Developpez.com
Club des Développeurs

Démarrer avec les MFC avec Visual studio 2005 Volume 2

Par Farscape

Date de publication : 8 Novembre 2006

Dernière mise à jour : 4 Novembre 2006

Sources :http://farscape.developpez.com/Samples/SampleMDI-1.zip
http://farscape.developpez.com/Samples/SampleMDI-2.zip

Ce tutoriel fait suite au document du même nom déjà publié, contrairement au premier volume j'utiliserai la plateforme de développement **Visual studio 2005** Pour continuer notre apprentissage du cadre de travail (FrameWork) des **MFC**.

I. Autres Modèles d'application ... 4
Une application boîte de dialogue : ... 4
Génération du projet : ... 4
Conclusions : ... 8
II. Le modèle d'application MDI (multiple document interface) : ... 9
II-A. Génération du projet : ... 10
II-A-1. Mode de fonctionnement du modèle MDI : ... 15
II-A-2.Quelques remarques : ... 16
II-B. Etude des éléments composants le framework ... 17
II-B-1. La classe CMDIFrameWnd ... 17
II-B-2. La classe CMDIChildWnd ... 18
II-B-2-a. Exemples : ... 18
II-B-2-b. Quelques fonctions utiles : ... 18
II-B-3. Séquences de création des objets dans une architecture MDI : ... 19
II-B-3-a. Séquences sur la création d'un document ... 19
II-B-3-b. Séquences sur la création d'une fenêtre cadre MDI et sa vue ... 20
II-B-3-c. Séquences d'initialisation d'une vue ... 22
II-B-4. Routages des messages : les classes du frame work : ... 23
II-C. Travailler avec plusieurs vues dans le modèle MDI ... 24
II-C-1. Les modifications avec l'éditeur de ressources dans le détail : ... 24
II-C-2. Quelques remarques générales sur l'environnement de développement ... 27
II-C-2-a. En ce qui concerne la modification des propriétés des contrôles : ... 27
II-C-2-b. Différences sur la génération de code avec Visual 6.: ... 28
II-C-2-c. Génération de la classe CFormView associée : ... 28
II-C-3. Paramétrage du document Template dans InitInstance ... 30
II-C-3-a. Notes : ... 30
II-C-4. Exécutons notre application : ... 31
II-C-5. Détails des modifications : ... 33
II-C-6. Conclusions : ... 35
II-D. Association de plusieurs vues sur un même document : ... 36
II-D-1. Introduction : ... 36
II-D-2. Mise en place : ... 36
II-D-2-a. Les étapes du traitement : ... 37
II-D-2-b. Le résultat : ... 38
II-D-3. Amélioration du traitement : ... 39
II-D-4.Création d'une nouvelle fenêtre : ... 43
II-D-4-a. Quelques remarques: ... 43
II-D-4-b. Conclusions : ... 43
II-E-1. Validité d'un objet : ... 45
II-E-2. Recherche d'une fenêtre ... 46
II-E-2-a. Fenêtres faisant parties du même document : ... 46
II-E-2-b. Cas de fenêtres issues de plusieurs documents : ... 50
II-E-2-b-1. Récuperer le cadre MDI ou la fenêtre active de l'application : ... 51
II-F. Mise en place d'une barre d'outils dans un cadre MDI. ... 52

II-F-1. Génération de la barre d’outils : 52
II-F-2. Modification de la classe générée : 54
II-F-3. Ajouter des fonctions de réponses à l’action des boutons : 56
II-F-3-a. Insertion du menu pour correspondance avec la barre d’outils : 57
II-F-4. Gestion de l’activité du bouton : 59
II-F-5. Personnaliser la barre d’outils : 60
II-F-5-1. Insérer des boutons dynamiquement dans la barre d’outils : 62
II-F-5-1-a. Utilisation : 66
II-F-5-1-b. Initialisations dans la classe CChildSampleView: 67
II-F-5-1-c. Le résultat : 68
II-F-5-2. Barre d’outils flottantes et positionnement : 68
II-F-5-3. Boutons à états : 70
II-F-5-3-a. Le résultat : 72
II-G. Mise en place d’une barre de dialogue dans un cadre MDI. 73
II-G-1. Problèmes typiques avec la barre de dialogue 75
II-H. Mise en place d’une barre de séparation dans un cadre MDI. (splitter). 80
III. Simplifier l’ajout d’éléments dans le cadre de travail (Framework). 85
III-A. Spécifier les éléments du cadre fille directement dans InitInstance 85
III-A-1. Détails du code 86
III-A-2. En Résumé : 92
III-A-3. Résultat final : 93
Remerciements 94

Le FrameWork deuxième partie

I. Autres Modèles d'application

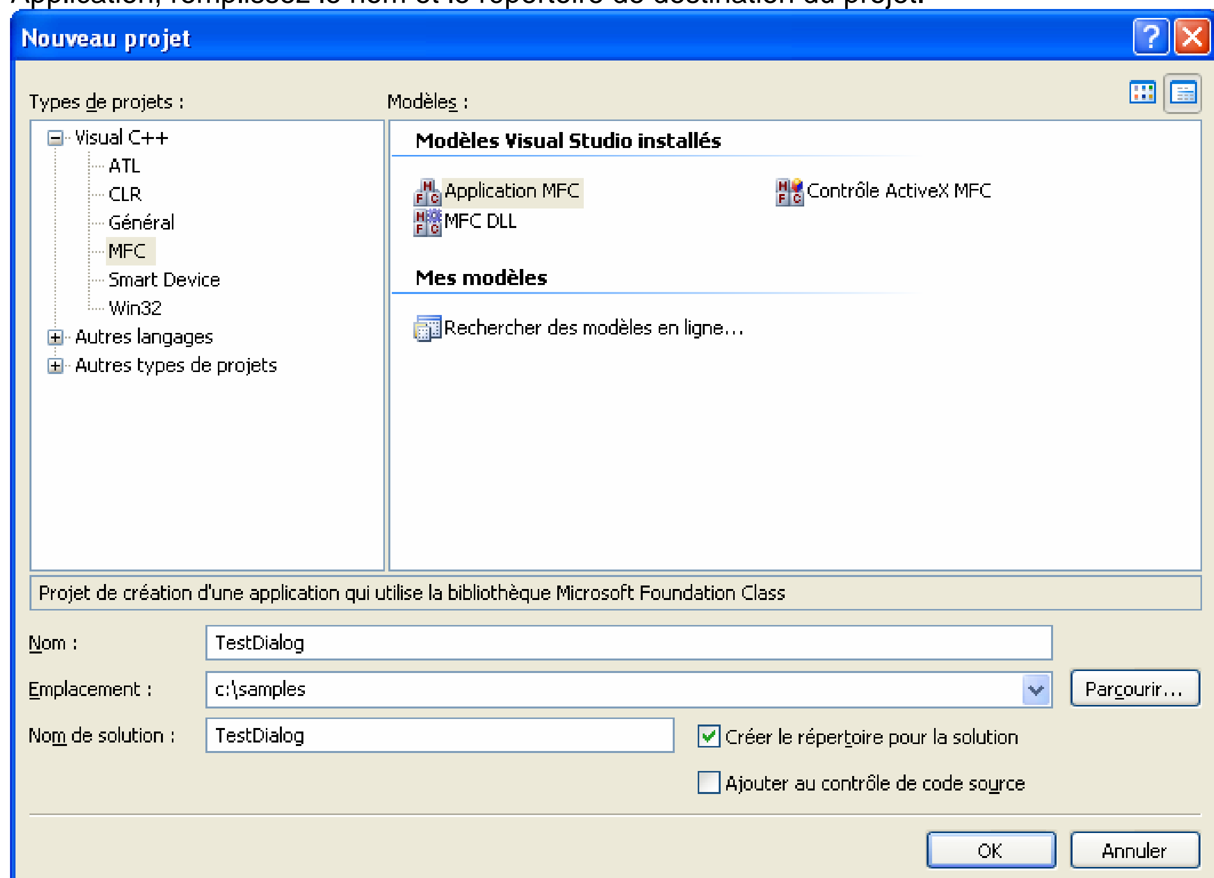
Parmi les modèles d'applications existants il nous reste à voir le modèle de projet boîte de dialogue et le modèle MDI.

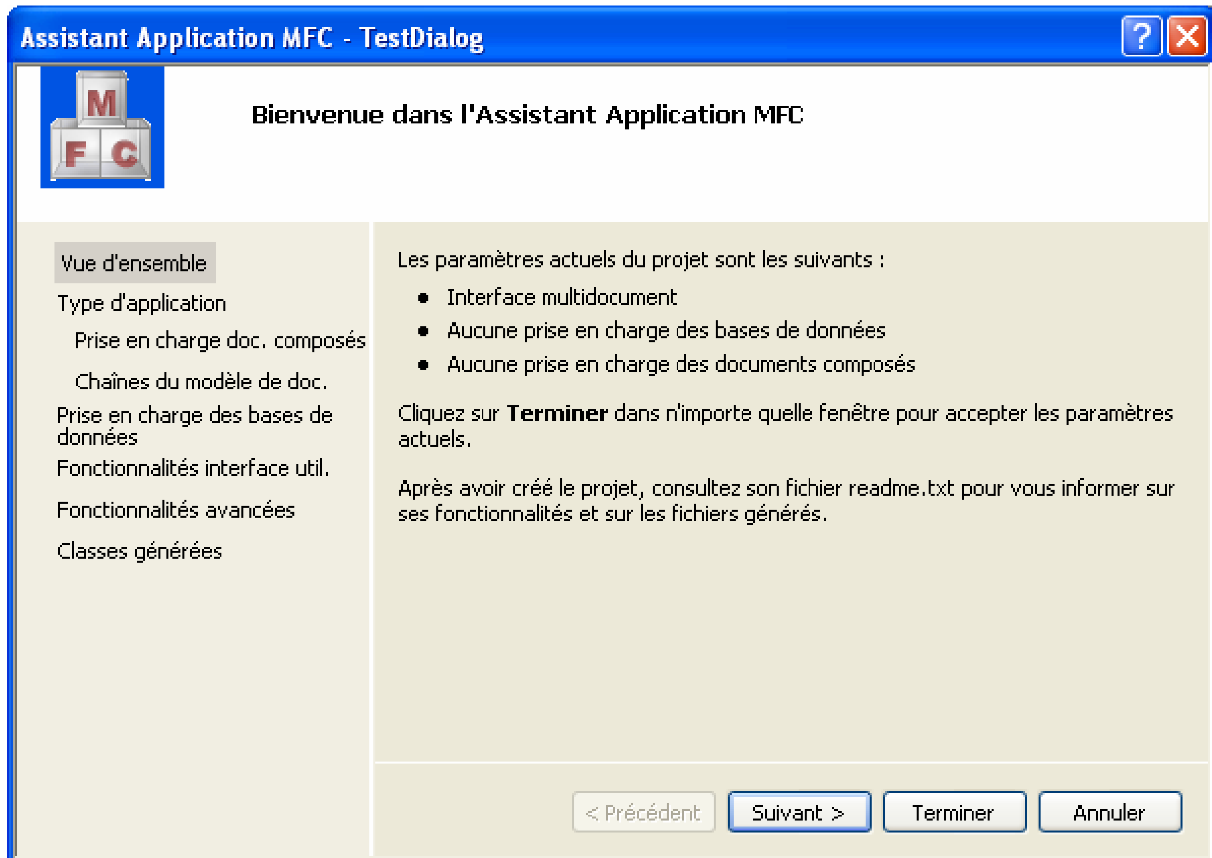
Une application boîte de dialogue :

Ce type d'application ne comportera qu'une seule fenêtre, dans ces conditions on peut se poser la question sur la différence avec un projet SDI.
Voici quelques éléments de réflexions pour mieux cerner les différences :
Une boîte de dialogue :

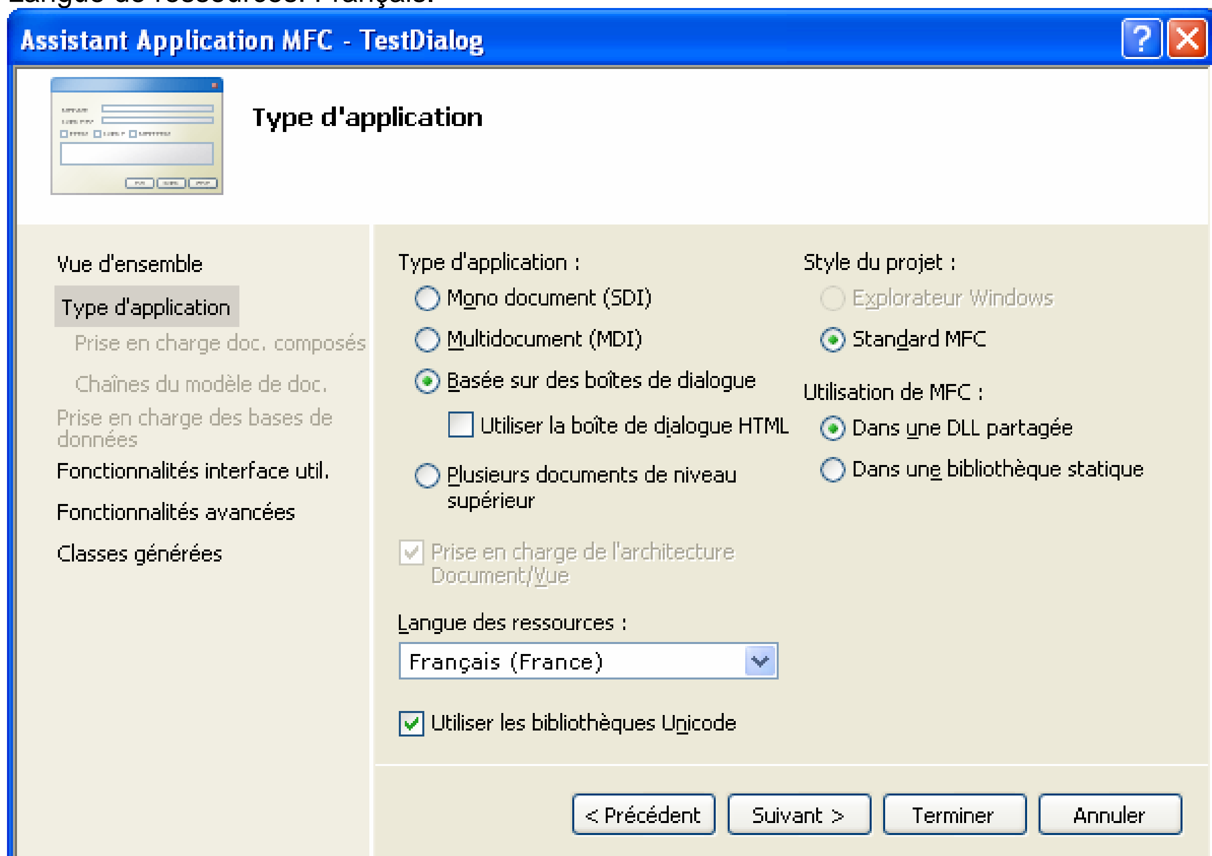
- N'utilise pas l'architecture document vue (ça tombe presque sous le sens !).
- Ne permet pas en standard la gestion d'une barre d'outils.
- Bien que la boite de dialogue puisse être redimensionnée par l'utilisateur celle-ci ne gérera pas d'ascenseurs sur la fenêtre.
- En cas de changement d'architecture, exemple : l'application doit utiliser plusieurs vues et donc passer dans le modèle de projet MDI, les modifications du code seront plus contraignantes que pour un modèle SDI.

Génération du projet :
Procédons maintenant à la génération du projet à l'aide du gestionnaire de projet :
Avec Visual 2005 sélectionnez le menu File **new project** sélectionnez le template MFC Application, remplissez le nom et le répertoire de destination du projet.

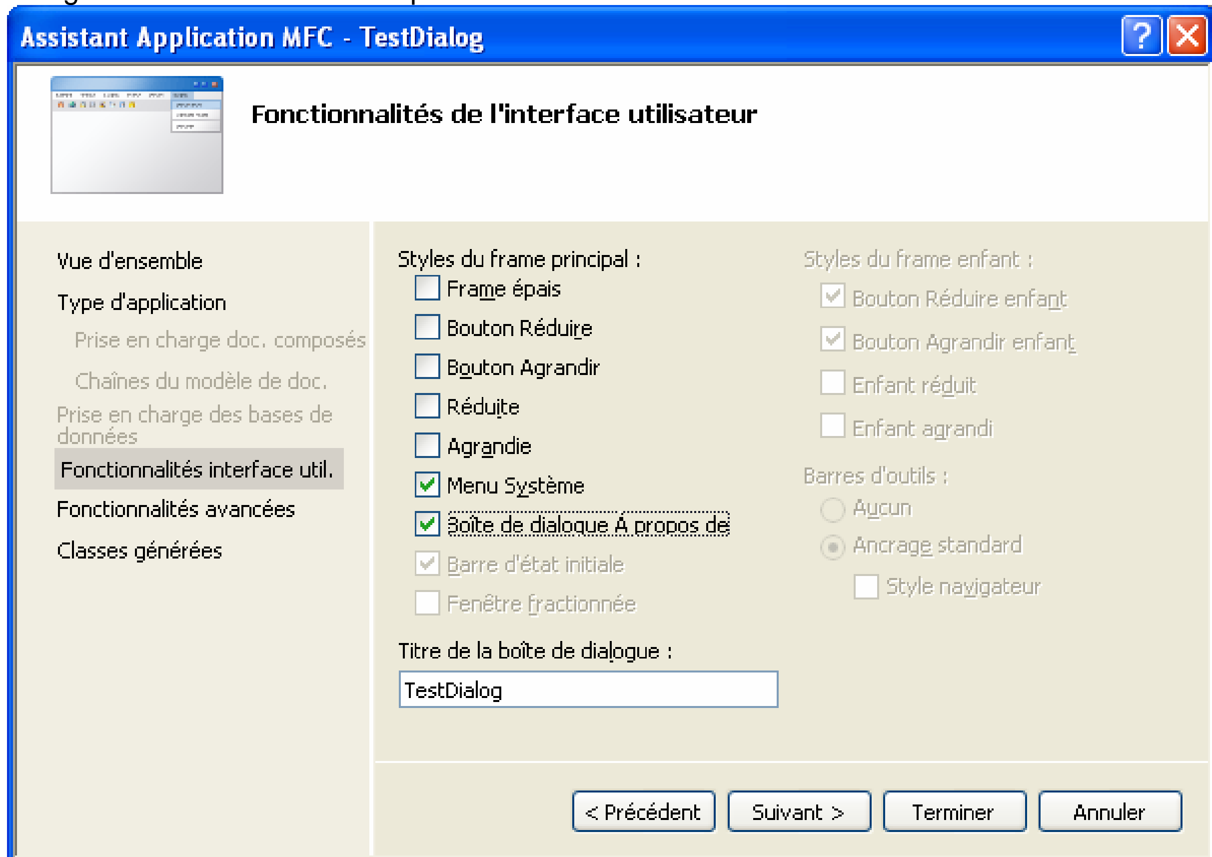




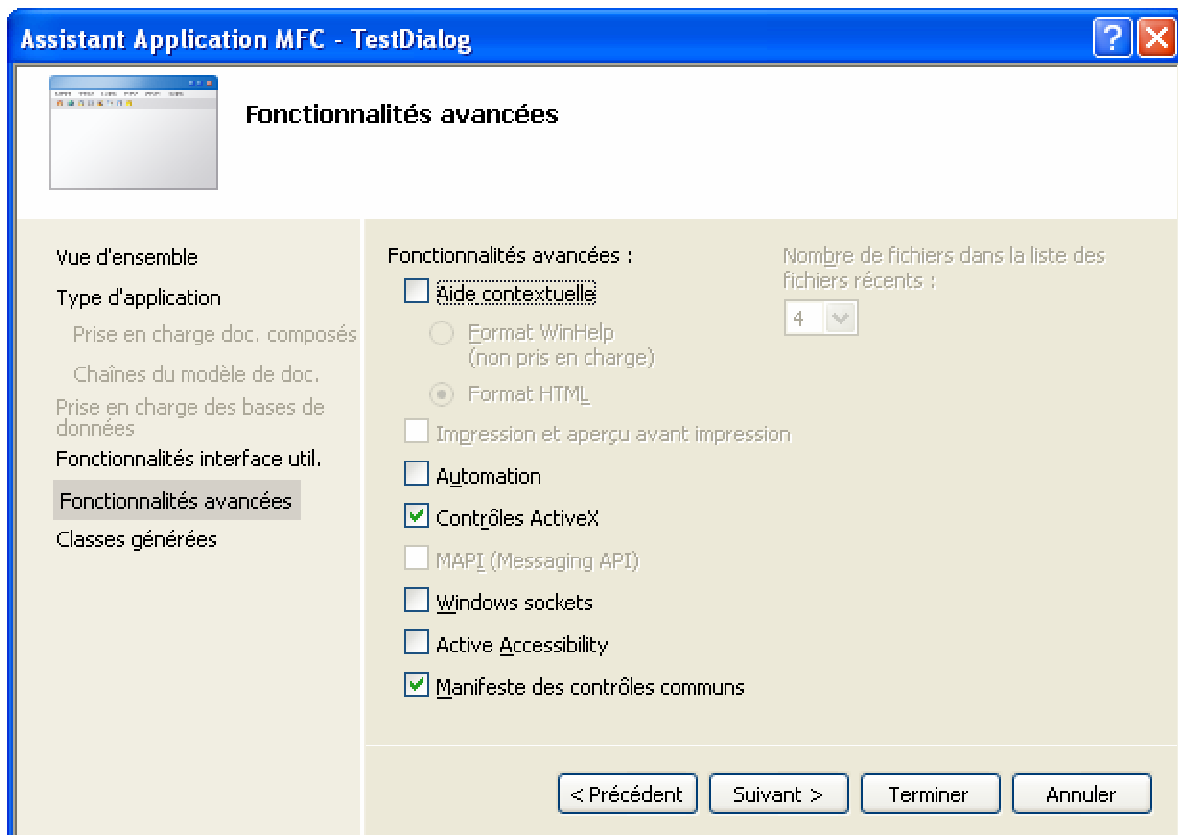
Sélectionnez la ligne Type d'application :
Puis les options :
Basée sur des boîtes de dialogue
Langue de ressources: Français.

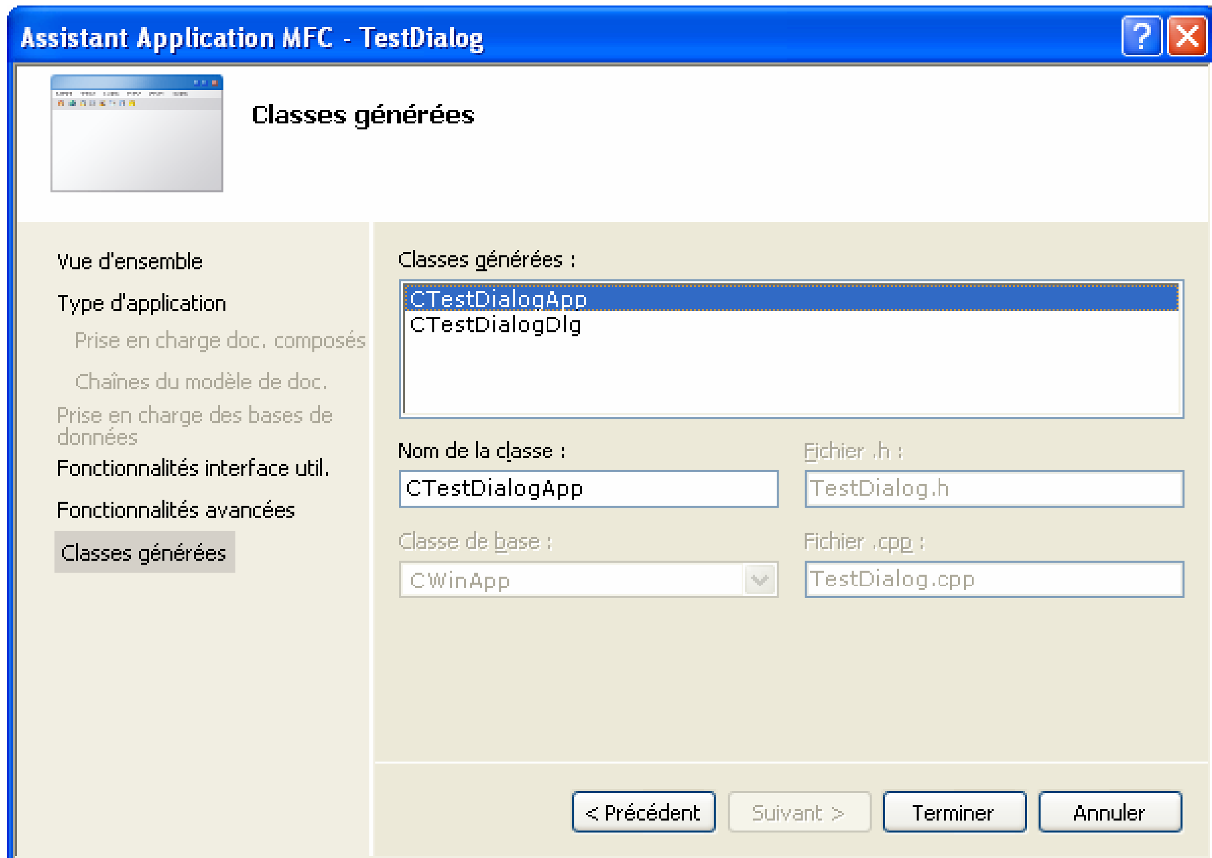


L'onglet suivant concerne les options de l'interface utilisateur :



Sur cet onglet on réglera des options comme le support des ActiveX, des sockets, de l'aide contextuelle.





