14.4.7. Ajuster les contrôles

Il est possible d'ajuster la taille des contrôles, soit par rapport à la grille, soit par rapport à un autre contrôle.

Si vous voulez ajuster la taille de votre contrôle par rapport à la grille, faites un clic droit sur le contrôle et sélectionnez « Ajustement ➤ » puis « Ajuster la taille à la grille ».

Pour ajuster des contrôles entre eux, il suffit de sélectionner les contrôles souhaités et de faire un clic droit sur le contrôle dont on souhaite utiliser la taille comme référence.

## **14.4.8. Aligner les contrôles**

Il est possible d'aligner les contrôles de différentes façons.

### **14.4.8.1. Alignement par rapport à la grille**

Faites un clic droit sur le contrôle que vous souhaitez aligner et sélectionnez « Alignement► » puis « Aligner sur la grille ». Le contrôle sera alors aligné automatiquement sur la grille.

Vous pouvez faire la même chose pour plusieurs contrôles à la fois. Pour cela sélectionnez d'abord tous les contrôles que vous voulez aligner sur la grille.

### **14.4.8.2. Alignement horizontal**

Si vous voulez aligner plusieurs contrôles sur une ligne horizontale, commencez par sélectionner tous les contrôles. Ensuite faites un clic droit sur le contrôle dont vous souhaitez utiliser la position comme référence, puis sélectionnez « Alignement► » et « Aligner horizontalement sur ce contrôle ».

### **14.4.8.3. Alignement vertical**

Si vous voulez aligner plusieurs contrôles sur une ligne verticale, commencez par sélectionner tous les contrôles. Ensuite faites un clic droit sur le contrôle dont vous souhaitez utiliser la position comme référence, puis sélectionnez « Alignement► » et « Aligner verticalement sur ce contrôle ».

## **14.4.9. Mettre au premier plan ou en arrière-plan**

On peut mettre les contrôles soient en premier plan soit en arrière-plan. Si le contrôle est en arrière-plan alors les autres contrôles seront affichés par-dessus lui. Pour mettre un contrôle au premier plan ou en arrière-plan, faites un clic droit sur le contrôle, puis dans « Disposition► » choisissez le mode « Mettre au premier plan » ou « Mettre en arrière-plan ».

Par défaut, le dernier contrôle inséré est passé au premier plan.

## **14.5. Autres**

### **14.5.1. Afficher la grille**

Il est possible d'affiche la grille de positionnement des contrôles, pour cela allez dans le menu « Option » puis cliquez sur « Afficher la grille ». Pour ne plus l'afficher répétez la même opération.

### **14.5.2. Quitter le programme**

Pour quitter le programme, il y a deux solutions, en utilisant la croix en haut à droite, ou bien en allant dans le menu « Fichier » puis en sélectionnant « Quitter ».

# 15. Glossaire

## **Accesseur**

Méthode permettant d'accéder à une propriété d'un champ privé.

## **Assembly**

Voir Librairie dynamique (DLL).

## **Classe**

Voir objet.

## **Composant visuel**

Voir Contrôle.

## **Contrôle**

C'est un objet visuel, comme une zone de texte, une case à cocher ou un bouton de commande, qui permet aux utilisateurs d'entrer ou d'afficher des données, d'afficher des choix de données ou d'exécuter une action.

## **Contrôle utilisateur**

C'est un contrôle créé par l'utilisateur, généralement propre à une application, bien que parfois réutilisable.

## **Drag and drop**

Le drag 'n' drop, littéralement Glisser-Déposer en langue anglaise, désigne une manipulation effectuée à l'aide de la souris, consistant à déplacer un objet virtuel (icône, élément graphique, widget, etc.) d'un point vers un autre sur le plan de l'écran, ou d'une zone contenue dans ce dernier.

## **Environnement .Net**

Voir Framework .Net.