Examinons maintenant le code généré :

```
// InitCommonControlsEx() est requis sur Windows XP si le manifeste de
l'application
	// spécifie l'utilisation de ComCtl32.dll version 6 ou ultérieure
pour activer les
	// styles visuels.  Dans le cas contraire, la création de fenêtres
échouera.
	INITCOMMONCONTROLSEX InitCtrls;
	InitCtrls.dwSize = sizeof(InitCtrls);
	// À définir pour inclure toutes les classes de contrôles communs à
utiliser
	// dans votre application.
	InitCtrls.dwICC = ICC_WIN95_CLASSES;
	InitCommonControlsEx(&InitCtrls);

	CWinApp::InitInstance();

	AfxEnableControlContainer();

	// Initialisation standard
	// Si vous n'utilisez pas ces fonctionnalités et que vous souhaitez
réduire la taille
	// de votre exécutable final, vous devez supprimer ci-dessous
	// les routines d'initialisation spécifiques dont vous n'avez pas
besoin.
	// Changez la clé de Registre sous laquelle nos paramètres sont
enregistrés
	// TODO : modifiez cette chaîne avec des informations appropriées,
	// telles que le nom de votre société ou organisation
	SetRegistryKey(_T("Applications locales générées par AppWizard"));
```

```
	CTestDialogDlg dlg;
	m_pMainWnd = &dlg;
	INT_PTR nResponse = dlg.DoModal();
	if (nResponse == IDOK)
	{
		// TODO : placez ici le code définissant le comportement 
lorsque la boîte de dialogue est
		//  fermée avec OK
	}
	else if (nResponse == IDCANCEL)
	{
		// TODO : placez ici le code définissant le comportement 
lorsque la boîte de dialogue est
		//  fermée avec Annuler
	}

	// Lorsque la boîte de dialogue est fermée, retourner FALSE afin de 
quitter
	//  l'application, plutôt que de démarrer la pompe de messages de 
l'application.
	return FALSE;
```

La fonction **Initinstance** de la classe d'application met en place la boîte de dialogue et affecte l'adresse de la fenêtre au pointeur sur la mainframe (**m_pMainWnd**).
Première conséquence :**AfxGetMainWnd()** renverra forcement l'adresse de la boîte de dialogue.
Ensuite comme avec une boîte de dialogue classique la fonction **DoModal** est appelée
Sur la fermeture de la boîte de dialogue on dispose du choix de sortie de l'utilisateur :
Entrée ou échappement.
Note : après le **DoModal** le pointeur **m_pMainWnd** est nettoyé et n'est plus valide.

En ce qui concerne la classe générée pour la boite de dialogue, le générateur de projet mettra en place le code nécessaire à la lecture du menu de l'application et la gestion de la réduction de l'application.
On trouvera respectivement ces ajouts dans la fonction **OnInitDialog** et **OnPaint** de la boîte de dialogue.

Conclusions :
Une boîte de dialogue permet de développer très vite une petite application avec une interface minimaliste pouvant convenir pour de petits programmes utilitaires.
En cas de changement de modèle de fonctionnement de l'application en SDI ou MDI, les adaptations seront plus difficiles à négocier et il faudra certainement reconstruire un nouveau projet.

II. Le modèle d'application MDI (multiple document interface) :

C'est le modèle d'application le plus complet avec les MFC, il permet d'avoir simultanément plusieurs vues différentes dans la fenêtre principale de l'application.
C'est aussi le modèle le plus compliqué à mettre en œuvre.

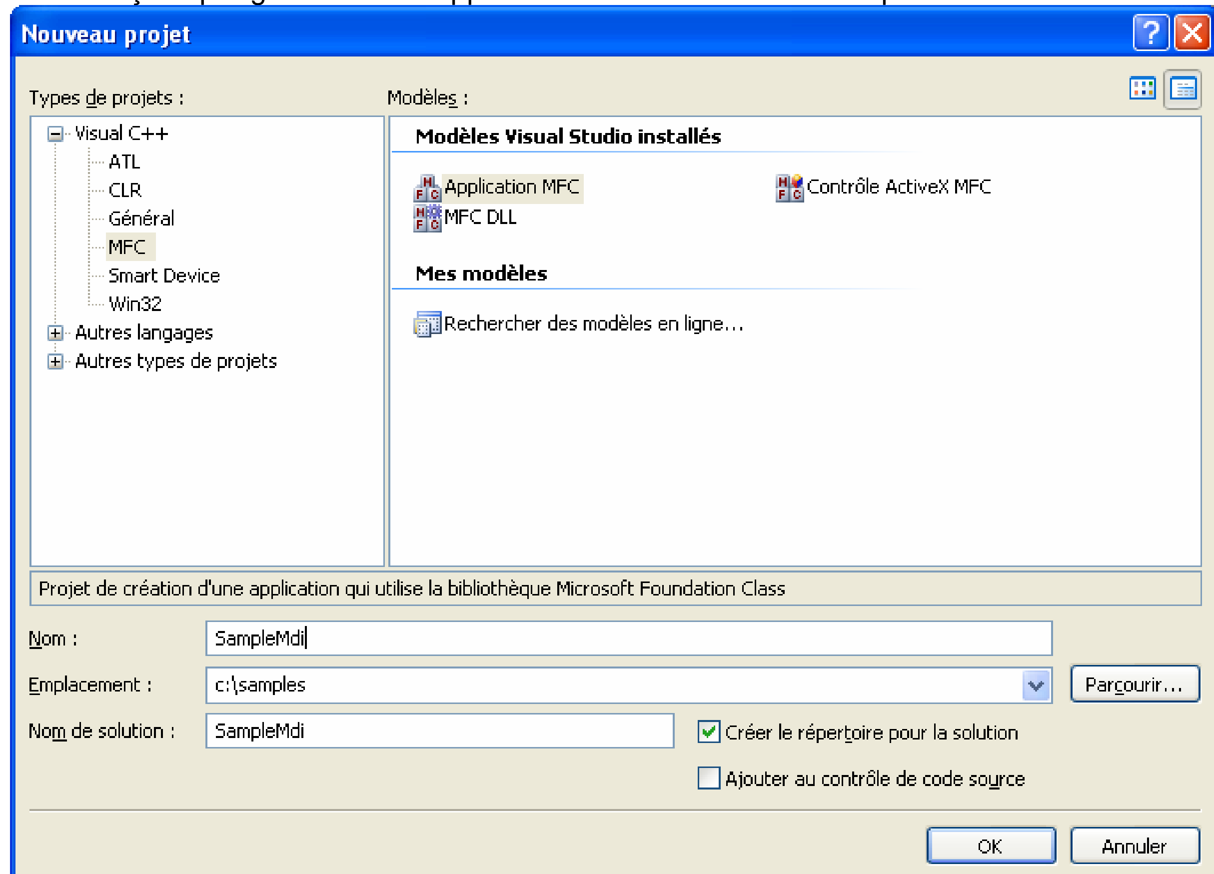
Comme je l'avais évoqué précédemment ce qui est fastidieux à mettre en œuvre avec les MFC se sont tous les enrichissements d'interfaces graphiques que l'on peut apporter à une vue,
Classiquement le programmeur MFC pourra être confronté aux problèmes d'architectures suivants :

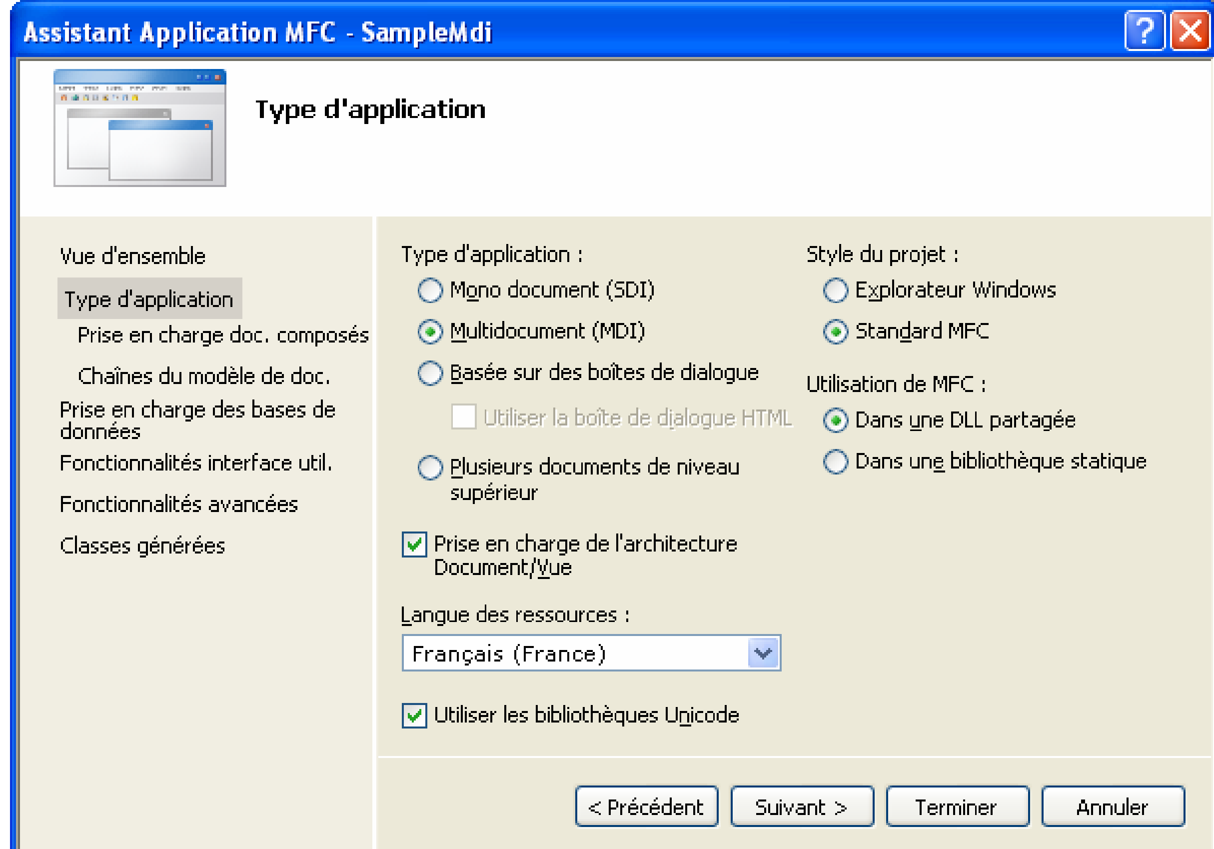
La mise en place de plusieurs vues dans une application.
La communication entre plusieurs vues.
La mise en place d'une barre d'outils
La mise en place d'une barre de dialogue et sa communication avec la vue associée.
La mise en place d'un rideau de séparation (splitter).

Je me propose dans les pages qui suivent de vous expliquer l'architecture MDI et ses composants, puis au fur et à mesure des problèmes rencontrés, de vous exposer des solutions pour améliorer la génération construite par défaut.
Pré-requis :
Le lecteur pour suivre ce qui va être développé, devra avoir déjà pratiqué les MFC en lisant par exemple mon précédent tutoriel sur le même sujet...
Et avoir un niveau correct en C++.

II-A. Génération du projet :

Commençons par générer notre application MDI en suivant les étapes ci-dessous :





Assistant Application MFC - SampleMdi

Chaînes du modèle de document

Vue d'ensemble
Type d'application
Prise en charge doc. composés
Chaînes du modèle de doc.
Prise en charge des bases de données
Fonctionnalités interface util.
Fonctionnalités avancées
Classes générées

Chaînes non localisées

Extension de fichier : SMDI
ID du type de fichier : SampleMdi.Document

Chaînes localisées

Langue : Français (France)
Titre du frame principal : SampleMdi
Nom du type de document : SampleMdi
Nom de filtre : SampleMdi Files (*.SMDI)
Nom court de nouveau fichier : SampleMdi
Nom long du type de fichier : SampleMdi.Document

< Précédent | Suivant > | Terminer | Annuler

Assistant Application MFC - SampleMdi

Fonctionnalités avancées

Vue d'ensemble
Type d'application
Prise en charge doc. composés
Chaînes du modèle de doc.
Prise en charge des bases de données
Fonctionnalités interface util.
Fonctionnalités avancées
Classes générées

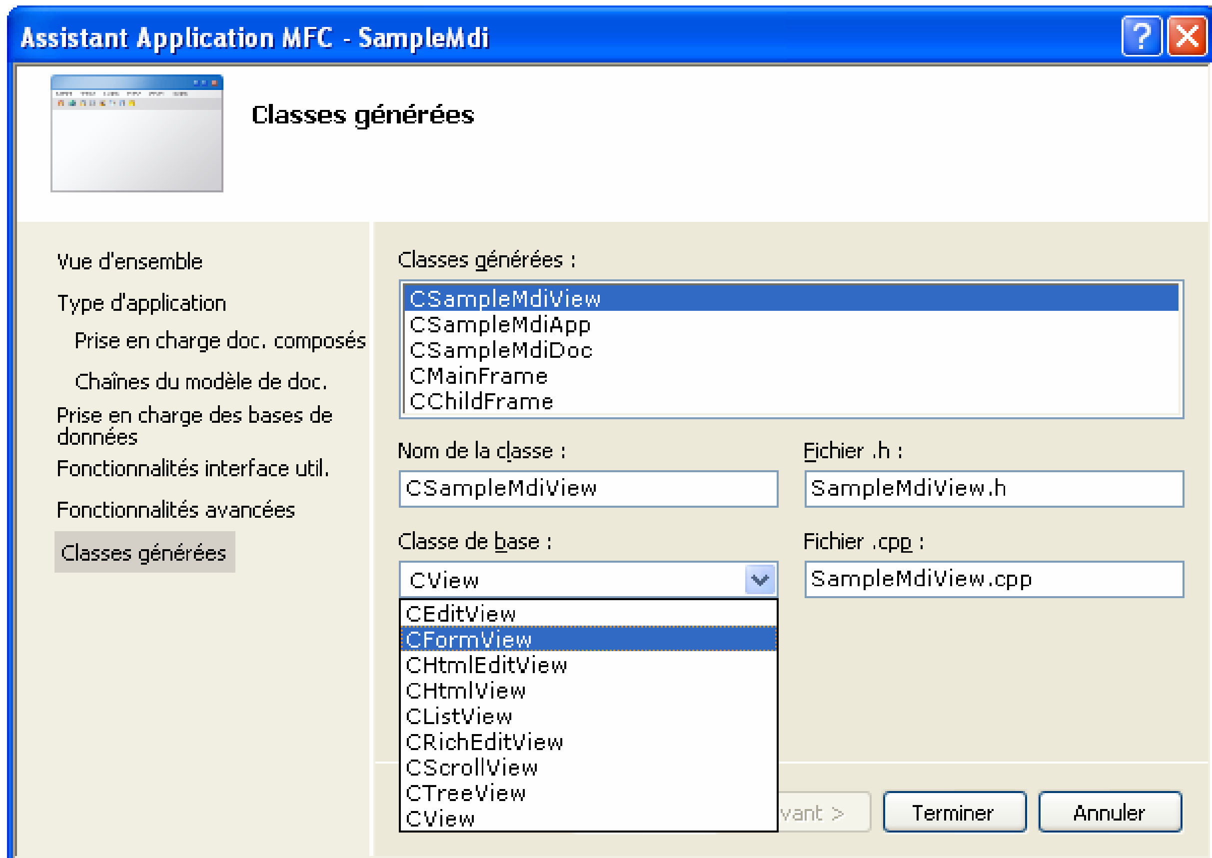
Fonctionnalités avancées :

- [ ] Aide contextuelle
  - Format WinHelp (non pris en charge)
  - Format HTML
- [x] Impression et aperçu avant impression
- [ ] Automation
- [x] Contrôles ActiveX
- [ ] MAPI (Messaging API)
- [ ] Windows sockets
- [ ] Active Accessibility
- [x] Manifeste des contrôles communs

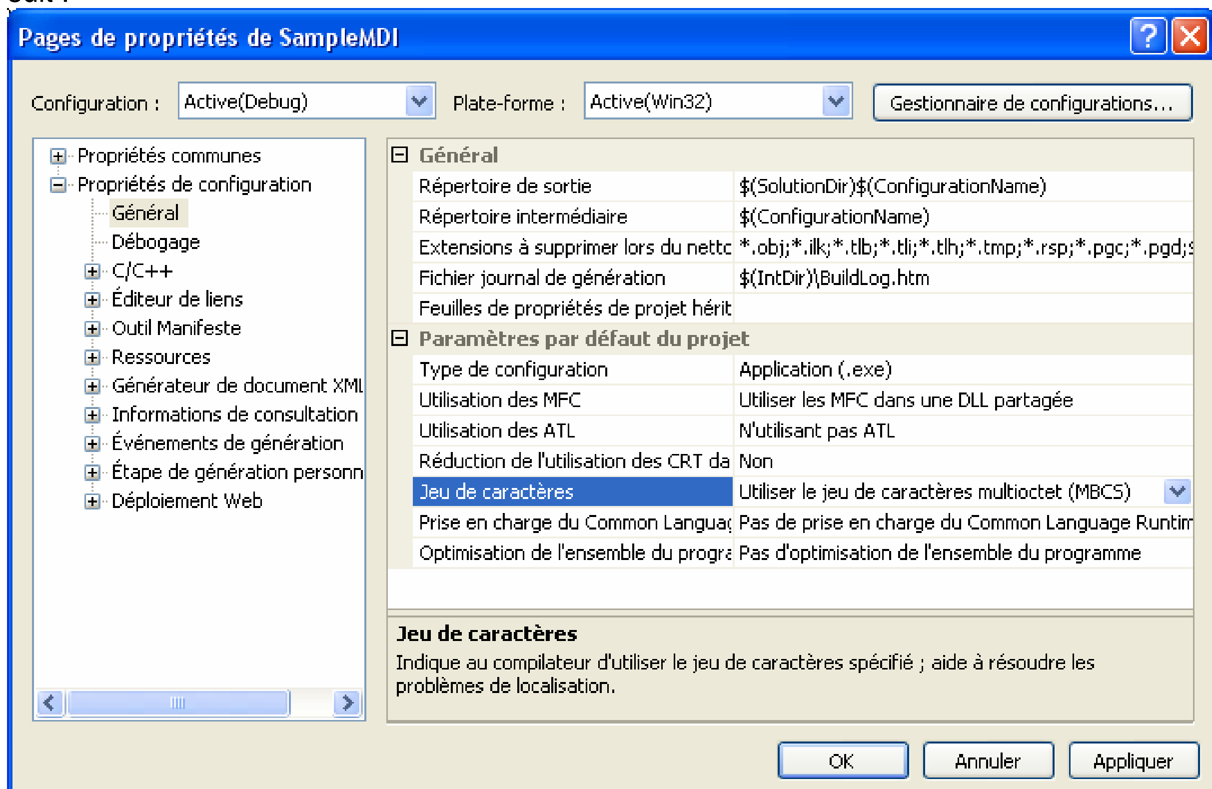
Nombre de fichiers dans la liste des fichiers récents : 4

< Précédent | Suivant > | Terminer | Annuler

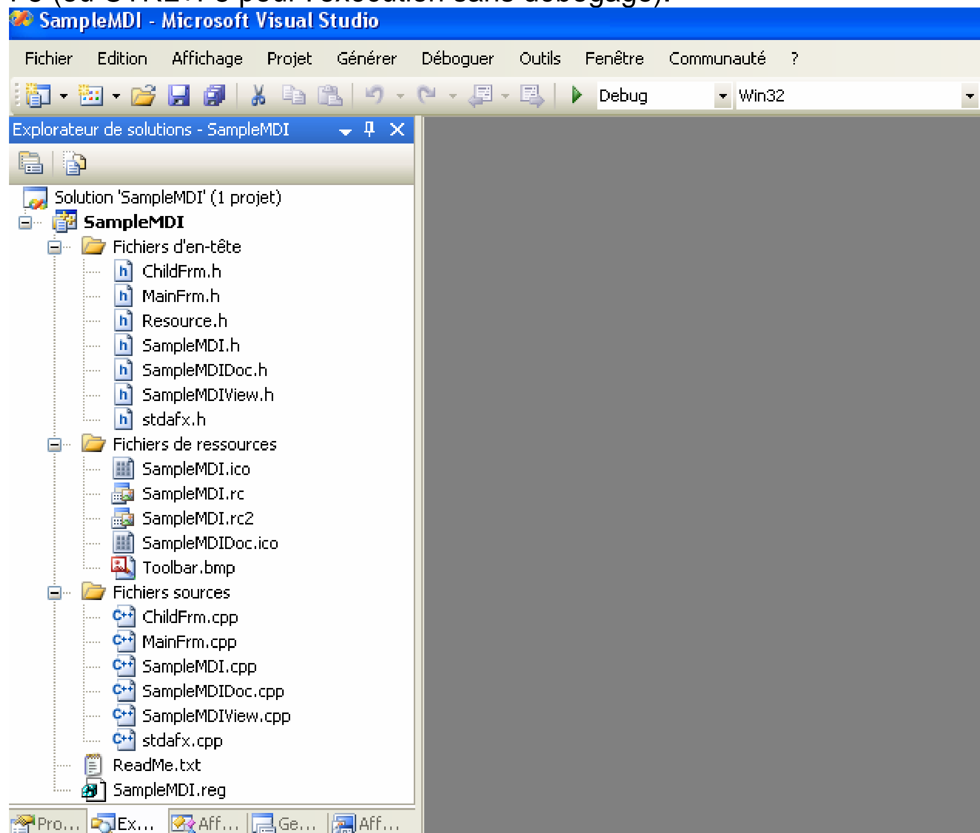
Enfin le dernier onglet nous permet de préciser le type classe pour la fenêtre générée par l’assistant :



Une fois le projet généré il faut savoir que le jeu de caractères utilisé est réglé par défaut sur Unicode, pour passer en mode char simple on modifiera les propriétés du projet comme suit :

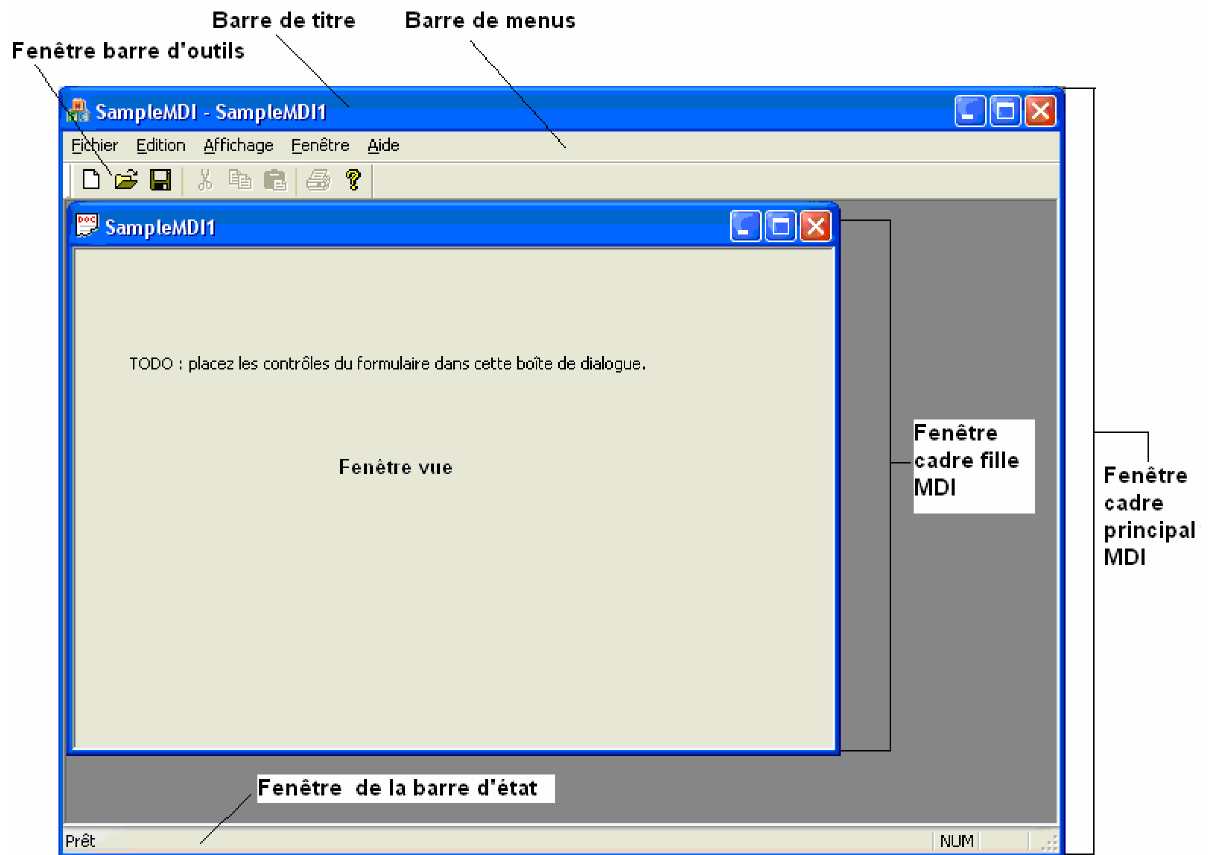


Après cette étape on passera à l'exécution du programme en mode debug par le raccourci F5 (ou CTRL+F5 pour l'exécution sans débogage).



Ou en utilisant la flèche verte à gauche de debug ou en passant par le menu déboguer.

On obtient le résultat suivant :



Le dessin ci-dessus nomme les différents éléments de l’architecture MDI.

II-A-1. Mode de fonctionnement du modèle MDI :

Commençons par étudier les déclarations des vues dans la fonction **InitInstance :**

```
BOOL CSampleMdiApp::InitInstance()
{
//…………..
CWinApp::InitInstance();
//…………..

// Inscrire les modèles de document de l'application. Ces modèles
	//  lient les documents, fenêtres frame et vues entre eux
	CMultiDocTemplate* pDocTemplate;
	pDocTemplate = new CMultiDocTemplate(IDR_SampleMdiTYPE,
		RUNTIME_CLASS(CSampleMdiDoc),
		RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
		RUNTIME_CLASS(CSampleMdiView));
	if (!pDocTemplate)
		return FALSE;
	AddDocTemplate(pDocTemplate);

	// crée la fenêtre frame MDI principale
	CMainFrame* pMainFrame = new CMainFrame;
	if (!pMainFrame || !pMainFrame->LoadFrame(IDR_MAINFRAME))
	{
		delete pMainFrame;
		return FALSE;
	}
	m_pMainWnd = pMainFrame;
	// appelle DragAcceptFiles uniquement s'il y a un suffixe
	//  Dans une application MDI, cet appel doit avoir lieu juste après la définition de m_pMainWnd
	// Activer les ouvertures via glisser-déplacer
	m_pMainWnd->DragAcceptFiles();

	// Activer les ouvertures d'exécution DDE
	EnableShellOpen();
	RegisterShellFileTypes(TRUE);
	// Analyser la ligne de commande pour les commandes shell standard, DDE, ouverture de
fichiers
	CCommandLineInfo cmdInfo;
	ParseCommandLine(cmdInfo);

	// Commandes de dispatch spécifiées sur la ligne de commande. Retournent FALSE si
	// l'application a été lancée avec /RegServer, /Register, /Unregserver ou /Unregister.
	if (!ProcessShellCommand(cmdInfo))
		return FALSE;
	// La fenêtre principale a été initialisée et peut donc être affichée et mise à jour
	pMainFrame->ShowWindow(m_nCmdShow);
	pMainFrame->UpdateWindow();

	return TRUE;
}
```

Le code généré par Visual est assez bien commenté, on retrouve dans l'ordre les différentes phases d'initialisation de l'application :

Les initialisations générales concernant le support des sockets, ou de contrôles richedit.
Ensuite la déclaration du document template.

On remarquera que les classes utilisées diffèrent de celles du modèle SDI.
La classe de description du template **CMultiDocTemplate** remplace **CSingleDocTemplate**.
Une classe **CChildFrame** est indiquée ,celle-ci est très importante dans un contexte MDI ,elle represente la classe parent de la vue (ici **CSampleMdiView**) .

C'est dans cette classe que tous les composants supplementaires de la view devront être initialisés comme par exemple une barre d'outils ,une barre de dialogue ou encore la mise en place d'un rideau de séparation (splitter).

On retrouve donc la similitude des initialisations faites dans le cas d'un projet SDI dans la classe fenêtre d'application **CMainFrame**.

Le document est ensuite inséré avec la fonction **AddDocTemplate**.
Si nous devions avoir une deuxième fenêtre dans notre application il faudrait fournir une nouvelle description en créant un autre objet **CMultiDocTemplate** .

Pour finir on appelera la fonction **AddDocTemplate** .
On procédera de même pour toutes les fenêtres de notre application.

II-A-2.Quelques remarques :

On devra indiquer une classe fille (MDIChild) différente à chaque fois que la vue aura besoin de composants spécifiques comme les barres de dialogues ou barres d'outils.

Si la vue a besoin du système de sauvegarde du document, on devra spécifier une classe document spécifique comme nous l'avons vu dans le modèle SDI.

Les autres fonctions appelées en dessous de la description des templates concernent l'initialisation de la fenêtre d'application (**CMainFrame)** ,le support des commandes DDE ,le traitement de la ligne de commande de l'application .

Le code généré est assez bien documenté et ne présente pas de nouveautés par rapport à celui du model SDI.

II-B. Etude des éléments composants le framework

II-B-1. La classe CMDIFrameWnd

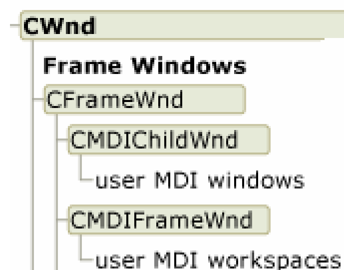
Notre classe **CMainFrame** (fenêtre de l'application) hérite de la classe **CMDIFrameWnd** cette classe fournit les fonctionnalités identiques à la fenêtre principale pour une application MDI, c'est-à-dire les fenêtres filles (child) à sa charge et créées par l'utilisateur .

Ces opérations consistent à :
L'activation de la fenêtre, le rangement en cascade des fenêtres, les opérations maximiser, restaurer.

Les fonctions propres à cette classe sont préfixées par le mot clef MDI en voici la liste :

CreateNewChild	Crée une nouvelle fenêtre fille.
MDIActivate	Active une autre fenêtre fille MDI.
MDICascade	Positionne les fenêtres filles MDI en cascade.
MDIGetActive	Retourne la fenêtre MDI active et positionne un flag pour savoir si elle est maximisée ou non.
MDIIconArrange	Repositionne toutes les fenêtres filles iconisées.
MDIMaximize	Maximise une fenêtre fille MDI.
MDINext	Active la fenêtre fille immédiatement derrière celle actuellement active, et la place derrière toutes les autres fenêtres filles.
MDIPrev	Active la fenêtre fille précédente et place la fenêtre fille actuellement active immédiatement derrière elle.
MDIRestore	Restaure une fenêtre fille MDI du mode maximisé ou minimisé
MDISetMenu	Remplace le menu d'une fenêtre fille MDI, le menu pop-up ou les deux
MDITile	Arrange toutes les fenêtres filles au format mosaïque.

Il est à noter que **CMDIFrameWnd** hérite de **CFrameWnd** qui elle-même hérite de la classe de base des fenêtres MFC la classe **CWnd**.



II-B-2. La classe CMDIChildWnd

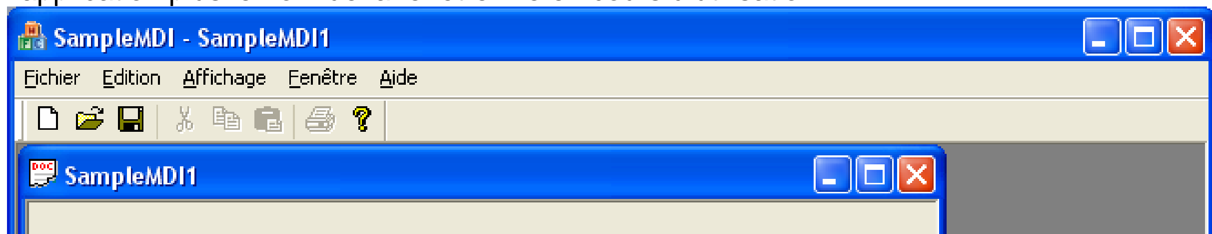
Notre classe **CChildWnd** hérite de la classe **CMDIChildWnd** qui représente le cadre fenêtre fille dans un context MDI.

Premier point important la fenêtre cadre fille communément appelée child ne dispose pas de menu.
Elle le partage avec celui de la classe d'application ; d'où ma remarque plus haut sur le fait qu'il est préférable à mon sens de disposer d'une seule barre de menu pour l'ensemble de l'application afin de ne pas perturber l'utilisateur.

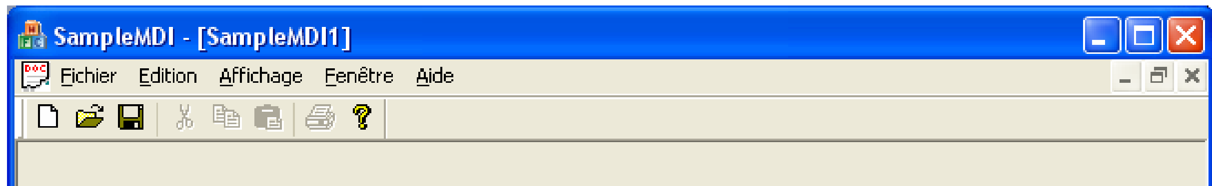
Autre point le titre du cadre principal MDI (MDI Frame) comprend aussi celui du cadre fille MDI . (Voir schéma plus haut pour les appellations utilisées).

II-B-2-a. Exemples :

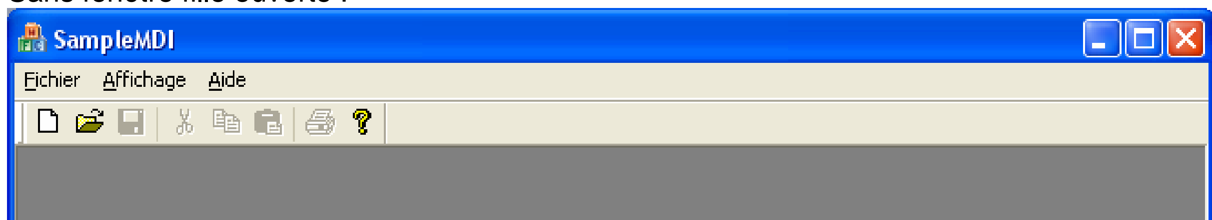
La fenêtre fille n'est pas maximisée, le nom du cadre MDI principal est composé du nom de l'application plus le nom de la fenêtre fille en cours d'utilisation.



Fenêtre fille maximisée on obtient :



Sans fenêtre fille ouverte :



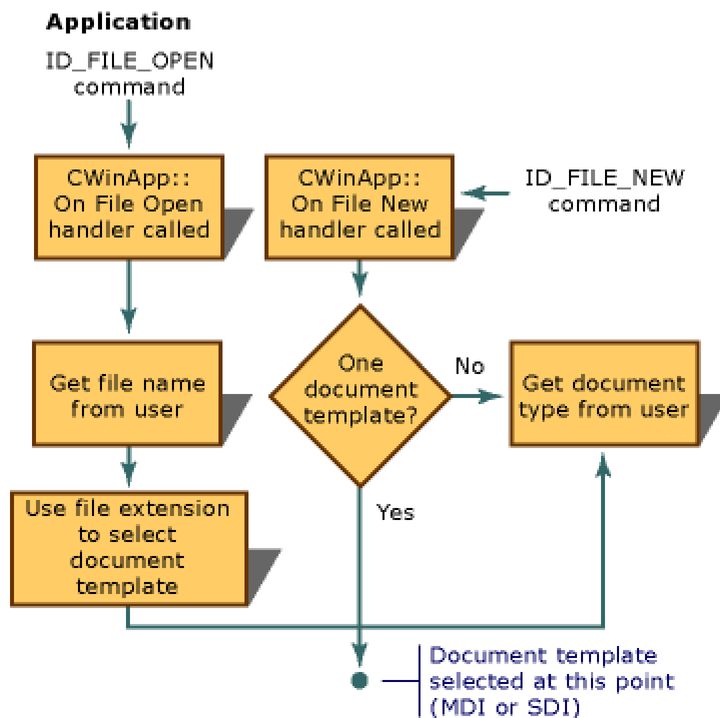
II-B-2-b. Quelques fonctions utiles :

CMDIFrameWnd* GetMDIFrame( )	Renvoie un pointeur sur la MDI frame ,la fonction AfxGetMainWnd() permet un accés dans tout le programme.
MDIActivate()	Active la MDI child
MDIDestroy()	Détruit la MDI child ,enléve le titre de la child dans la frame et désactive la child.
MDIMaximize()	Maximise la MDI child
MDIRestore()	Restaure la MDI child d'une position maximisée/minimisée

II-B-3. Séquences de création des objets dans une architecture MDI :

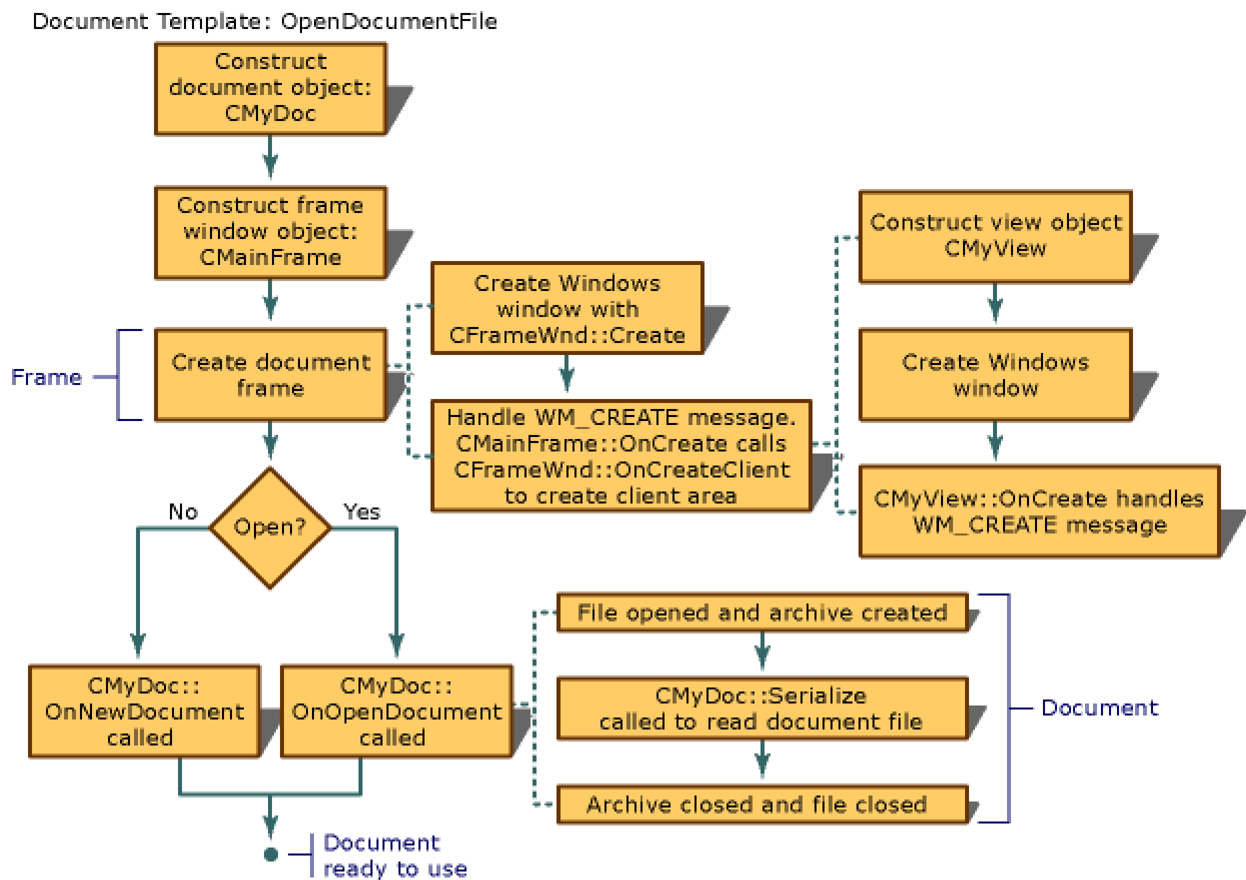
Pour bien comprendre l'architecture d'un programme MDI je vous propose les schémas suivants issus de la documentation MSDN :Creating New Documents, Windows, and Views

II-B-3-a. Séquences sur la création d'un document



Le schéma montre les deux formes d'appel pour l'ouverture d'un document soit par la commande **ID_FILE_OPEN** ce qui équivaut à la commande dans le programme « menu fichier ouvrir », l'utilisateur doit donc fournir le fichier qui correspondra à l'extension du fichier document template associé.
Soit la commande **ID_FILE_NEW** est appelée, et dans le cas d'une application MDI un nouveau document est créé.

II-B-3-b. Séquences sur la création d'une fenêtre cadre MDI et sa vue



Le schéma démarre avec la fonction **OpenDocumenFile** de la classe **CMultidocTemplate** Cette fonction reçoit en argument le nom du fichier à ouvrir ce qui conditionnera l'appel de la fonction **OnOpenDocument**, ou si l'argument est nul, la fonction **OnNewDocument** du document.

On distingue dans ce schéma trois grosses parties :

- La création de l'objet document avec en final dans le cas d'un fichier à lire, l'appel au mécanisme de sérialisation.
- La création de la fenêtre cadre MDI fille
- Et enfin le cadre MDI fille qui construit l'objet vue et l'initialise graphiquement.

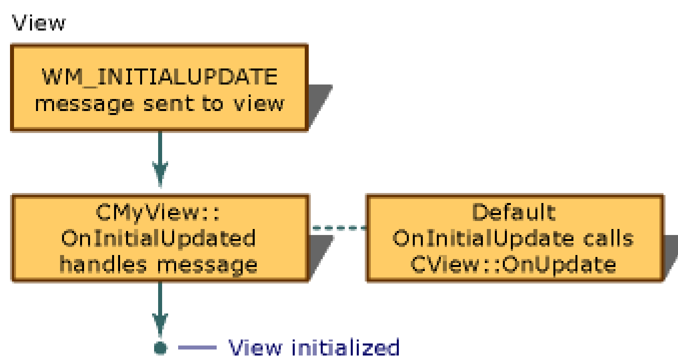
Une remarque : le schéma laisse suggérer l'indépendance du traitement lié au mécanisme de sérialisation du document avec la création du cadre MDI fille puis celle de la vue.
Cette impression est confirmée à l'examen du code MFC :

```
CDocument* CMultiDocTemplate::OpenDocumentFile(LPCTSTR lpszPathName,
	BOOL bMakeVisible)
{
	CDocument* pDocument = CreateNewDocument();
	if (pDocument == NULL)
	{
	TRACE(traceAppMsg, 0, "CDocTemplate::CreateNewDocument returned NULL.\n");
		AfxMessageBox(AFX_IDP_FAILED_TO_CREATE_DOC);
		return NULL;
	}
	ASSERT_VALID(pDocument);

	BOOL bAutoDelete = pDocument->m_bAutoDelete;
	pDocument->m_bAutoDelete = FALSE;   // don't destroy if something goes wrong
	CFrameWnd* pFrame = CreateNewFrame(pDocument, NULL);
	pDocument->m_bAutoDelete = bAutoDelete;
	if (pFrame == NULL)
	{
		AfxMessageBox(AFX_IDP_FAILED_TO_CREATE_DOC);
		delete pDocument;       // explicit delete on error
		return NULL;
	}
	ASSERT_VALID(pFrame);

	if (lpszPathName == NULL)
	{
		// create a new document - with default document name
		SetDefaultTitle(pDocument);

		// avoid creating temporary compound file when starting up invisible
		if (!bMakeVisible)
			pDocument->m_bEmbedded = TRUE;

		if (!pDocument->OnNewDocument())
		{
			// user has be alerted to what failed in OnNewDocument
		TRACE(traceAppMsg, 0, "CDocument::OnNewDocument returned FALSE.\n");
			pFrame->DestroyWindow();
			return NULL;
		}

		// it worked, now bump untitled count
		m_nUntitledCount++;
	}
	else
	{
		// open an existing document
		CWaitCursor wait;
		if (!pDocument->OnOpenDocument(lpszPathName))
		{
			// user has be alerted to what failed in OnOpenDocument
		TRACE(traceAppMsg, 0, "CDocument::OnOpenDocument returned FALSE.\n");
			pFrame->DestroyWindow();
			return NULL;
		}
		pDocument->SetPathName(lpszPathName);
	}

	InitialUpdateFrame(pFrame, pDocument, bMakeVisible);
	return pDocument;
}
```

On retrouve bien les différentes phases du schéma avec le choix lorsque le fichier du document est nul ou non, ce qui va conditionner l'appel à la lecture du document par son mécanisme de sérialisation.
Pour terminer, l'appel à la fonction **InitialUpdateFrame** va activer la vue et enverra le message **WM_INITIALUPDATE**.
Dans ces conditions il est clair que l'on ne doit pas essayer d'accéder à la vue dans le processus de lecture de l'archive dans l'objet document.
Les contrôles de la vue ne seront pas initialisés graphiquement (voir section suivante).

C'est lors de l'initialisation de la vue que l'on accédera aux données du document pour afficher la vue.

II-B-3-c. Séquences d'initialisation d'une vue



La séquence d'initialisation d'une vue est assez simple, elle reçoit le message **WM_INITIALUPDATE** qui déclenche la fonction **OnInitialUpdate** de votre vue.
L'appel à la fonction de la classe de base finit les initialisations graphiques de la vue ; en appelant dans le cas d'une **CFormView** la fonction **UpdateData(FALSE)** pour subclasser les contrôles définis dans la fonction **DoDataExchange**.
Il faudra donc bien veiller à commencer à travailler avec les variables contrôles après l'appel de la fonction de la classe base sous peine d'assertion d'erreur…

Developpez.com
Club des Développeurs

II-B-4. Routages des messages : les classes du frame work :

Après avoir décrit l'ensemble du mécanisme de création MDI, il nous reste à voir maintenant comment sont routés les messages de l'application, c'est à dire quels chemins ils empruntent à travers les différents objets en présence.

Lorsqu'un objet de cette classe reçoit une commande	..il traite lui même le message si possible, ou passe la main à...
La MDI frame window (**CMDIFrameWnd**)	1. **CMDIChildWnd** active 2. lui-même 3. l'application (objet **CWinApp**)
Document frame window (**CFrameWnd**, **CMDIChildWnd**)	1. La vue active 2. Lui même 3. L'application (objet **CWinApp**)
La vue (**CView, CFormView etc.)**	1. Lui même 2. Le document attaché à la vue
Document (**CDocument**)	1. Lui même 2. Le document template qui lui est attaché
Boîte de dialogue (**CDialog**)	1. Lui même 2. La fenêtre parent 3. L'application (objet **CWinApp**)

Le tableau ci-dessus montre bien le chemin de traitement des messages selon l'objet qui le reçoit.

Voila ce chapitre clôture (pour l'instant) les explications sur la création des objets du framework dans un contexte MDI.

II-C. Travailler avec plusieurs vues dans le modèle MDI

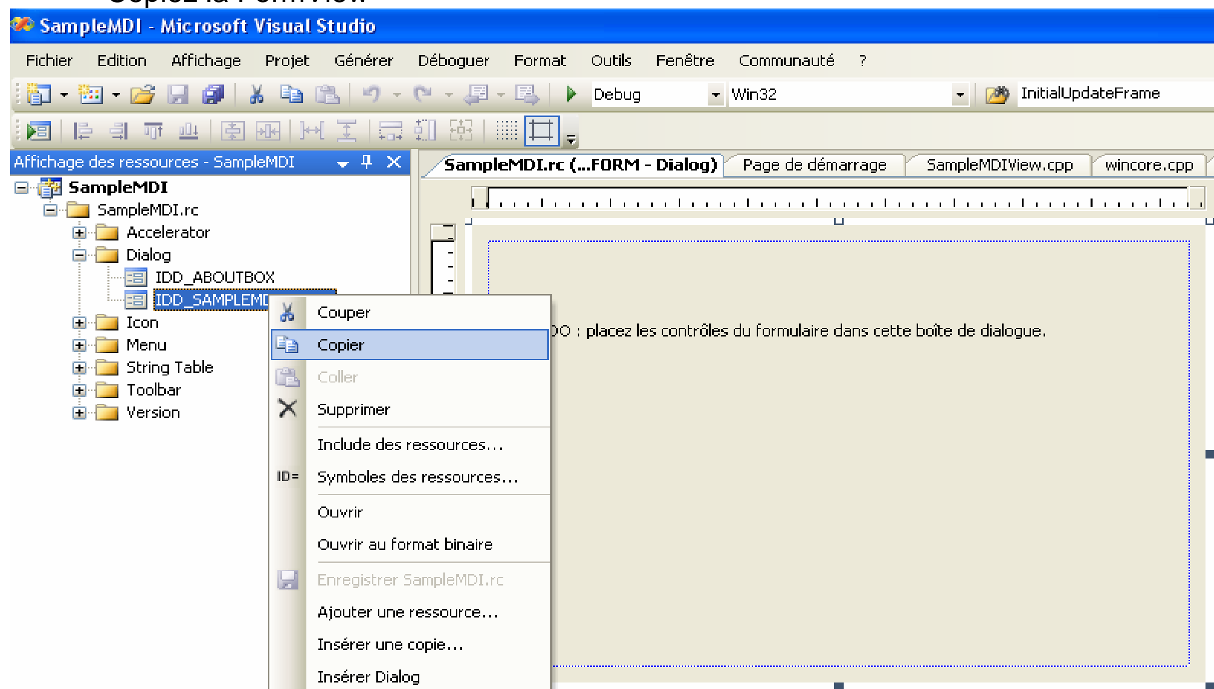
Puisque nous étudions le modèle MDI nous allons tout de suite rajouter une deuxième fenêtre à notre application.
On procédera comme suit :

Nous allons commencer par définir la fenêtre dans l'éditeur de ressources.
Le plus simple pour ne pas se tromper, c'est de faire un copier coller de la vue existante.
Cette précaution permet d'éviter de se tromper sur les attributs de notre vue :
Car il ne faut pas oublier qu'il existe deux sortes de fenêtres avec des contrôles : les boîtes de dialogues et les vues, les deux disposent d'attributs différents.