## **Événement**

C'est une action, comme un clic de souris ou l'enfoncement d'une touche, qui est reconnue par un objet et pour laquelle vous pouvez définir une réponse. Un événement peut être déclenché par une action utilisateur, par le code d'un script ou d'un programme, ou par le système d'exploitation.

## **Flux**

Le flux est un ensemble de données structurées se déplaçant du fichier jusqu'à l'application, où il sera attrapé puis analysé.

## **Formulaire (Form)**

Le formulaire représente la fenêtre contenant l'ensemble des composants visuels.

## **Formulaires MDI**

Formulaire inclus dans d'autre formulaire. Voir Formulaire.

## **Framework**

C'est un ensemble de bibliothèques permettant le développement rapide d'applications. Ils fournissent suffisamment de briques logicielles pour pouvoir produire une application aboutie. Ces composants sont organisés pour être utilisés en interaction les uns avec les autres.

## **Héritage**

C'est un mécanisme qui diffuse les caractéristiques d'un objet père vers un objet fils. Un objet fils hérite donc des propriétés et méthodes d'un objet père. L'objet fils peut posséder des caractères supplémentaires par rapport au père.

## **IHM**

(Interface Homme – Machine) : Ce qui est visible par l'utilisateur dans une application.

## **Implémenter**

Réaliser la phase finale d'élaboration d'un système qui permet au matériel, aux logiciels et aux procédures d'entrer en fonction.

## **Instancier**

Concept clé de la programmation objet. Opération qui consiste à définir un programme à partir d'un modèle, plus précisément la classe d'objets à laquelle il appartient, à fixer les valeurs des éléments variables et à exécuter le tout. En d'autres termes, une instanciation revient à créer une copie exécutable du modèle.

## **Interface graphique**

Voir IHM.

## **Librairie dynamique (DLL)**

(Dynamic Link Library) : bibliothèque de fonctions chargées dynamiquement à l'exécution. En informatique, on nomme bibliothèque logicielle un ensemble de routines regroupées pour réaliser un groupe de tâches du même domaine. Les bibliothèques logicielles se distinguent des exécutables dans la mesure où elles sont utilisées par des programmes plutôt que d'être exécutées directement elles-mêmes; elles fournissent un code « assistant » un programme indépendant en lui fournissant des services (par exemple le calcul d'un cosinus, ou l'inversion d'une matrice).

## **Métalangage**

Langage permettant de définir un autre langage (souvent lui-même en tout premier lieu).

## **Méthode**

C'est une procédure similaire à une commande ou une fonction opérant sur des objets spécifiques. Une méthode est une fonction membre d'un objet.

## **Mutateur**

Méthode permettant de modifier une propriété d'un champ privé.

## **Objet**

Un objet peut être une personne, une couleur, un tableau, une chaîne, une date ou encore un objet créé de toute pièce par le concepteur. Concrètement un objet se caractérise par un ensemble de méthodes et de propriétés.

## **Parser**

Récupérer les informations contenues dans les balises d'un document XML.

## **Réflexion**

Voir chapitre 5.1 – Réflexion.

## **Sérialisation**

Voir chapitre 5.2 – Sérialisation.

## **Type**

Le type d'une expression ou d'une variable indique le domaine des valeurs qu'elle peut prendre et les opérations qu'on peut lui appliquer.

## **Type complexe**

Un type complexe sera assimilé à un objet.

## **Type simple**

On considérera qu'un type simple est :

- Soit un nombre entier : int
- Soit une chaîne de caractères : string
- Soit un booléen : bool, qui prend les valeurs 0 ou 1 (true ou false)
- Soit un octet : byte

## **UML**

(Unified Modeling Language/Langage unifié pour la modélisation) : c'est un langage graphique qui permet de représenter de manière claire et précise, sous forme de modèle objet, d'une part la structure et le comportement des processus métiers de l'entreprise, d'autre part des applications ou des composants logiciels. En tant que tel, il facilite la création et la compréhension des logiciels actuels.

## **XML**

Voir chapitre 8.1 – Notion sur le XML.