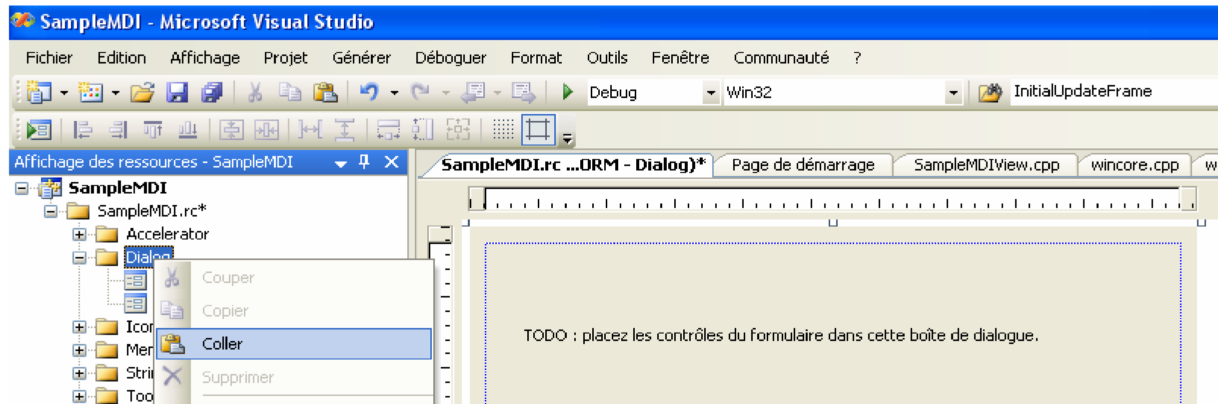
Une boîte de dialogue possédera l'attribut style pop up alors que la vue aura l'attribut style child.
On rajoutera un ou deux contrôles supplémentaires dans la nouvelle vue pour bien distinguer nos deux fenêtres.

II-C-1. Les modifications avec l'éditeur de ressources dans le détail :

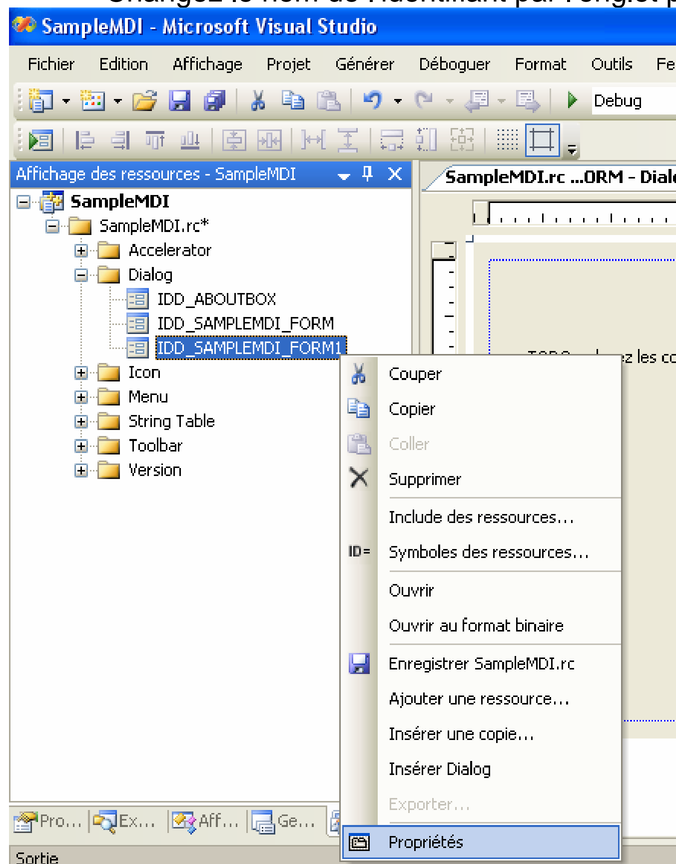
- Copiez la FormView



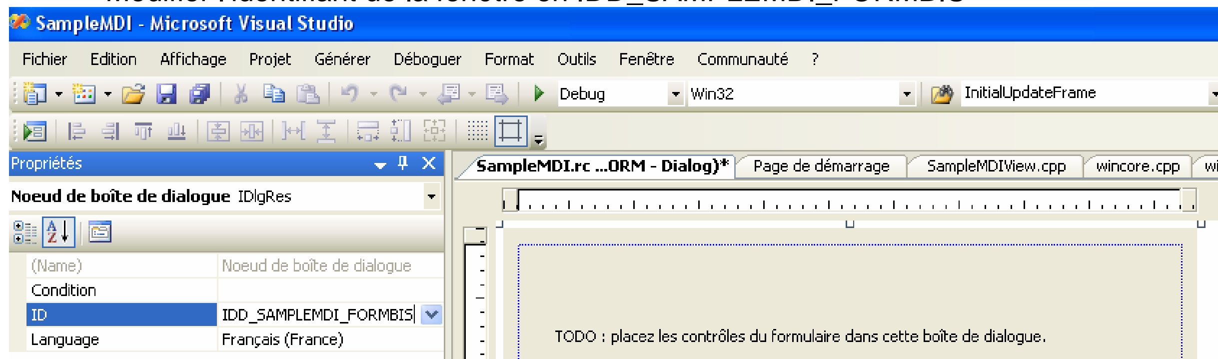
- Collez la ressource



- Changez le nom de l'identifiant par l'onglet propriétés ou clic droit propriétés.

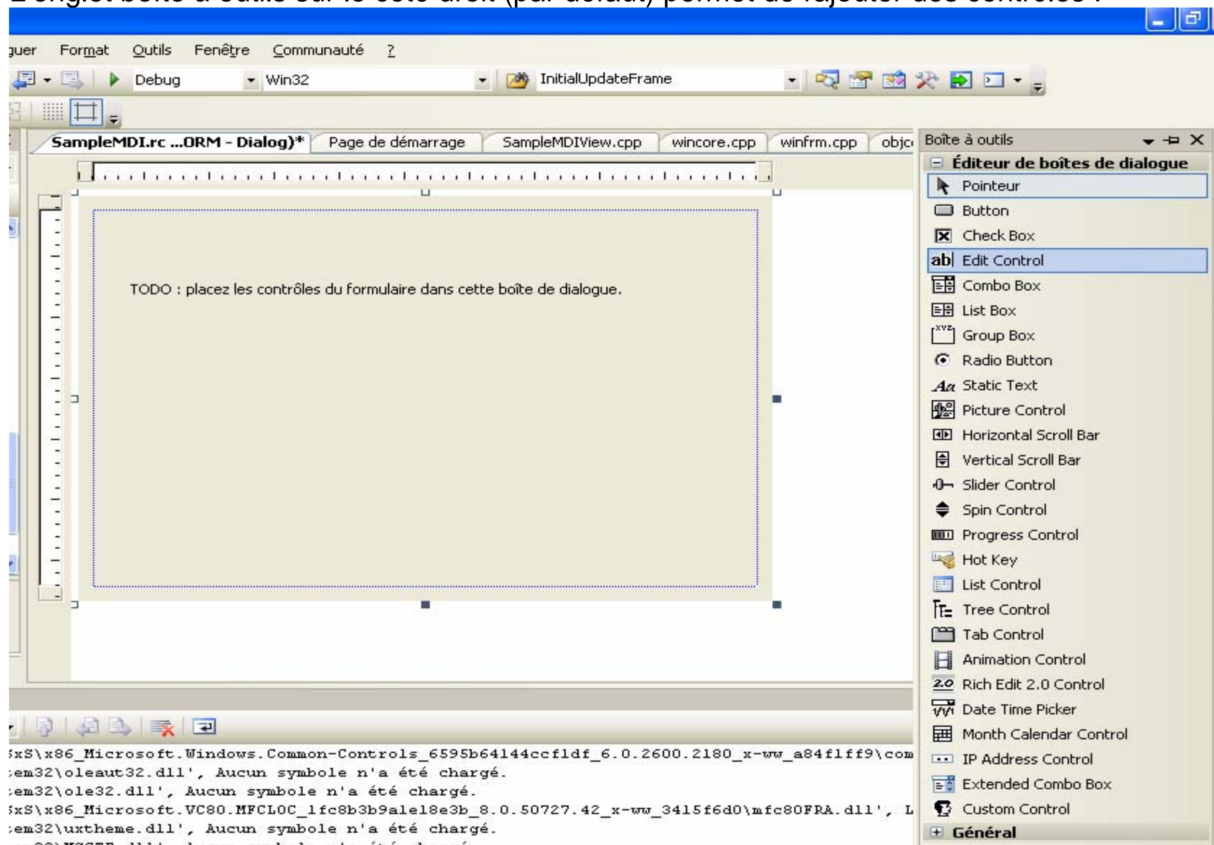


- Modifier l'identifiant de la fenêtre en IDD_SAMPLEMDI_FORMBIS

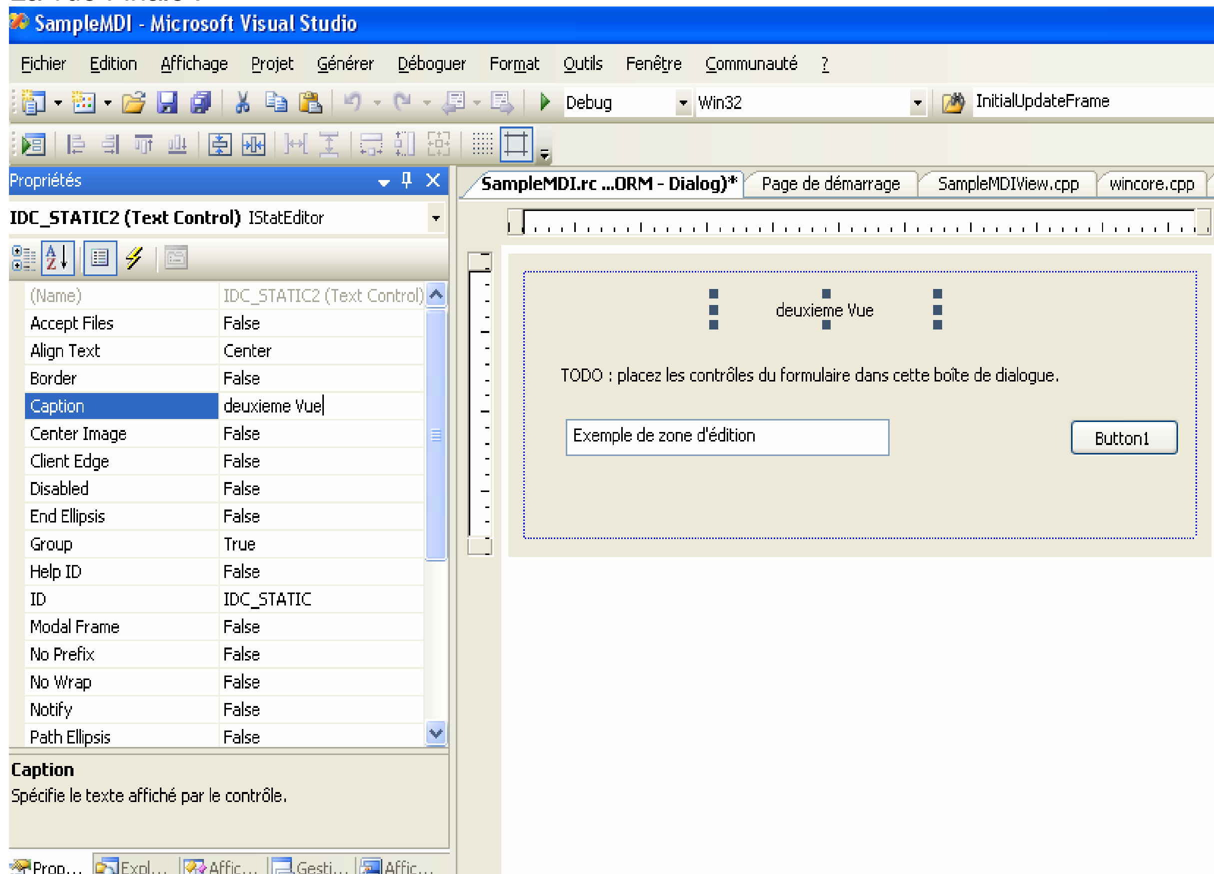


- Rajoutez des contrôles sur notre vue :

L'onglet boîte à outils sur le coté droit (par défaut) permet de rajouter des contrôles :



La vue Finale :



II-C-2. Quelques remarques générales sur l'environnement de développement

Je profite de ces étapes pour donner quelques explications qui seront bénéfiques aux anciens utilisateurs de **Visual 6.0**, qui à l'instant précis doivent se sentir un peu perdus …
ClassWizard n'existe plus ! , il est remplacé par des listes disponibles dans l'onglet propriétés (écran ci-dessus).

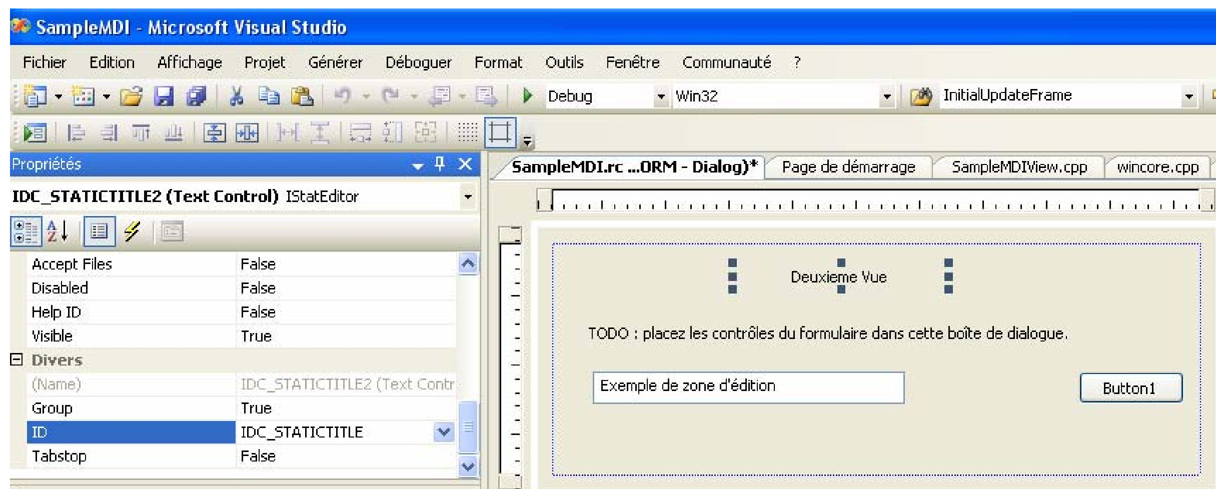
II-C-2-a. En ce qui concerne la modification des propriétés des contrôles :

Il n'y a plus une fenêtre propriétés avec différents onglets mais une liste générale des propriétés comme sur l'écran ci-dessus, qui montre la modification du titre d'un static.

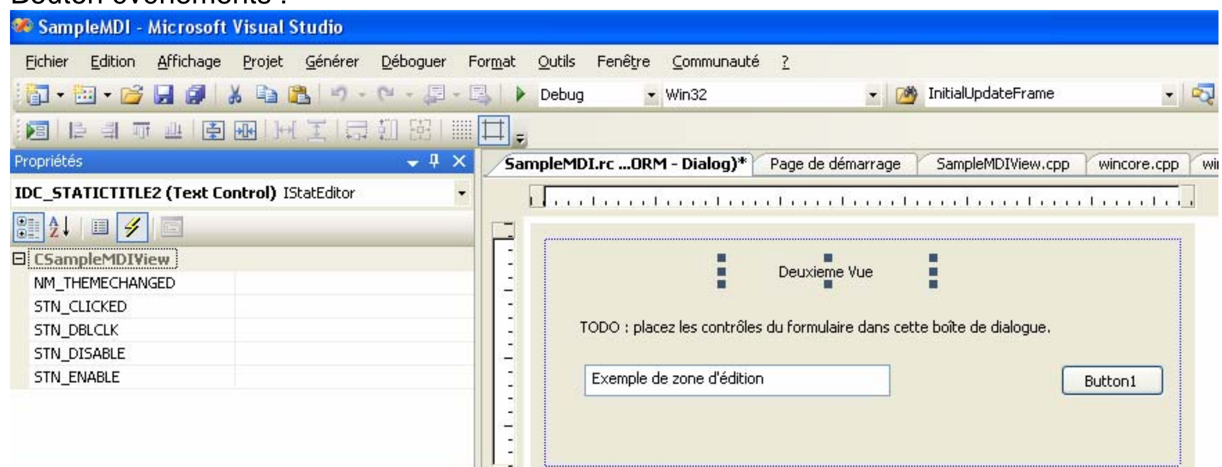
Pour les habitués l'appui sur la touche entrée sur le contrôle supprime irrémédiablement le titre du contrôle, il va falloir revoir nos habitudes d'utilisation …

Pour accéder aux propriétés d'un contrôle on sélectionnera l'onglet propriétés ou on appuiera sur la combinaison d'ALT+ Entrée.

Le bouton en forme d'éclair jaune permet de traiter les événements du contrôle.
Je rappelle que dans le cas d'un static pour accéder aux événements il faut que son identifiant soit différent de celui par défaut (IDC_STATIC)



Bouton événements :



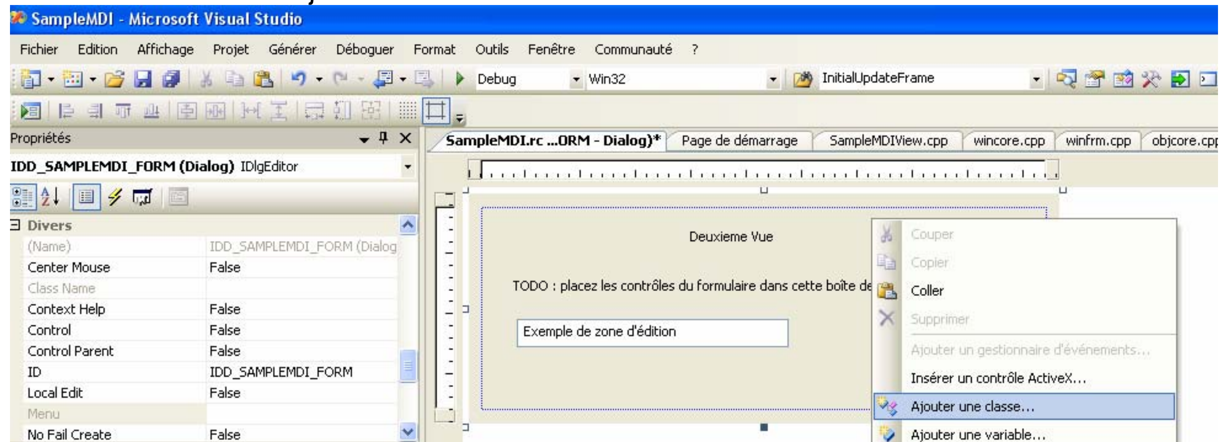
Les deux premiers boutons permettent de passer à une liste triée dans l'ordre alphabétique
Ou par catégorie, ce qui ressemblera dans le cas des propriétés du contrôle à nos différents onglets de notre défunte fenêtre propriétés.

II-C-2-b. Différences sur la génération de code avec Visual 6.:

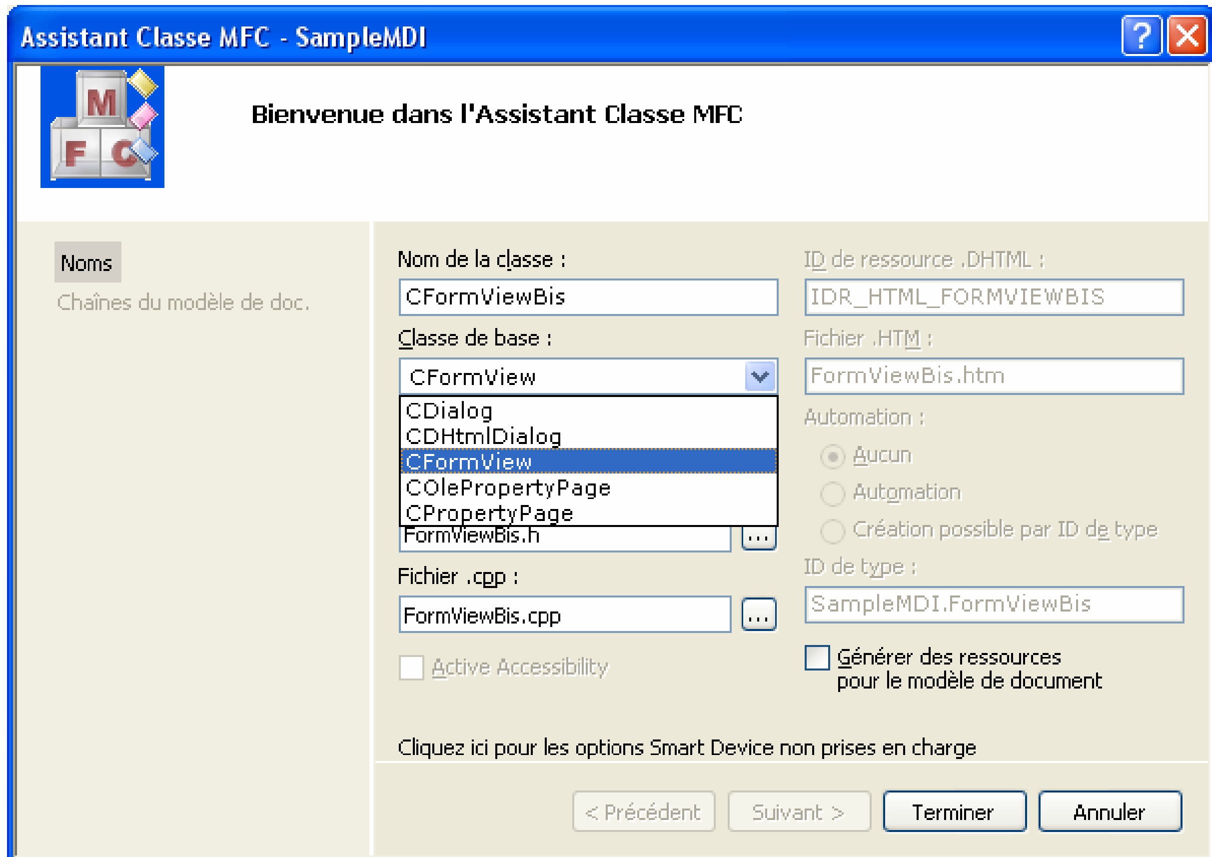
Visual 6.0 utilisait des lignes spécifiques de commentaires pour repérer les différents ajouts dans la classe : les messages, les données membres etc. ..
Ou encore les gardes anti-inclusions pour les fichiers d'entête de classe.
Visual 2005 n'utilise plus ce procédé.
Tout se fait de manière dynamique par le maintient du fichier .ncb de l'application.
Conséquence : lors d'un portage d'une application **Visual 6.0** vers **Visual 2005**, les nouvelles classes créées dans le nouvel environnement ne seront plus modifiables par l'assistant de **Visual 6.0.**
En dehors de ces différences d'interfaces il y a bien sur une évolution majeure du compilateur qui respecte la norme C++.
Les MFC ont été aussi modifiées, je vous invite pour plus de renseignements sur le portage d'une application **Visual 6.0** vers **Visual 2005** à consulter mon tutoriel :
http://farscape.developpez.com/tutoriels/migration-vc6-vc2005/

II-C-2-c. Génération de la classe CFormView associée :

Là aussi nos habitudes sont perturbées : un double clic sur la fenêtre ne déclenche pas l'assistant de génération de la classe.
On fera un clic droit : Ajouter une classe :



Pour arriver sur l'écran :



Il suffit de préciser le nom de notre classe : **CFormViewBis**
Et surtout de sélectionner la bonne classe de base **CFormView,** le réglage étant par défaut sur la classe **CDialog**.
Le bouton terminer clôture notre intervention et génère le .h et le .cpp correspondant.

II-C-3. Paramétrage du document Template dans InitInstance

Il nous reste plus qu'à rajouter notre description dans la fonction **InitInstance** en déclarant un nouveau document Template :

```
//…….
// Inscrire les modèles de document de l'application. Ces modèles
      //  lient les documents, fenêtres frame et vues entre eux
      CMultiDocTemplate* pDocTemplate;
      pDocTemplate = new CMultiDocTemplate(IDR_SampleMdiTYPE,
            RUNTIME_CLASS(CSampleMdiDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CSampleMdiView));
      if (!pDocTemplate)      return FALSE;
      AddDocTemplate(pDocTemplate);

      pDocTemplate = new CMultiDocTemplate(IDR_SampleMdiTYPE,
            RUNTIME_CLASS(CSampleMdiDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CFormViewBis));// notre nouvelle fenêtre
      if (!pDocTemplate)            return FALSE;

      AddDocTemplate(pDocTemplate);

      // crée la fenêtre frame MDI principale
      CMainFrame* pMainFrame = new CMainFrame;
      if (!pMainFrame || !pMainFrame->LoadFrame(IDR_MAINFRAME))
      {
            delete pMainFrame;
            return FALSE;
      }
//…….
```

II-C-3-a. Notes :

Pour chaque fenêtre un menu différent peut être associé, je ne vous le conseille pas :
Avoir un menu différent à chaque fois que l'on clique sur une fenêtre, peut être très perturbant pour un utilisateur.
J'ai utilisé volontairement la même classe document et Child pour cette nouvelle fenêtre n'ayant pas de traitement particulier à mettre en place.

On n'oubliera pas d'insérer l'entête de fichier de notre classe au sommet de la source de notre classe d'application : à la suite de la déclaration de notre première fenêtre.

```
// SampleMdi.cpp : Définit les comportements de classe pour l'application.
//
#include "stdafx.h"
#include "SampleMdi.h"
#include "MainFrm.h"

#include "ChildFrm.h"
#include "SampleMdiDoc.h"
#include "SampleMdiView.h"
#include "FormViewBis.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif
```

II-C-4. Exécutons notre application :

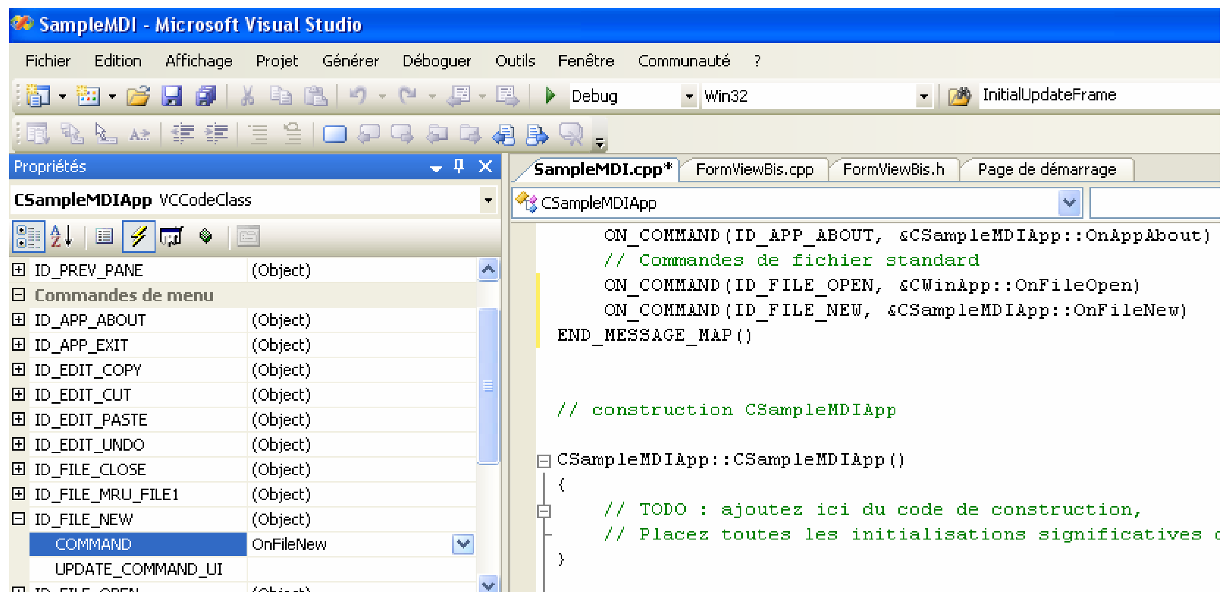
Premier constat un sélecteur de fenêtre apparaît pour nous inviter à choisir la fenêtre à ouvrir
Ce qui implique que l'application ouvre automatiquement une fenêtre au démarrage.
Pour contrer ce mécanisme Il suffira de mettre en commentaire la séquence suivante contenue dans la fonction **InitInstance** de la classe d'application :

```
//……….
// Activer les ouvertures d'exécution DDE
	EnableShellOpen();
	RegisterShellFileTypes(TRUE);

	// Analyser la ligne de commande pour les commandes shell
standard, DDE, ouverture de fichiers
	//CCommandLineInfo cmdInfo;
	//ParseCommandLine(cmdInfo);


	// Commandes de dispatch spécifiées sur la ligne de commande.
Retournent FALSE si
	// l'application a été lancée avec /RegServer, /Register,
/Unregserver ou /Unregister.
	//if (!ProcessShellCommand(cmdInfo))
	//	return FALSE;
	// La fenêtre principale a été initialisée et peut donc être
affichée et mise à jour
	pMainFrame->ShowWindow(m_nCmdShow);
	pMainFrame->UpdateWindow();

	return TRUE;
}
```

Reste le problème du sélecteur de fenêtre :
Lorsque l'on a défini plusieurs objets de la classe **CMultiDocTemplate** dans **InitInstance** à l'appel la commande **ID_FILE_NEW** une fenêtre apparaît avec la liste des différents noms de documents disponibles.
Cette fenêtre est générée par l'appel de la fonction **CWinApp :: OnfileNew** associée à la commande **ID_FILE_NEW**.

```
BEGIN_MESSAGE_MAP(CSampleMdiApp, CWinApp)
	ON_COMMAND(ID_APP_ABOUT, &CSampleMdiApp::OnAppAbout)
	// Commandes de fichier standard
	ON_COMMAND(ID_FILE_NEW, &CWinApp::OnFileNew)
	ON_COMMAND(ID_FILE_OPEN, &CWinApp::OnFileOpen)
END_MESSAGE_MAP()
```

Pour intercepter le message placez le curseur sur la ligne **END_MESSAGE_MAP()**
Puis tapez ALT+ Entrée.
Sur le bouton éclair correspondant aux événements, interceptez le message **ID_FILE_NEW** :




```
// gestionnaires de messages pour CSampleMDIApp
void CSampleMDIApp::OnFileNew()
{
	// TODO : ajoutez ici le code de votre gestionnaire de commande
}
```

Ce qui va nous amener aux modifications suivantes dans la classe d'application :
Si nous voulons, lancer une fenêtre par l'intermédiaire de l'interface menu ou bouton de l'application nous devons disposer d'un pointeur sur le document Template correspondant.
Ce qui veut dire que nous devons stocker ce pointeur dans notre classe d'application, et ce pour chaque fenêtre.
Une solution plus fine consistera à utiliser la liste stockée par les MFC en modifiant un peu la classe **CMultidocTemplate**, pour finalement faire une fonction dans la classe d'application Permettant de lancer une fenêtre en utilisant l'argument correspondant à la signature de classe d'une fenêtre (CRuntimeClass).

II-C-5. Détails des modifications :

La classe **CMultidocTemplateEx** et la déclaration de la fonction d'appel d'ouverture d'une fenêtre par sa signature :

```
class CMultiDocTemplateEx :public CMultiDocTemplate
{
public:
      CMultiDocTemplateEx( UINT nIDResource,
                              CRuntimeClass* pDocClass,
                              CRuntimeClass* pFrameClass,
                              CRuntimeClass* pViewClass ):
      CMultiDocTemplate(nIDResource,pDocClass,
                          pFrameClass,
                          pViewClass),m_pDocClass(pDocClass),

m_pFrameClass(pFrameClass),
m_pViewClass(pViewClass){}

      CRuntimeClass* GetDocClass(){return m_pDocClass;}
      CRuntimeClass* GetFrameClass(){return m_pFrameClass;}
      CRuntimeClass* GetViewClass(){return m_pViewClass;}

public:
      CRuntimeClass* m_pDocClass;
      CRuntimeClass* m_pFrameClass;
      CRuntimeClass* m_pViewClass;
};
class CSampleMDIApp : public CWinApp
{
public:
      CSampleMDIApp();

private:

// Substitutions
public:
      virtual BOOL InitInstance();
      // ouverture d'une vue correspondant à la signature pclass
(correspond a OpenDocumentFile )
    CDocument* OpenByRuntimeClass(CRuntimeClass *pClass,LPCTSTR
lpszPathName= NULL, BOOL bMakeVisible = TRUE );
```

Dans initInstance:

```
CMultiDocTemplateEx *pDocTemplate = new
CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CSampleMDIView));
      if (!pDocTemplate)return FALSE;

      AddDocTemplate(pDocTemplate);

      pDocTemplate = new CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CFormViewBis));

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);
```

La fonction d'ouverture et la fonction d'appel pour **ID_FILE_NEW** :

```
// gestionnaires de messages pour CSampleMDIApp
CDocument* CSampleMDIApp::OpenByRuntimeClass(CRuntimeClass *pClass,LPCTSTR
lpszPathName,
   BOOL bMakeVisible )
{
      CMultiDocTemplateEx* pTemplate;
     POSITION pos = GetFirstDocTemplatePosition();
      while (pos != NULL)
      {
            pTemplate = static_cast<CMultiDocTemplateEx
*>(GetNextDocTemplate(pos));
            ASSERT(pTemplate);
            if(pTemplate->GetViewClass()==pClass)
                  return pTemplate-
>OpenDocumentFile(lpszPathName,bMakeVisible);
      }
      return NULL;

}
void CSampleMDIApp::OnFileNew()
{
      // TODO : ajoutez ici le code de votre gestionnaire de commande
      OpenByRuntimeClass(RUNTIME_CLASS(CSampleMDIView));
}
```

Ce mode d'ouverture est intéressant, mais a l'inconvénient de devoir disposer de l'include de la classe au moment de l'appel.

Une autre approche serait de faire une fonction permettant l'appel par l'identifiant de fenêtre, On modifiera le code comme suit :

La classe **CMultidocTemplateEx :**

```
class CMultiDocTemplateEx :public CMultiDocTemplate
{
public:
      CMultiDocTemplateEx( UINT nIDResource,
                                    CRuntimeClass* pDocClass,
                                    CRuntimeClass* pFrameClass,
                                    CRuntimeClass* pViewClass,
                                    UINT nID=0
                                    ):
      CMultiDocTemplate(nIDResource,pDocClass,
                                pFrameClass,
                                pViewClass),m_pDocClass(pDocClass),
                                            m_pFrameClass(pFrameClass),
                                      m_pViewClass(pViewClass),m_nID(nID)
      {
      }
      CRuntimeClass* GetDocClass(){return m_pDocClass;}
      CRuntimeClass* GetFrameClass(){return m_pFrameClass;}
      CRuntimeClass* GetViewClass(){return m_pViewClass;}
public:
      CRuntimeClass* m_pDocClass;
      CRuntimeClass* m_pFrameClass;
      CRuntimeClass* m_pViewClass;
      unsigned int   m_nID;
};
```

La fonction d'appel par identifiant de fenêtre :

```
CDocument* CSampleMDIApp::OpenByIdClass(UINT nID,LPCTSTR lpszPathName/*=
NULL*/,BOOL bMakeVisible /*= TRUE */)
{
CMultiDocTemplateEx* pTemplate;
    POSITION pos = GetFirstDocTemplatePosition();
      while (pos != NULL)
      {
pTemplate = static_cast<CMultiDocTemplateEx *>(GetNextDocTemplate(pos));
            ASSERT(pTemplate);
if(pTemplate->m_nID==nID)
            return pTemplate->OpenDocumentFile(lpszPathName,bMakeVisible);
      }
      return NULL;
}
```

Les modifications dans **InitInstance** :

```
CMultiDocTemplateEx *pDocTemplate = new
CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CSampleMDIView),IDD_SAMPLEMDI_FORM);

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);

      pDocTemplate = new CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CFormViewBis),IDD_SAMPLEMDI_FORMBIS);

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);
```

Par rapport au code d'origine on rajoute juste l'identifiant de la fenêtre.

II-C-6. Conclusions :

Nous savons déclarer plusieurs fenêtres dans notre application.
Nous contrôlons leur création en utilisant le stockage des documents Template réalisés par les MFC.
L'appel pouvant se faire par des éléments de l'interface menu, bouton, ou par un appel direct de nos fonctions déclarées dans la classe d'application.
Comme nous disposons d'un accès permanent à celle-ci dans tout notre programme avec la fonction **AfxGetApp()** ; Il sera facile de l'appeler à tout moment, Il suffira de caster le retour de cette fonction avec la classe d'application adéquate :

Exemple : Appel de la deuxième fenêtre à partir de la première

```
void CSampleMDIView::OnInitialUpdate()
{
      CFormView::OnInitialUpdate();
      ResizeParentToFit();
      static_cast<CSampleMDIApp *>(AfxGetApp())-
>OpenByRuntimeClass(RUNTIME_CLASS(CFormViewBis));
// ou
      static_cast<CSampleMDIApp *>(AfxGetApp())-
>OpenByIdClass(IDD_SAMPLEMDI_FORMBIS);
}
```

Dans mon exemple nous aurons au final une application avec deux fenêtres de la classe **CFormView** gérées par deux objets **MDIChild** le tout contrôlé par un objet document, donc au final deux vues, deux cadres de travail (Child), deux documents.

II-D. Association de plusieurs vues sur un même document :

II-D-1. Introduction :

Il existe d'autres configurations intéressantes donnant à la classe document un nouveau rôle.
En effet il est possible d'avoir plusieurs fenêtres associées à un même document.
Dans quel cas a-t-on besoin d'avoir des fenêtres différentes pour un même document ?
Chaque fois que l'on voudra avoir un lien entre plusieurs fenêtres pour des échanges de données ou des représentations de données d'une manière différente, le document donnant par ses méthodes, un point d'accès pratique et facile aux différentes fenêtres.
Exemple :
Dans une **CFormView** je saisis des données pour l'affichage d'un graphe, dans une **CView** je les dessine sous forme de tableau récapitulatif**,** et dans une autre **CView** je les représente sous forme de diagrammes, histogrammes etc...

II-D-2. Mise en place :

L'association de plusieurs fenêtres sur un seul document s'effectue sur la déclaration du document Template dans la fonction **InitInstance** de la classe d'application.
La seule différence c'est que nous n'avons pas forcement besoin d'appeler la fonction **AddocTemplate** puisque cette vue sera rattachée à un document existant.
Modifions un peu notre code précédent en rajoutant un autre document Template en stockant cette fois-ci le pointeur :

```
CMultiDocTemplateEx *pDocTemplate = new
CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CSampleMDIView),IDD_SAMPLEMDI_FORM);

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);
      pDocTemplate = new CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CFormViewBis),IDD_SAMPLEMDI_FORMBIS);
      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);

      m_pTemlate= new CMultiDocTemplateEx(IDR_SampleMDITYPE,
                        RUNTIME_CLASS(CSampleMDIDoc),
                        RUNTIME_CLASS(CChildFrame), // frame enfant MDI
personnalisé
                        RUNTIME_CLASS(CEditView));
```

La troisième déclaration comprend donc juste une déclaration du Template sans faire l'appel à **AddDocTemplate**.
Une petite remarque sur ce dernier point :
La création de document Template par un **new** et l'appel de la fonction **AddDocTemplate** nous dispensent de détruire l'objet, mais dans le cas présent nous n'utilisons pas **AddDocTemplate** il faudra donc se charger de détruire le document Template .(ici m_pTemplate), le meilleur endroit sera la fonction virtuelle **OnExitInstance**.

Pour rajouter un objet de la classe **CEditView** par exemple dans la fonction **OnInitialUpdate** de la classe **CSampleMDIView** on procédera comme suit:

```
void CSampleMDIView::OnInitialUpdate()
{
	CFormView::OnInitialUpdate();
	ResizeParentToFit();

	CSampleMDIApp *TheApp=static_cast<CSampleMDIApp *>(AfxGetApp());
	TheApp->OpenByIdClass(IDD_SAMPLEMDI_FORMBIS);

	CMultiDocTemplateEx *pTemplate=GetDocument()->GetDocTemplate();
	if(pTemplate)
	{
	   CFrameWnd * pFrame =TheApp->m_pTemlate-
>CreateNewFrame(GetDocument(),GetParentFrame());
	   pTemplate->InitialUpdateFrame(pFrame,GetDocument());
	}
}
```

Le détail de la fonction **GetTemplateByIdClass** dans la classe d'application**:**

```
CDocument* CSampleMDIApp::OpenByIdClass(UINT nID,LPCTSTR lpszPathName/*=
NULL*/,BOOL bMakeVisible /*= TRUE */)
{
   CMultiDocTemplateEx* pTemplate=GetTemplateByIdClass(nID);
   if(pTemplate) return pTemplate-
>OpenDocumentFile(lpszPathName,bMakeVisible);
	return NULL;
}
CMultiDocTemplateEx *CSampleMDIApp::GetTemplateByIdClass(UINT nID)
{
  CMultiDocTemplateEx* pTemplate;
    POSITION pos = GetFirstDocTemplatePosition();
	while (pos != NULL)
	{
		pTemplate = static_cast<CMultiDocTemplateEx
*>(GetNextDocTemplate(pos));
		ASSERT(pTemplate);
		if(pTemplate->m_nID==nID) return pTemplate;
	}
	return NULL;
}
```

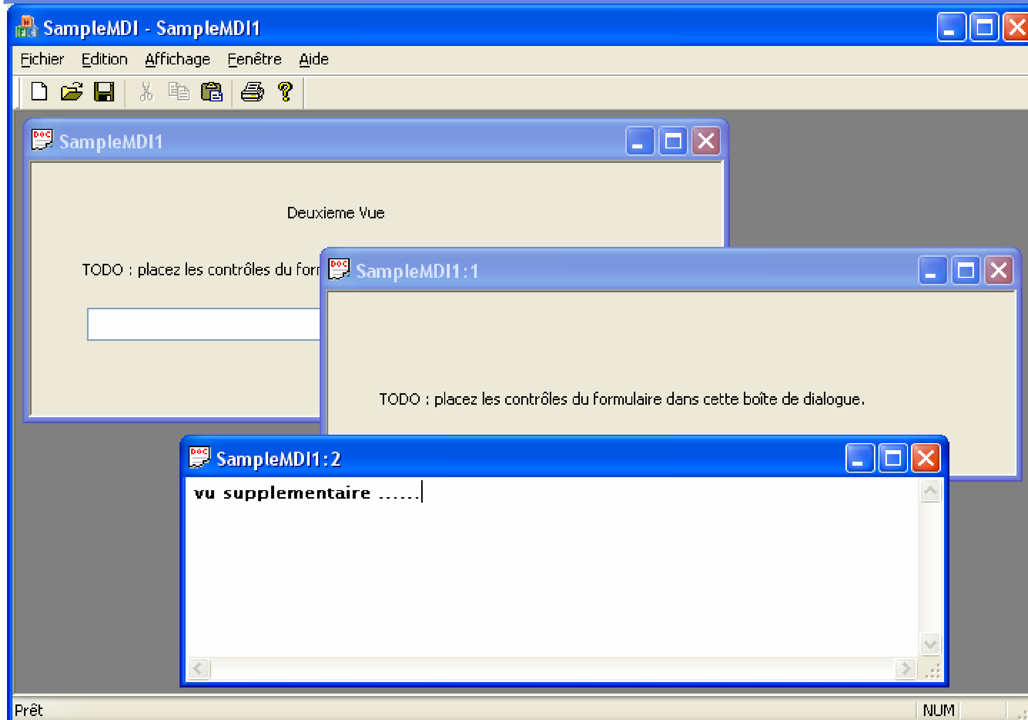
Bien sûr le même appel pouvant être fait en réponse à une commande sur un menu pour la vue en cours.(**CSampleMDIView**).

II-D-2-a. Les étapes du traitement :

Je commence par récupérer le document Template de la vue grâce à la fonction **GetDocTemplate** de la classe **CDocument .**
Ensuite il suffit de créer un cadre MDI fille pour la nouvelle vue , pour cela on utilisera le template associé en appelant la fonction **CreateNewFrame**.
On fournira à cette fonction le document actuel de la vue (**GetDocument()**) ainsi que son cadre MDI (**GetParentFrame()**).
Enfin pour finir on initialise le cadre MDI avec la fonction **InitialUpdateFrame**.

On répétera le même processus pour la description et la création d'une fenêtre supplémentaire.

II-D-2-b. Le résultat :



Vous noterez comment sont numérotées les fenêtres d'un même document.
Au final nous disposons d'un objet document gérant plusieurs fenêtres.
A partir de cet objet il nous est facile d'énumérer les fenêtres qui lui sont rattachées :

```
    POSITION pos=GetDocument()->GetFirstViewPosition();
    CView *pView;
    do
    {
       pView=GetDocument()->GetNextView(pos);
    }
    while(pView);
```

II-D-3. Amélioration du traitement :

Pour finir il sera plus aisé de faire une fonction plus générale pour faciliter le travail de création de vue et la rattacher à un document, mais le fait d'avoir des documents Template indépendant du système de stockage des MFC casse un peu la logique mise en place.
Je vais donc modifier la classe **CMultiDocTemplateEx** ainsi que la classe d'application et profiter par la même occasion d'utiliser la classe template tableau des MFC **CArray** pour stocker les pointeurs sur document template.
Include de la classe d'application :

```
#define IDD_TPLVIEW IDD_SAMPLEMDI_FORM+1000
// CSampleMDIApp:
// Consultez SampleMDI.cpp pour l'implémentation de cette classe
class CMultiDocTemplateEx :public CMultiDocTemplate
{
public:
	CMultiDocTemplateEx( UINT nIDResource,
				    CRuntimeClass* pDocClass,
				    CRuntimeClass* pFrameClass,
				    CRuntimeClass* pViewClass,
				    UINT nID=0
				    ):
	CMultiDocTemplate(nIDResource,pDocClass,
				pFrameClass,
				pViewClass),m_pDocClass(pDocClass),
					m_pFrameClass(pFrameClass),
				      m_pViewClass(pViewClass),m_nID(nID)
	{
		m_arDocTemplateEx.Add(this);
	}
	static CMultiDocTemplateEx *GetTemplateByRuntimeClass(CRuntimeClass
*pClass);
	static CMultiDocTemplateEx *GetTemplateByIdClass(UINT nID);

	CRuntimeClass* GetDocClass(){return m_pDocClass;}
	CRuntimeClass* GetFrameClass(){return m_pFrameClass;}
	CRuntimeClass* GetViewClass(){return m_pViewClass;}

	static INT_PTR GetCount(){return m_arDocTemplateEx.GetCount();}
	static CMultiDocTemplateEx* GetAt(int i)
	{
		VERIFY(i>=0 && i<GetCount());
		return m_arDocTemplateEx.GetAt(i);
	}
	bool CreateNewView(CMultiDocTemplate *pTemplateFrom,CView *pViewDest)
	{
	CFrameWnd * pFrame =pTemplateFrom->CreateNewFrame(pViewDest-
>GetDocument(),pViewDest->GetParentFrame());
		if(!pFrame) return false;
	    InitialUpdateFrame(pFrame,pViewDest->GetDocument());
	    return true;
	}
public:
// attention necessite de mettre #include  <afxtempl.h> dans stdafx.h
	static CArray<CMultiDocTemplateEx *,CMultiDocTemplateEx *>
m_arDocTemplateEx;
	CRuntimeClass* m_pDocClass;
	CRuntimeClass* m_pFrameClass;
	CRuntimeClass* m_pViewClass;
	unsigned int   m_nID;
};
```

```
class CSampleMDIApp : public CWinApp
{
public:
	CSampleMDIApp();

private:

// Substitutions
public:
	virtual BOOL InitInstance();
	virtual int  ExitInstance();
   // ouverture d'une vue correspondant à la signature pclass (correspond a
OpenDocumentFile )
    CDocument* OpenByRuntimeClass(CRuntimeClass *pClass,LPCTSTR
lpszPathName= NULL,BOOL bMakeVisible = TRUE );
	// ouverture d'une vue correspondant à l'identifiant de fenetre nID
(correspond a OpenDocumentFile )
	CDocument* OpenByIdClass(UINT nID,LPCTSTR lpszPathName= NULL,BOOL
bMakeVisible = TRUE );
// rajoute une vue identifier par nIdTpl au document de la vue pViewDest
	bool CreateNewView(CView *pViewDest,UINT NidTpl);

// Implémentation
	afx_msg void OnAppAbout();
	DECLARE_MESSAGE_MAP()
public:
//	afx_msg void OnFileNew();

public:
	afx_msg void OnFileNew();
};
```

Les modifications de la source de la classe d'application :

```
CMultiDocTemplateEx
*CMultiDocTemplateEx::GetTemplateByRuntimeClass(CRuntimeClass *pClass)
{
  CMultiDocTemplateEx* pTemplate;
  for(int i=0;i<GetCount();i++)
  {
      pTemplate=GetAt(i);
      if(pTemplate->GetViewClass()==pClass) return pTemplate;
  }
  return NULL;
}
CMultiDocTemplateEx *CMultiDocTemplateEx::GetTemplateByIdClass(UINT nID)
{
	CMultiDocTemplateEx* pTemplate;
	for(int i=0;i<GetCount();i++)
	{
		pTemplate=GetAt(i);
		if(pTemplate->m_nID==nID) return pTemplate;
	}
	return NULL;
}
CArray<CMultiDocTemplateEx *,CMultiDocTemplateEx *>
CMultiDocTemplateEx::m_arDocTemplateEx;
```

La fonction **InitInstance** :

```
CMultiDocTemplateEx *pDocTemplate = new
CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CSampleMDIView),IDD_SAMPLEMDI_FORM);

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);

      pDocTemplate = new CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CChildFrame), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CFormViewBis),IDD_SAMPLEMDI_FORMBIS);

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);

      pDocTemplate= new CMultiDocTemplateEx(IDR_SampleMDITYPE,
                        RUNTIME_CLASS(CSampleMDIDoc),
                        RUNTIME_CLASS(CChildFrame), // frame enfant MDI
personnalisé
                        RUNTIME_CLASS(CEditView),IDD_TPLVIEW);
      // #define IDD_TPLVIEW IDD_SAMPLEMDI_FORM+1000
```

Les méthodes modifiées de la classe d'application:

```
int CSampleMDIApp::ExitInstance()
{
      // liberation du template
      delete CMultiDocTemplateEx::GetTemplateByIdClass(IDD_TPLVIEW);
      return CWinApp::ExitInstance();
}
// gestionnaires de messages pour CSampleMDIApp
CDocument* CSampleMDIApp::OpenByRuntimeClass(CRuntimeClass *pClass,LPCTSTR
lpszPathName,
   BOOL bMakeVisible )
{
    CMultiDocTemplateEx*
pTemplate=CMultiDocTemplateEx::GetTemplateByRuntimeClass(pClass);
      if(pTemplate) return pTemplate-
>OpenDocumentFile(lpszPathName,bMakeVisible);
      return NULL;
}
CDocument* CSampleMDIApp::OpenByIdClass(UINT nID,LPCTSTR lpszPathName/*=
NULL*/,BOOL bMakeVisible /*= TRUE */)
{
   CMultiDocTemplateEx*
pTemplate=CMultiDocTemplateEx::GetTemplateByIdClass(nID);
   if(pTemplate) return pTemplate-
>OpenDocumentFile(lpszPathName,bMakeVisible);
      return NULL;
}

bool CSampleMDIApp::CreateNewView(CView *pViewDest,UINT NidTpl)
{
CMultiDocTemplateEx*
pTemplate=static_cast<CMultiDocTemplateEx*>(pViewDest->GetDocument()-
>GetDocTemplate());

CMultiDocTemplateEx*
pTemplateFrom=CMultiDocTemplateEx::GetTemplateByIdClass(NidTpl);
```

```
        if(!pTemplate ||!pTemplateFrom) return false;

        return pTemplate->CreateNewView(pTemplateFrom,pViewDest);
}
void CSampleMDIApp::OnFileNew()
{
        // TODO : ajoutez ici le code de votre gestionnaire de commande
        OpenByRuntimeClass(RUNTIME_CLASS(CSampleMDIView));
}
```

L'ajout de la vue :

```
void CSampleMDIView::OnInitialUpdate()
{
        CFormView::OnInitialUpdate();
        ResizeParentToFit();

        CSampleMDIApp *TheApp=static_cast<CSampleMDIApp *>(AfxGetApp());
        TheApp->OpenByIdClass(IDD_SAMPLEMDI_FORMBIS);

        /*CMultiDocTemplateEx *pTemplate=TheApp-
>GetTemplateByIdClass(IDD_SAMPLEMDI_FORM);
        if(pTemplate)
        {
           CFrameWnd * pFrame =TheApp->m_pTemlate-
>CreateNewFrame(GetDocument(),GetParentFrame());
           pTemplate->InitialUpdateFrame(pFrame,GetDocument());
        }*/
        TheApp->CreateNewView(this,IDD_TPLVIEW);
}
```

Le code ne présente pas de difficultés majeures, vous noterez dans la fonction **CreateNewView** de la classe d'application les manipulations liées à la signature d'une classe.

La classe **CMultiDocTemplateEx** utilise la classe template **CArray** pour stocker les documents template ,vous remarquerez que l'objet est static à la classe, devenant ainsi une variable globale de la classe .
Son initialisation (obligatoire pour une donnée membre static) est faite dans le source de la classe d'application :

```
CArray<CMultiDocTemplateEx *,CMultiDocTemplateEx *>
CMultiDocTemplateEx::m_arDocTemplateEx;
```

II-D-4.Création d'une nouvelle fenêtre :

L'interface générée par défaut permet l'ouverture d'une nouvelle fenêtre (commande **ID_WINDOW_NEW**).
Pour créer une nouvelle fenêtre le traitement est similaire à celui de la fonction **CreateNewView**
Je vais donc rajouter à la classe d'application la fonction **GetWIndowNew** qui permettra d'après le pointeur sur la vue passée en argument de demander une nouvelle fenêtre de cette vue :

```
CFrameWnd *CSampleMDIApp::GetWindowNew(CView *pView)
{
CMultiDocTemplateEx* pTemplate    =static_cast<CMultiDocTemplateEx*>(pView-
>GetDocument()->GetDocTemplate());
CFrameWnd * pFrame =pTemplate->CreateNewFrame(pView->GetDocument(),pView-
>GetParentFrame());
    if(!pFrame) return NULL;
pTemplate->InitialUpdateFrame(pFrame,pView->GetDocument());
return pFrame;
}
```

On commence par récupérer le document Template par l'objet document de la vue (pView).
On crée ensuite une nouvelle fenêtre cadre MDI fille et on initialise la vue.
Rien de nouveau.

II-D-4-a. Quelques remarques:

Les codes proposés plus haut : la classe **CMultiDocTemplateEx** etc.. ,sont fournis à titre d'exemples pour faciliter la manipulation des documents templates ils n'ont rien d'obligatoires..

Le modèle plusieurs vues sur le même document ne nous oblige pas pour autant à stocker les données communes dans le document.
Celui-ci devenant quand même un endroit privilégié puisqu'à partir de chaque vue nous avons accès au document par la fonction **GetDocument()**.

II-D-4-b. Conclusions :

Ce chapitre nous a permis d'apprendre les manipulations sur les documents Template et les différents modes de construction des vues.

Pour terminer je vais énumérer quelques fonctions intéressantes au niveau de la classe **CDocument** en plus de celles que nous avons étudiées:

Les opérations :

GetTitle	Permet d'obtenir le titre du document en cours.
IsModified	Permet de savoir si les données du document ont été modifiées Par **SetMofiedFlag**.
SetModifiedFlag	Notifie au document que les données ont été modifiées, le framework posera automatiquement la question « sauvegarde des données modifiées » lorsque la vue sera fermée.
SetTitle	Permet de modifier le titre du document en cours qui représente le plus souvent le nom du fichier ouvert par le document
UpdateAllViews	Permet de forcer le rafraichissement des vues attachées au document . Le premier paramètre pSender peut servir à spécifier la vue qui ne doit pas être rafraichie, l'argument peut être nul. Les deux autres paramètres permettent de passer des éléments à la vue rafraichie. Vous devez alors surcharger la fonction **OnUpdate** de votre classe vue.

Les fonctions pouvant être surchargées :

CanCloseFrame	Appelée par le framework lorsque la vue est fermée.
DeleteContents	Cette fonction est appelée pour nettoyer les données du document sans le détruire. Elle sera appelée par le framework chaque fois qu'il sera nécessaire de nettoyer le document. C'est donc l'endroit désigné pour placer la destruction de vos données plutôt que dans le destructeur du document.
OnOpenDocument	Appelée lorsque l'utilisateur ouvre un fichier par la commande **ID_FILE_OPEN**. Cette fonction appelle **DeleteContents** puis **Serialize** qui permet de lire ou écrire les données gérées par le document (voir le volume 1).
OnSaveDocument	Appelée par le framework si il y a une sauvegarde de données à faire. Initialement cette fonction ouvre le fichier et appelle la sérialisation du document.
OnCloseDocument	Fermeture du document, appelle **DeleteContents**

II-E-1. Validité d'un objet :

Pour établir une communication entre des vues on devra disposer d'un pointeur sur la fenêtre soit pour appeler une méthode soit pour lui poster un message.
De ce fait si vous devez stocker ce pointeur, ne perdez jamais de vue qu'il ne sera pas forcement valide au moment où vous allez l'utiliser.
En effet rien ne dit que la fenêtre n'a pas été fermée par l'utilisateur.
Ce qui implique que l'utilisation d'un pointeur sur une vue doit être très réduite dans le temps et assortie de précautions :

- Utilisation limitée dans le temps du pointeur : la plus proche possible de son acquisition.
- Vérification que la zone mémoire est toujours valide pour la signature de l'objet stocké (debug)
- Ensuite Vérification que le handle de fenêtre est toujours valide.

Pour réaliser le deuxième point nous utiliserons la fonction :

BOOL AfxIsValidAddress(const void* *lp*, **UINT** *nBytes*, **BOOL** *bReadWrite* **= TRUE);**

Cette fonction permet de vérifier que l'adresse pointée par **lp** sur **nBytes** octets est valide même si celle-ci ne correspond pas à un **new** …
Malheureusement son résultat est utilisable en debug, en release elle renvoie **true** si **lp** est différent de zéro.
Comment appliquer tout cela ?
Comme suit:

```
/*static*/ bool CSampleMDIApp::IsValidCWnd(CWnd *pWnd,int nSizeClass)
{
	if(!AfxIsValidAddress(pWnd,nSizeClass))  return false;
	if(!::IsWindow(pWnd->m_hWnd) ||
	    CWnd::FromHandle(pWnd->m_hWnd)!=pWnd) return false;
	return true;
}
```

Utilisation:

```
void CSampleMDIView::OnInitialUpdate()
{
	CFormView::OnInitialUpdate();
	ResizeParentToFit();

	CSampleMDIApp *TheApp=static_cast<CSampleMDIApp *>(AfxGetApp());
	TheApp->OpenByIdClass(IDD_SAMPLEMDI_FORMBIS);

	TheApp->CreateNewView(this,IDD_TPLVIEW);
	if(!CSampleMDIApp::IsValidCWnd(this,sizeof(CSampleMDIView)))
		AfxMessageBox("fenêtre invalide");

}
```

le code ci-dessus est un exemple d'utilisation ,mais il y a peu de chances que dans ce contexte un message d'erreur survienne ..

II-E-2. Recherche d'une fenêtre

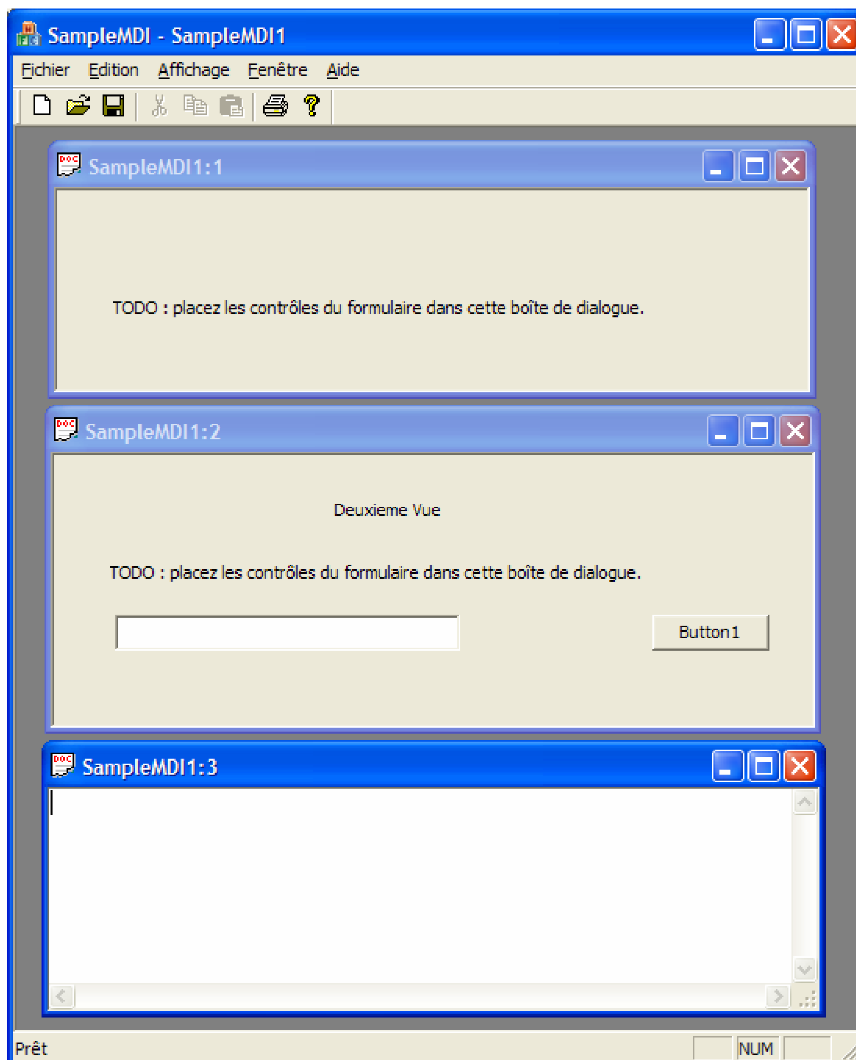
II-E-2-a. Fenêtres faisant parties du même document :

Dans un contexte MDI, le modèle document vue permet une communication aisée entre les vues d'un même document.

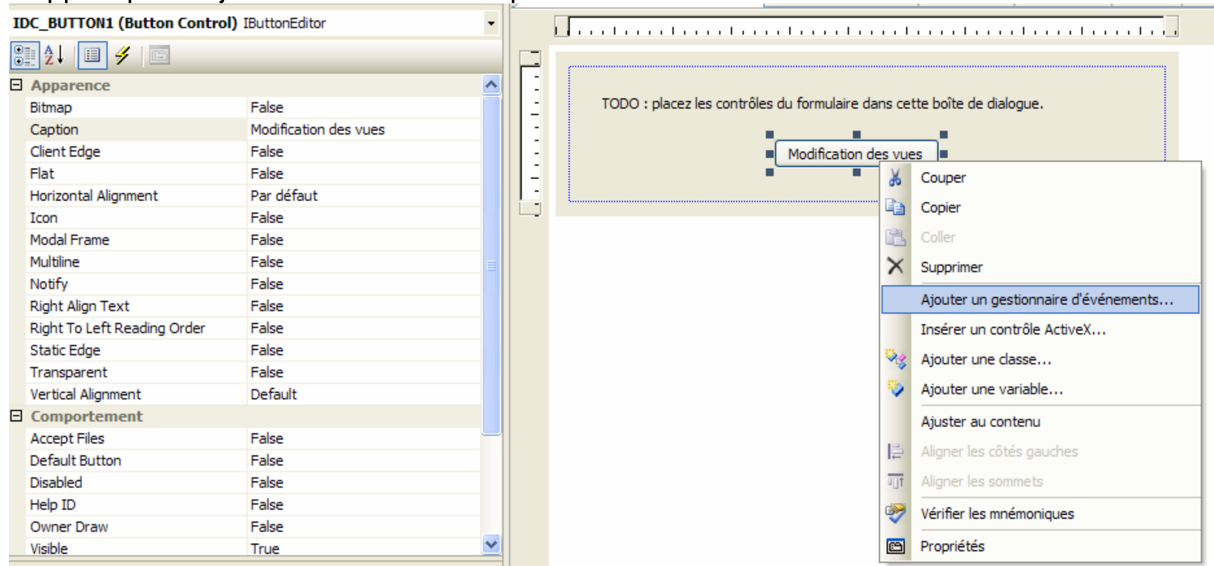
Pour illustrer le travail avec plusieurs fenêtres dans le même document je vais modifier mon exemple comme suit :
Dans la fonction **OnInitialupdate** de ma première vue, je place le code suivant :

```
void CSampleMDIView::OnInitialUpdate()
{
	CFormView::OnInitialUpdate();
	ResizeParentToFit();
	CSampleMDIApp *TheApp=static_cast<CSampleMDIApp *>(AfxGetApp());
	TheApp->CreateNewView(this,IDD_SAMPLEMDI_FORMBIS);
	TheApp->CreateNewView(this,IDD_TPLVIEW);
}
```

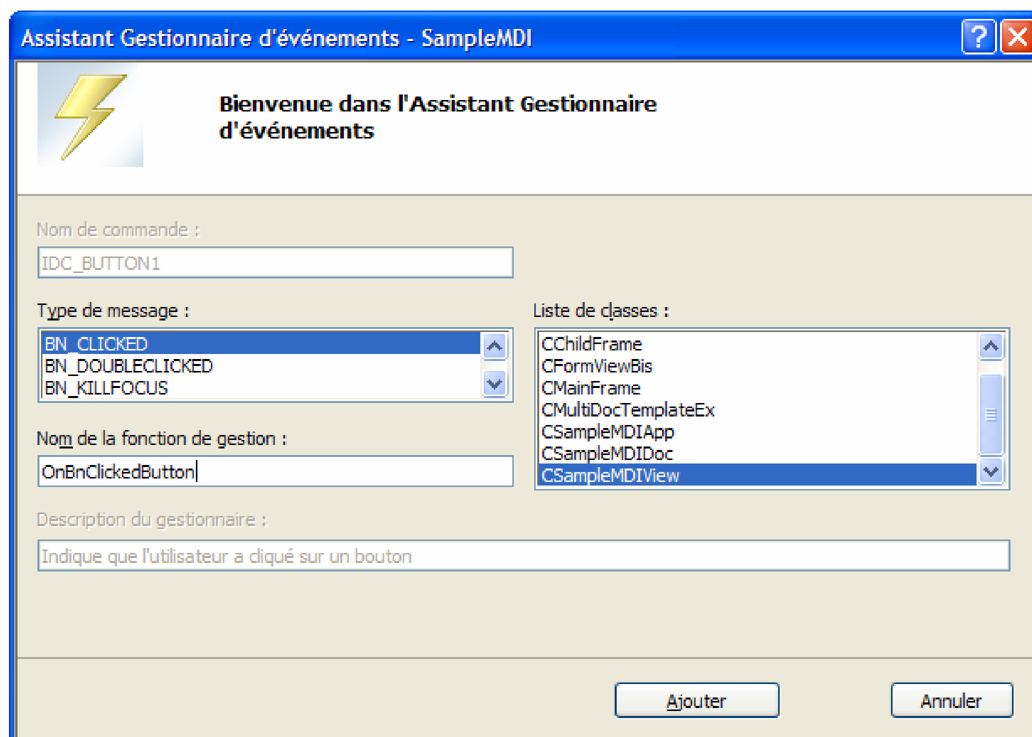
Le résultat à l'exécution donne ceci :



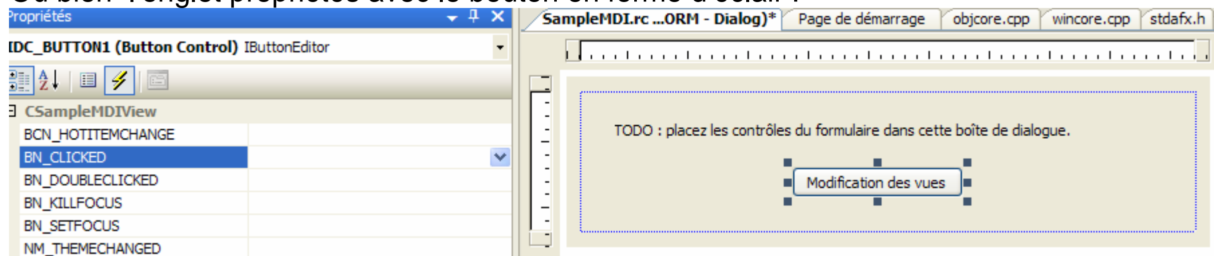
je vais maintenant modifier ma première vue et rajouter un bouton dans lequel je vais placer un traitement pour modifier la deuxième et troisième vue :
Rappel : pour rajouter la fonction de réponse au bouton on fera clic droit sur celui-ci:



On Utilisera au choix : l'option ajouter un gestionnaire d'événements



Ou bien l'onglet propriétés avec le bouton en forme d'éclair :

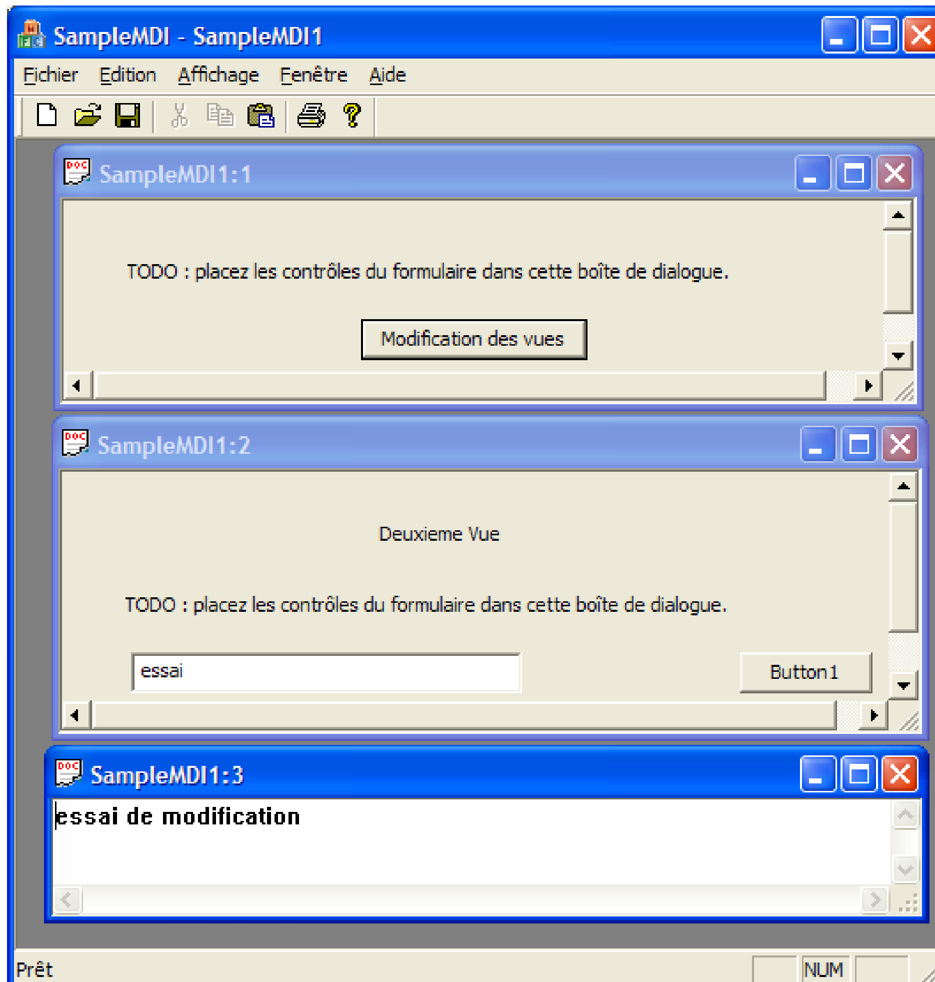


dans la fonction de réponse je place le code suivant :

```
void CSampleMDIView::OnBnClickedButton()
{
      // TODO : ajoutez ici le code de votre gestionnaire de notification
de contrôle
   POSITION pos=GetDocument()->GetFirstViewPosition();
   CView *pView;
   CFormViewBis *pViewBis=NULL;
   CEditView *pEditView=NULL;
   do
   {
      pView=GetDocument()->GetNextView(pos);
        if(pView && pView->IsKindOf(RUNTIME_CLASS(CFormViewBis)))
pViewBis=static_cast<CFormViewBis *>(pView);
        if(pView && pView->IsKindOf(RUNTIME_CLASS(CEditView)))
pEditView=static_cast<CEditView *>(pView);
   }
   while(pView);
   if(pViewBis) pViewBis->SetDlgItemText(IDC_EDIT1,"essai");
   if(pEditView) pEditView->SetWindowText("essai de modification");
}
```

Je parcours l'ensemble des fenêtres du document, quand je détecte la bonne classe j'affecte le pointeur correspondant en conséquence.

Ensuite je modifie le contenu des deux fenêtres .
Le résultat est le suivant :



pour voir l'utilité de la fonction **IsValidCWnd** ,je vais modifier mon code pour mémoriser les pointeurs de ces vues dans la fonction **OnInialUpdate** :

```
void CSampleMDIView::OnInitialUpdate()
{
	CFormView::OnInitialUpdate();
	ResizeParentToFit();

	CSampleMDIApp *TheApp=static_cast<CSampleMDIApp *>(AfxGetApp());
	TheApp->CreateNewView(this,IDD_SAMPLEMDI_FORMBIS);
	TheApp->CreateNewView(this,IDD_TPLVIEW);

   POSITION pos=GetDocument()->GetFirstViewPosition();
   CView *pView;
   do
   {
      pView=GetDocument()->GetNextView(pos);
        if(pView && pView->IsKindOf(RUNTIME_CLASS(CFormViewBis)))
m_pViewBis=static_cast<CFormViewBis *>(pView);
        if(pView && pView->IsKindOf(RUNTIME_CLASS(CEditView)))
m_pEditView=static_cast<CEditView *>(pView);
   }
   while(pView);

}
```

les deux pointeurs sont maintenant déclarés dans la classe **CSampleMDIView** et initialisés à NULL dans le constructeur de la classe.

Le code du message sur le bouton :

```
void CSampleMDIView::OnBnClickedButton()
{
	// TODO : ajoutez ici le code de votre gestionnaire de notification
de contrôle
   if(m_pViewBis) m_pViewBis->SetDlgItemText(IDC_EDIT1,"essai");
   if(m_pEditView) m_pEditView->SetWindowText("essai de modification");
}
```

testons l'application, tout fonctionne correctement au premier abord..
Fermez maintenant une des deux fenêtres supplémentaires et cliquez sur le bouton …
Le debbuger mettra une assertion sur une des deux fonctions utilisées.
Modifions le code comme suit :

```
void CSampleMDIView::OnBnClickedButton()
{
	// TODO : ajoutez ici le code de votre gestionnaire de notification
de contrôle
	if(CSampleMDIApp::IsValidCWnd(m_pViewBis,sizeof(CFormViewBis)))
		m_pViewBis->SetDlgItemText(IDC_EDIT1,"essai");
	 if(CSampleMDIApp::IsValidCWnd(m_pEditView,sizeof(CEditView)))
		m_pEditView->SetWindowText("essai de modification");
}
```

Refaites le test en fermant une des deux fenêtres.

II-E-2-b. Cas de fenêtres issues de plusieurs documents :

Il est tout à fait possible de réaliser une fonction qui renvoie un pointeur sur une fenêtre en recherchant celle-ci par la signature de sa classe.

```
/*static*/ bool CSampleMDIApp::FindWindowByRuntimeClass(CRuntimeClass
*pClass,CArray<CWnd *,CWnd*> *parCWnd/*=NULL*/)
{
	bool bFind=false;
	if(parCWnd) parCWnd->RemoveAll();

	CWinApp* pApp = AfxGetApp();
	// parcourir tous les templates
	CDocTemplate* pTemplate;
	POSITION pos = pApp->GetFirstDocTemplatePosition();
	while (pos != NULL)
	{
		pTemplate = pApp->GetNextDocTemplate(pos);
		ASSERT(pTemplate);

		// tous les documents du template.
		POSITION pos2 = pTemplate->GetFirstDocPosition();
		while (pos2)
		{
			CDocument* pDoc = pTemplate->GetNextDoc(pos2);
			ASSERT(pDoc);

			// toutes les vues du document
			POSITION pos3 = pDoc->GetFirstViewPosition();
			while (pos3 != NULL)
			{
				CView* pView = pDoc->GetNextView(pos3);
				ASSERT(pView);
				if (::IsWindow(pView->GetSafeHwnd()))
				{
					if(pView->GetRuntimeClass()==pClass)
					{
						bFind=true;
						if(parCWnd) parCWnd->Add(pView);
					}
				}
			}
		}
	}
	return bFind;
}
```

Exemple d'application : fermer toutes les fenêtres de la classe **CFormViewBis**

```
CArray<CWnd *,CWnd*> arCWnd;
if(CSampleMDIApp::FindWindowByRuntimeClass(RUNTIME_CLASS(CFormViewBis),&arC
Wnd))
   {
	for(int i=0;i<arCWnd.GetSize();i++)
	{
		// fermeture du cadre MDI.
   arCWnd[i]->GetParentFrame()->PostMessage(WM_SYSCOMMAND,SC_CLOSE,0);
	}
   }
```

II-E-2-b-1. Récupérer le cadre MDI ou la fenêtre active de l'application :

pour cela on mettra à profit les fonctions du framework que nous avons étudiées :

```
CMDIFrameWnd *pFrame = static_cast<CMDIFrameWnd*>(AfxGetMainWnd());
// recupére le cadre MDI actif
CMDIChildWnd *pChild = static_cast<CMDIChildWnd *>(pFrame-
>GetActiveFrame());

// ou CMDIChildWnd *pChild = pFrame->MDIGetActive();

// recupere la fenetre active attachée au cadre MDI actif
CView *pView = pChild->GetActiveView();
```

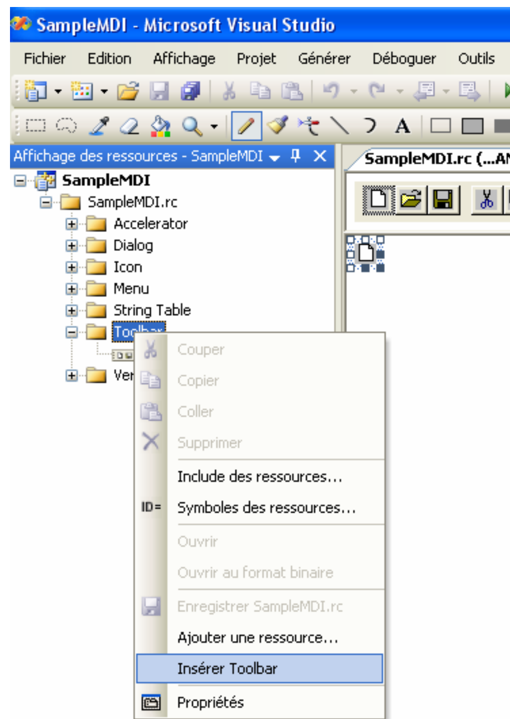
Continuons notre enrichissement de la fenêtre en mettant en place une barre d'outils.

II-F. Mise en place d'une barre d'outils dans un cadre MDI.

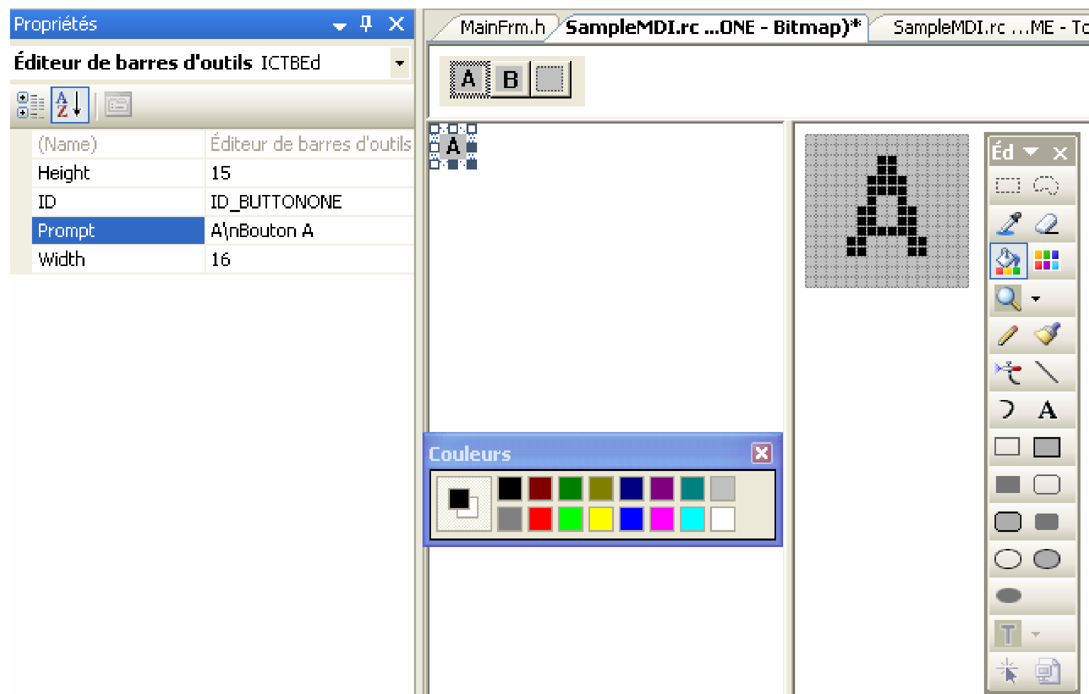
Tous les enrichissements graphiques d'une vue doivent se faire sur la fenêtre cadre MDI.
Conséquence : il faudra définir pour chaque fenêtre ayant des composants propres à la vue une classe **CMDIChildWnd** spécifique.

II-F-1. Génération de la barre d'outils :

La première étape sera de définir une barre d'outils dans les ressources :

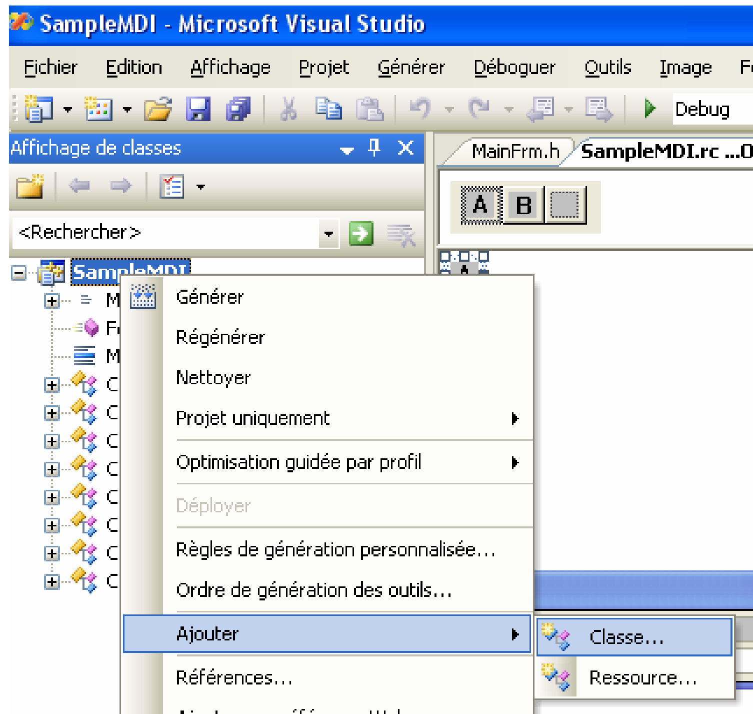


Dans le dossier Toolbar des ressources on fera clic droit insérer Toolbar.
Pour modifier l'identifiant de la barre d'outils on devra faire clic droit propriétés dessus.

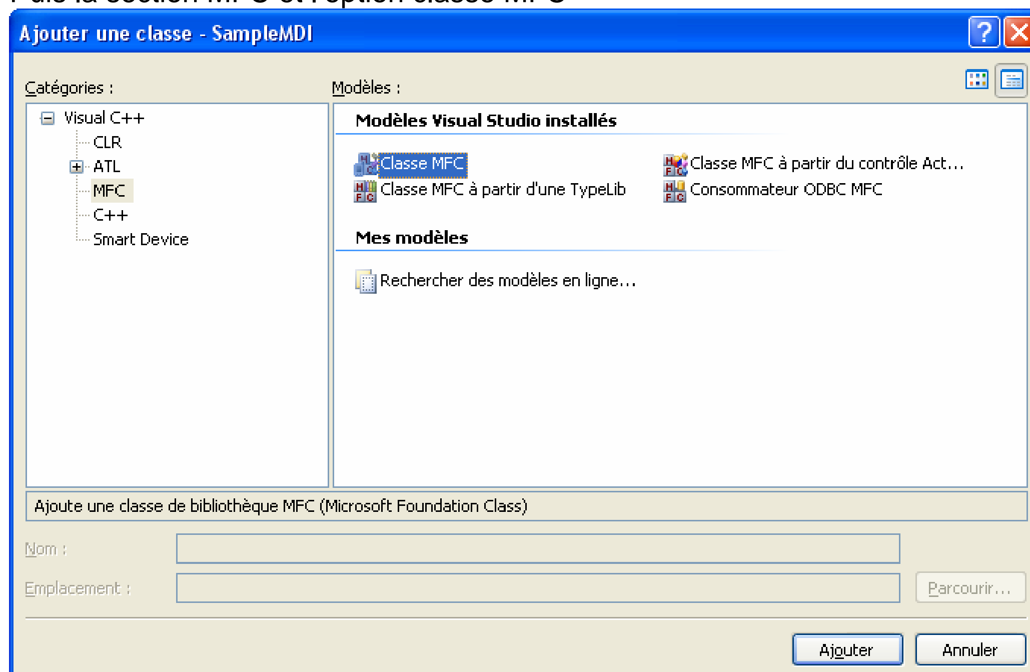


Sur chaque bouton on peut mettre le texte correspondant à l'info bulle (tooltip) du bouton et celui qui apparaitra dans la barre de statuts.
Ce texte est à définir dans la ligne prompt, en premier on indiquera le texte de l'info bulle puis celui de la barre de statuts, les deux textes étant séparés par un « \n »

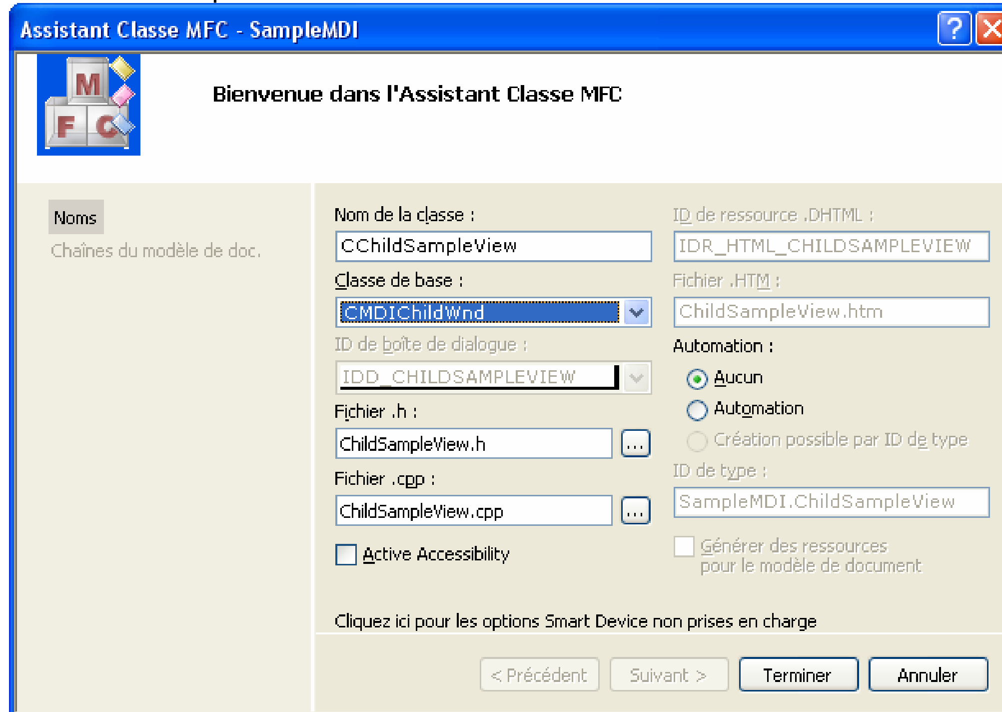
Deuxième étape : générer une classe cadre fille MDI (MDIChild) pour y placer l'initialisation de la barre d'outils.
Sur le gestionnaire de classes (classview) on fera un clic droit ajouter une classe :



Puis la section MFC et l'option classe MFC

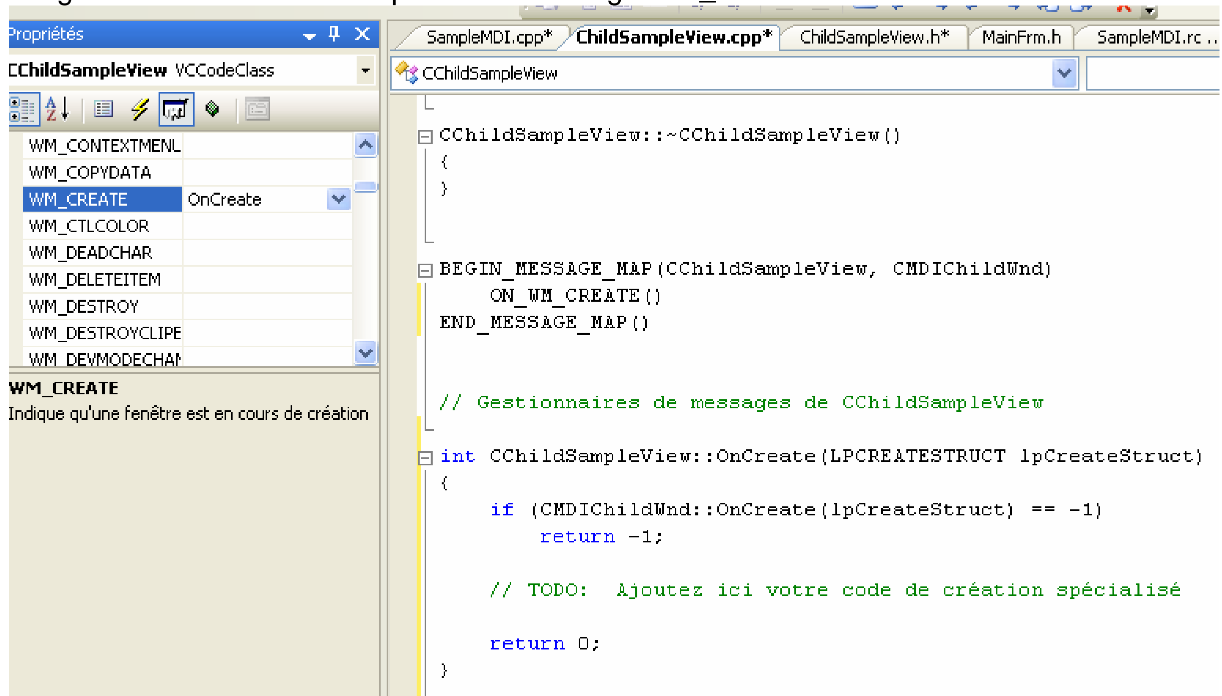


Ensuite on indique le nom de la classe et sa classe de base :



II-F-2. Modification de la classe générée :

On générera la fonction de réponse au message **WM_CREATE** :



Ensuite on modifiera le code comme suit :

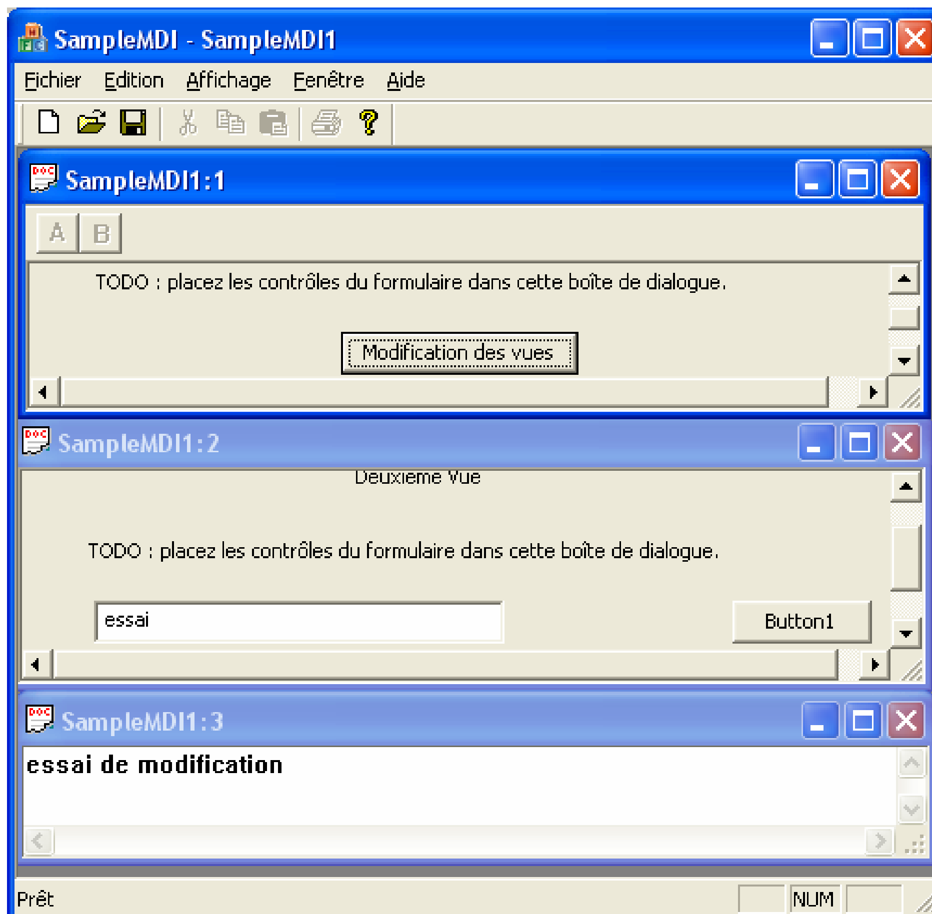
```
int CChildSampleView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1) return -1;
    // CToolBar  m_ToolBar;
    if(!m_ToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_TOP| CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
       !m_ToolBar.LoadToolBar(IDR_TOOLBARONE))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }
    return 0;
}
```

Explications :
On laisse le cadre fille s'initialiser avec l'appel de la fonction de la classe de base.
Cette ligne est bien sûr générée automatiquement avec l'assistant.
Ensuite on procède à la création graphique de la barre d'outils.
Les paramètres CBRS_xxx correspondent à des options d'affichage de la barre d'outils que l'on pourra consulter sur le site MSDN.

Troisième étape : il nous reste à modifier la fonction **InitInstance** pour prendre en compte la nouvelle classe :

```
CMultiDocTemplateEx *pDocTemplate = new
CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc)  ,
            RUNTIME_CLASS(CChildSampleView)     , // frame enfant MDI
personnalisé
            RUNTIME_CLASS(CSampleMDIView) ,IDD_SAMPLEMDI_FORM);

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);
```

Voila il nous reste plus qu'à tester tout cela :



La barre apparaît bien sur la vue principale.

II-F-3. Ajouter des fonctions de réponses à l'action des boutons :

Les boutons sont grisés car nous n'avons pas traité les commandes associées.

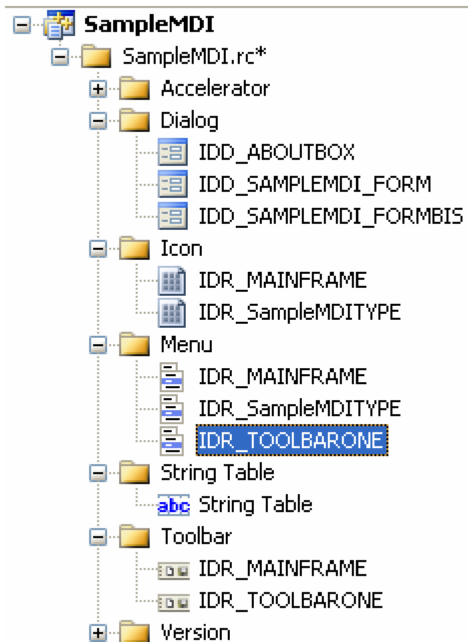
Une petite parenthèse à ce sujet l'environnement de Visual 2005 comporte de nombreuses différences de comportement par rapport à Visual 6.0.

En l'occurrence ici, si j'essaye d'intercepter les commandes correspondant aux identifiants de ma nouvelle barre d'outils avec l'assistant, et bien ce n'est pas possible.

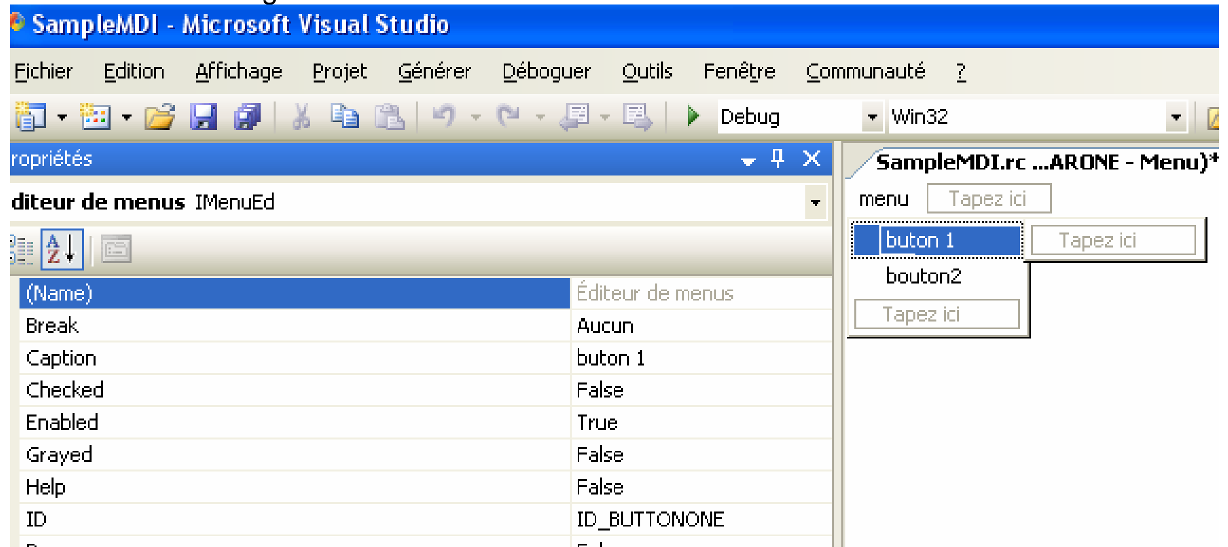
Visual 2005 considère à tort que seules les commandes issues des menus sont visibles !
Ce n'était pas le cas avec Visual 6.0.

Ce qui va nous obliger dans le cas présent à créer un menu fictif en utilisant les identifiants issus de notre barre d'outils pour pouvoir utiliser l'assistant, je trouve ça fortement regrettable !

II-F-3-a. Insertion du menu pour correspondance avec la barre d'outils :

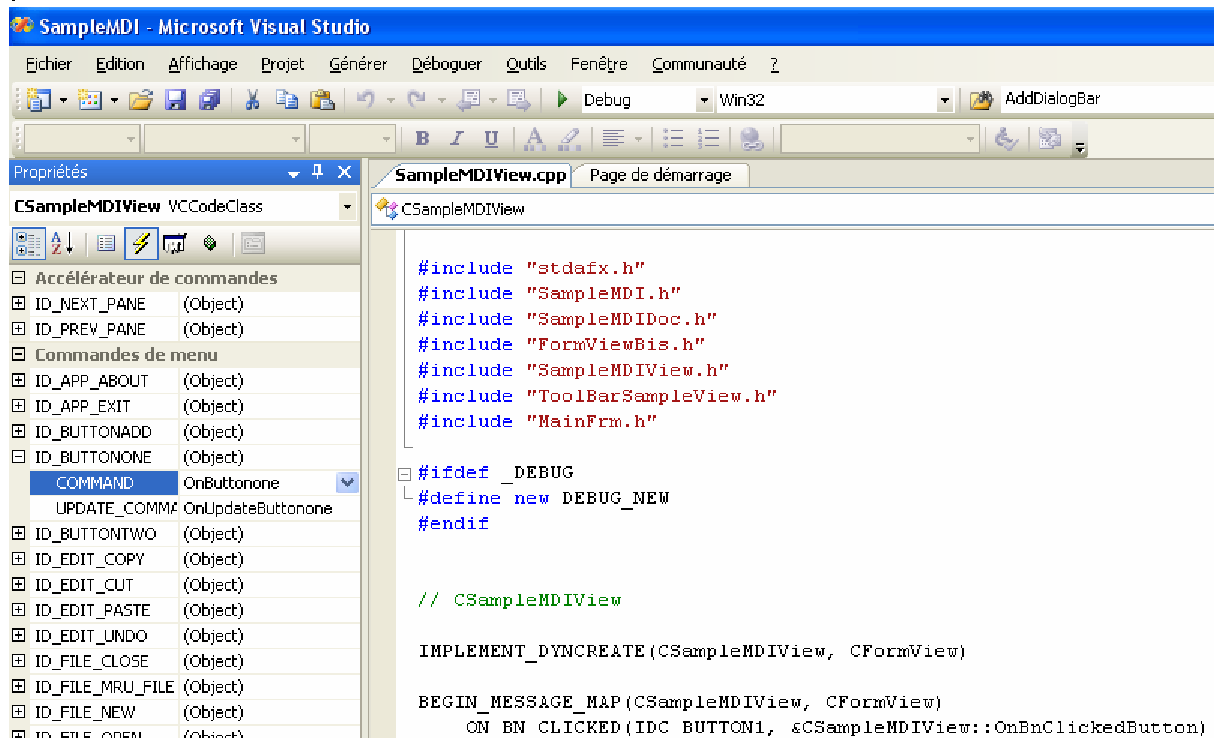


Association d'une ligne de menu à un identifiant du bouton de notre barre d'outils :



Ceci fait, nous pouvons maintenant générer les fonctions associées à ces commandes

.



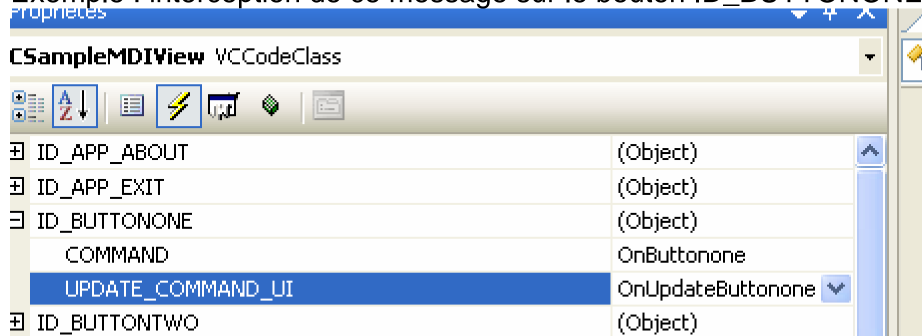

```
void CSampleMDIView::OnButtonone()
{
	// TODO : ajoutez ici le code de votre gestionnaire de commande
	AfxMessageBox("Bouton A");
}

void CSampleMDIView::OnButtontwo()
{
	// TODO : ajoutez ici le code de votre gestionnaire de commande
	AfxMessageBox("Bouton B");
}
```

Le programme peut être à nouveau lancé et testé.

II-F-4. Gestion de l'activité du bouton :

Le traitement d'une commande de bouton (ou de menu) s'effectuant avec un message **ON_COMMAND** le contrôle de son activité se fera par le message **ON_UPDATE_COMMAND_UI**

Exemple : interception de ce message sur le bouton ID_BUTTONONE




```
void CSampleMDIView::OnButtonone()
{
	// TODO : ajoutez ici le code de votre gestionnaire de commande
	AfxMessageBox("Bouton A");
}

void CSampleMDIView::OnButtontwo()
{
	// TODO : ajoutez ici le code de votre gestionnaire de commande
	m_bActif=!m_bActif;
}

void CSampleMDIView::OnUpdateButtonone(CCmdUI *pCmdUI)
{
	// TODO : ajoutez ici le code du gestionnaire d'interface utilisateur
de mise à jour des commandes
	  pCmdUI->Enable(m_bActif); // si m_bActif=false bouton grisé sinon
actif.
}
```

J'ai déclaré dans la classe **CSampleMDIView** une variable booléenne **m_bActif** initialisée à true dans le constructeur.
Lorqu'on appuie sur le bouton deux, le bouton un commute à l'état passif/actif suivant la valeur de **m_bActif.**

II-F-5. Personnaliser la barre d'outils :

Il est parfois utile d'insérer des contrôles dans une barre d'outils, dans l'exemple qui va suivre je vais décrire la méthode pour insérer une **CComboBox** entre nos deux boutons de la barre d'outils de notre fenêtre.
Pour commencer il faudra générer avec l'aide de l'assistant une classe dérivée de la classe **CToolBar** et que nous nommerons **CToolBarSampleView.**

Ensuite nous lui rajouterons une méthode **OnInitToolBar()** :

```
bool CToolBarSampleView::OnInitToolBar()
{
	TBBUTTON button;

	button.iBitmap=-1;
	button.idCommand=0;
	button.fsStyle=TBSTYLE_SEP;
	button.dwData=0;
	button.iString=-1;
	int n=GetToolBarCtrl().GetButtonCount();
	GetToolBarCtrl().InsertButton(n-1,&button);
	GetToolBarCtrl().InsertButton(n-1,&button);
	SetButtonInfo(n-1,1,TBBS_SEPARATOR,100);

	CRect rect;
	GetItemRect(n-1,&rect);
	rect.top=2;
	rect.bottom=rect.top+100;

	if(!m_ComboBox.Create(CBS_DROPDOWNLIST | WS_VISIBLE |
WS_TABSTOP,rect,this,1))
	{
	TRACE0("Failed to create combobox\n");
	return false;	// fail to create
	}

	m_ComboBox.AddString("item 0");
	m_ComboBox.AddString("item 1");
	m_ComboBox.AddString("item 2");
	m_ComboBox.AddString("item 3");

	return true;
}
```

On commence par insérer deux séparateurs avec la structure **TBBUTTON** et la fonction **InsertButton**.
Celle-ci insert un élément à gauche du numéro d'index passé en argument.

On fixe ensuite les infos sur le bouton à savoir son identifiant ici 1 et sa largeur en pixels :100.

Note : dans le cas d'un séparateur la fonction **SetButtonInfo** accepte la largeur en pixels au lieu de l'indice de l'image.
Voir documentation MSDN pour plus de détails.

On crée la **ComboBox**.
Et pour finir on l'alimente par **AddString**.

Ensuite on récupère la place de l'item choisi et on lui recalcule sa hauteur.
Il faudra insérer l'include de cette classe dans celui de la classe **CChildSampleView:**
Et remplacer la classe **CToolBar** par **CToolBarSampleView** :

```
#include "ToolBarEx.h"
// Frame CChildSampleView

class CChildSampleView : public CMDIChildWnd
{
	DECLARE_DYNCREATE(CChildSampleView)
protected:
	CChildSampleView();		// constructeur protégé utilisé par la
création dynamique
	virtual ~CChildSampleView();
public:
CToolBarSampleView  m_ToolBar;
protected:
	DECLARE_MESSAGE_MAP()
public:
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
};
```

Dans la fonction **OnCreate** il nous reste à appeler la fonction d'initialisation de la barre d'outils.

```
int CChildSampleView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1) return -1;
    // CToolBar  m_ToolBar;
    if(!m_ToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_TOP
            | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
         !m_ToolBar.LoadToolBar(IDR_TOOLBARONE))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }
      m_ToolBar.OnInitToolBar();
    return 0;
}
```

Pour manipuler les notifications de la **CComboBox** on pourra rajouter manuellement la notification **OnComboChange** sur le message map de notre fenêtre vue ce qui est le plus pratique pour nous :

```

// CSampleMDIView

IMPLEMENT_DYNCREATE(CSampleMDIView, CFormView)

BEGIN_MESSAGE_MAP(CSampleMDIView, CFormView)
	ON_BN_CLICKED(IDC_BUTTON1, &CSampleMDIView::OnBnClickedButton)
	ON_COMMAND(ID_BUTTONONE, &CSampleMDIView::OnButtonone)
	ON_COMMAND(ID_BUTTONTWO, &CSampleMDIView::OnButtontwo)
```

```
      ON_UPDATE_COMMAND_UI(ID_BUTTONONE,
&CSampleMDIView::OnUpdateButtonone)
      ON_CBN_SELCHANGE(1,OnComboChange)
END_MESSAGE_MAP()
```

```
void CSampleMDIView::OnComboChange()
{
      //
      CChildSampleView
*pChild=static_cast<CChildSampleView*>(GetParentFrame());
      int nSel=pChild->m_ToolBar.m_ComboBox.GetCurSel();
      if(nSel!=CB_ERR)
      {
            CString str;
            pChild->m_ToolBar.m_ComboBox.GetLBText(nSel,str);
            TRACE("\nCombo Sel:%s",(const char *)str);
      }
}
```

Vous vous posez la question pourquoi avoir fait une classe dérivée de **CToolBar** pour rajouter un contrôle dans notre barre d'outils ?
Et bien c'est parce que j'ai dans l'idée de vous proposer un traitement qui nous permettra de nous affranchir de la création d'une classe personnalisée de **CMDIChildWnd** pour l'ajout d'une barre d'outils.
Mais nous verrons cela plus tard, et tout deviendra plus clair …

II-F-5-1. Insérer des boutons dynamiquement dans la barre d'outils :

L'insertion d'un bouton dynamiquement dans une barre d'outils cause quelques soucis de mise à jour.
Le code qui suit est une proposition de code permettant de gérer correctement ce problème :

```
///////////////////////////////////////////////////////////////////////////
//
// CToolBarEx window
class CToolBarEx : public CToolBar
{
// Construction
public:
CToolBarEx();

// Attributes
public:
    void  SetSizes(SIZE sizeButton, SIZE sizeImage)
    {
        GetToolBarCtrl( ).SetButtonSize(sizeButton);
        if(GetToolBarCtrl( ).SetBitmapSize(sizeImage))
        m_sizeImage=sizeImage;
    }

   inline void InitSizes(CSize cSizeBitmapSize)
    {
        m_SizeButton=cSizeBitmapSize;

SetSizes(CSize(cSizeBitmapSize.cx+7,cSizeBitmapSize.cy+6),cSizeBitmapSize);
        GetToolBarCtrl().AutoSize();
    }
    bool  SetButtonWidth(UINT nMinWidth,UINT nMaxWidth);
```

```
    CSize GetButtonsWidth() const;
    CSize GetButtonsSize()  const;
```

```
    bool AddButtonToolBar(int nIndexPos,int nidCommand,UINT nIdbitmap,int
istring=0);
    bool DeleteButtonToolBar(int nIndexPos);

    void RedrawToolBar(BOOL bRecalcLayout=TRUE,BOOL bOnlyFrame=FALSE);
    void RedrawButton(int nIndex);
    void UpdateSizes();

    void ReCalcDynamicLayout(CRect rect,int nIndexPos=-1);

// Attributes
private:
    CSize m_sizeMiniMaxi;
    CSize m_SizeButton;
    CSize m_sizeImage;

// Operations
public:

// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CToolBarEx)
//}}AFX_VIRTUAL

// Implementation
public:
virtual ~CToolBarEx();

// Generated message map functions
protected:
//{{AFX_MSG(CToolBarEx)
// NOTE - the ClassWizard will add and remove member functions here.
//}}AFX_MSG

DECLARE_MESSAGE_MAP()
};
```

Le code:

```
///////////////////////////////////////////////////////////////////////////
//
// CToolBarEx

CToolBarEx::CToolBarEx()
{
    m_SizeButton.cx=16;
    m_SizeButton.cy=15;
}

CToolBarEx::~CToolBarEx()
{
}


BEGIN_MESSAGE_MAP(CToolBarEx, CToolBar)
//{{AFX_MSG_MAP(CToolBarEx)
// NOTE - the ClassWizard will add and remove mapping macros here.
//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//
// CToolBarEx message handlers
bool CToolBarEx::SetButtonWidth(UINT nMinWidth,UINT nMaxWidth)
{
    ASSERT(::IsWindow(GetSafeHwnd()));
    ASSERT(nMaxWidth-m_sizeImage.cx>=7);
    if(SendMessage(TB_SETBUTTONWIDTH, 0, MAKELPARAM(nMinWidth, nMaxWidth)))
    {
        m_sizeMiniMaxi.cx=nMinWidth;
        m_sizeMiniMaxi.cy=nMaxWidth;
        return true;
    }
    return false;
}
//-----------------------------------------------------------------
CSize CToolBarEx::GetButtonsWidth() const
{
    ASSERT(::IsWindow(GetSafeHwnd()));
    return m_sizeMiniMaxi;
}
//-----------------------------------------------------------------
CSize CToolBarEx::GetButtonsSize() const
{
    ASSERT(::IsWindow(GetSafeHwnd()));
    DWORD result=(DWORD)::SendMessage(m_hWnd,TB_GETBUTTONSIZE,0,(LPARAM)0);
    return CSize(LOWORD(result),HIWORD(result));
}
//-----------------------------------------------------------------
void CToolBarEx::RedrawToolBar(BOOL bRecalcLayout/*=TRUE*/,
   BOOL bOnlyFrame/*=FALSE*/)
{
    ASSERT(::IsWindow(GetSafeHwnd()));

    if(!IsWindowVisible())return;

    if(bRecalcLayout)
    {
        CWnd *pParent=GetToolBarCtrl( ).GetParent();
        CFrameWnd* pFrameWnd=(CFrameWnd *)pParent->GetParent();
        if(pFrameWnd!=NULL)
        {
            pFrameWnd->RecalcLayout();
            for(int nIndex=0; nIndex<GetToolBarCtrl( ).GetButtonCount();
nIndex++)
            {
                RedrawButton(nIndex);
                CRect rect;
                GetItemRect(nIndex,rect);
                ValidateRect(rect);
            }
        }
    }
    else
    {
        if(!bOnlyFrame)
        {
            GetToolBarCtrl( ).RedrawWindow(NULL,NULL,
```

```
RDW_INVALIDATE|RDW_UPDATENOW|RDW_ERASE|RDW_FRAME|RDW_ALLCHILDREN);
        }
    }
    if(bOnlyFrame)
    {
        GetToolBarCtrl( ).SetWindowPos(NULL,0,0,0,0,
        SWP_NOZORDER|SWP_NOMOVE|SWP_NOSIZE|SWP_DRAWFRAME);
    }
}
//------------------------------------------------------------------
void CToolBarEx::RedrawButton(int nIndex)
{
    ASSERT(::IsWindow(GetSafeHwnd()));

    if(nIndex<0 || nIndex>GetToolBarCtrl().GetButtonCount())
    {
        return;
    }
    CRect rect;
    GetToolBarCtrl( ).GetItemRect(nIndex,rect);
    GetToolBarCtrl( ).RedrawWindow(rect);
}
//------------------------------------------------------------------
void CToolBarEx::UpdateSizes()
{
    SetSizes(GetButtonsSize(),m_sizeImage);
    GetToolBarCtrl().AutoSize();
}
//------------------------------------------------------------------
bool CToolBarEx::DeleteButtonToolBar(int nIndexPos)
{
    CRect rect;
    GetToolBarCtrl( ).GetWindowRect(&rect);

    if(!GetToolBarCtrl( ).DeleteButton(nIndexPos)) return false;

    // resize window
    rect.right-=(m_SizeButton.cx+7);
    SetWindowPos(NULL,0,0,rect.Width(),rect.Height(),
    SWP_NOMOVE|SWP_NOZORDER|SWP_DRAWFRAME|SWP_FRAMECHANGED);

    ReCalcDynamicLayout(rect);
    return true;
}
//------------------------------------------------------------------
void CToolBarEx::ReCalcDynamicLayout(CRect rect,int nIndexPos/*=-1*/)
{
    SetWindowPos(NULL,0,0,rect.Width(),rect.Height(),
        SWP_NOMOVE|SWP_NOZORDER|SWP_DRAWFRAME|SWP_FRAMECHANGED);

    if(IsFloating())
    {
        CPoint newPos(0,0);
        ClientToScreen(&newPos);
        CRect rcNew;
        //  GetToolBarCtrl().SetRows(GetToolBarCtrl().GetRows(),TRUE,
&rcNew);

        CalcDynamicLayout(rect.Width(),LM_HORZ | LM_COMMIT);
```

```
        CWnd *pParent=GetToolBarCtrl( ).GetParent();
        CFrameWnd* pFrameWnd=(CFrameWnd *)pParent->GetParent();

        if(pFrameWnd)
            pFrameWnd->FloatControlBar(this, newPos,CBRS_ALIGN_TOP |
CBRS_SIZE_DYNAMIC);
    }

    RedrawToolBar();
    if(nIndexPos>0)RedrawButton(nIndexPos);

}
//-------------------------------------------------------------------
bool CToolBarEx::AddButtonToolBar(int nIndexPos,int nidCommand,UINT
nIdbitmap,int istring/*=0*/)
{
    BOOL bok;

    CRect rect;
    GetToolBarCtrl( ).GetWindowRect(&rect);

    TBBUTTON Buttons;
    GetToolBarCtrl( ).AddBitmap(1,nIdbitmap);
    Buttons.iBitmap=nIndexPos;
    Buttons.idCommand=nidCommand;
    Buttons.fsState=TBSTATE_ENABLED;
    Buttons.fsStyle=TBSTYLE_BUTTON;
    Buttons.dwData=0;
    Buttons.iString=istring;
    bok=GetToolBarCtrl( ).AddButtons(1,&Buttons);

    SetButtonWidth(0,m_SizeButton.cx+7);

    InitSizes(m_SizeButton);

    // resize window
    rect.right+=(m_SizeButton.cx+7);

    ReCalcDynamicLayout(rect,nIndexPos);
    return (bok?true:false);
}
```

II-F-5-1-a. Utilisation :

Pour utiliser notre nouvelle classe nous allons modifier notre classe **CToolBarSampleView**
Il nous faut juste modifier le parent, à la place de **CToolBar** on mettra **CToolBarEx.**

II-F-5-1-b. Initialisations dans la classe CChildSampleView:

```
int CChildSampleView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
      if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1) return -1;

        // portion de code relatif à la CToolbarEx
        if (!m_ToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_ALIGN_TOP| CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
!m_ToolBar.LoadToolBar(IDR_TOOLBARONE))
{
TRACE0("Failed to create toolbar\n");
return -1;      // fail to create
}
    m_ToolBar.OnInitToolBar();

    m_ToolBar.InitSizes(CSize(16,15));

    m_ToolBar.SetWindowText(_T("Toolbar"));
    m_ToolBar.UpdateSizes();
    m_ToolBar.SetBarStyle(m_ToolBar.GetBarStyle() | CBRS_TOOLTIPS |
CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

    // TODO: Delete these three lines if you don't want the toolbar to
    //  be dockable
    m_ToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_ToolBar);
```

Notez l'utilisation du style **CBRS_SIZE_DYNAMIC** ,
l'utilisation de la fonction **InitSizes** pour spécifier la taille des boutons , et la fonction **UpdateSizes()** pour la prise en compte des modifications.

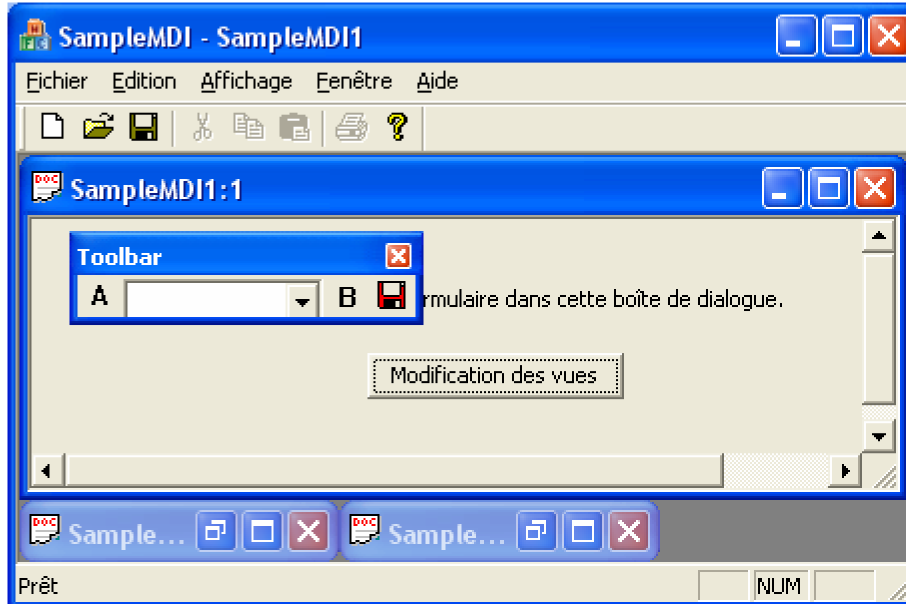
Nous allons maintenant rajouter à notre barre d'outils un troisième bouton à l'emplacement 2 :

```
// Rajout d'un bouton par un bitmap IDB_BITMAP1 a la position 2 associé a
la commande ID_BUTTONADD
CChildSampleView * pChild=static_cast<CChildSampleView
*>(GetParentFrame());
   pChild->m_ToolBar.AddButtonToolBar(2,ID_BUTTONADD,IDB_BITMAP1);

// del d'un bouton
CChildSampleView * pChild=static_cast<CChildSampleView
*>(GetParentFrame());
pChild->m_ToolBar.DeleteButtonToolBar(2);
```

Dans le code ci-dessus, l'ajout d'un bouton prend en charge le recalcule et le dessin de la **CToolBar** qu'elle soit ancrée ou non.
Deux méthodes sont utilisées suivant que la **CToolBar** est ancrée ou non, le détail du traitement est visible dans la fonction **ReCalcDynamicLayout.**

La commande **ID_BUTTONADD** a été rajoutée dans notre menu factice afin de pouvoir utiliser l'assistant pour générer un message en réponse à la commande.

II-F-5-1-c. Le résultat :



II-F-5-2. Barre d'outils flottantes et positionnement :

Comme nous venons de le voir les barres d'outils peuvent être flottantes et accrochables il suffit de rajouter ces lignes de code :

```
    m_ToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_ToolBar);
```

Le positionnement initial de la barre d'outils est précisé lors de sa création, en l'occurrence ici avec l'attribut **CBRS_ALIGN_TOP** .

Il arrive parfois que l'on dispose de plusieurs barres d'outils sur un fenêtre et que l'on souhaite les disposer à notre convenance, par exemple deux barres d'outils positionnées côte à côte .

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
                return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_ALIGN_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
        {
                TRACE0("Failed to create toolbar\n");
                return -1;      // fail to create
        }

        if (!m_wndStatusBar.Create(this) ||
                !m_wndStatusBar.SetIndicators(indicators,
                  sizeof(indicators)/sizeof(UINT)))
        {
                TRACE0("Failed to create status bar\n");
                return -1;      // fail to create
        }
        m_wndToolBar.SetWindowText(_T("Toolbar"));
```

```
        m_wndToolBar.SetBarStyle(m_wndToolBar.GetBarStyle() | CBRS_TOOLTIPS
| CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

// deuxieme toolbar .
    if (!m_wndToolBar2.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_ALIGN_TOP| CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar2.LoadToolBar(IDR_MAINFRAME))
        {
                TRACE0("Failed to create toolbar\n");
                return -1;      // fail to create
        }

        m_wndToolBar2.SetWindowText(_T("Toolbar2"));
        m_wndToolBar2.SetBarStyle(m_wndToolBar.GetBarStyle() |
CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

        // TODO: Delete these three lines if you don't want the toolbar to
        //  be dockable
        m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
         m_wndToolBar2.EnableDocking(CBRS_ALIGN_ANY);

       EnableDocking(CBRS_ALIGN_ANY);
       DockControlBar(&m_wndToolBar); //placement 1 toolbar

       CRect RectOne,RectTwo;
       RecalcLayout();

    m_wndToolBar.GetWindowRect(&RectOne);

    m_wndToolBar2.GetWindowRect(&RectTwo);
    int nWidth=RectTwo.Width();
    int nHeight=RectTwo.Height();

// calcul emplacement de la toolbar2 a droite de la 1 toolbar
    RectTwo.left=RectOne.right;
    RectTwo.right=RectTwo.left+nWidth;
    RectTwo.top=RectOne.top;
    RectTwo.bottom=RectTwo.top+nHeight;

// placement final.
    DockControlBar(&m_wndToolBar2,(UINT)0,RectTwo);
    RecalcLayout();

        return 0;
}
```

Les deux barres d'outils sont initialisées, ensuite je récupère leurs positions, calcule la position de la deuxième barre, et pour finir j'utilise la fonction **DockControlBar** pour indiquer la nouvelle position de la barre d'outils.

II-F-5-3. Boutons à états :

Il est parfois interressant de pouvoir disposer d'un bouton à état sur une barre d'outils.
Lorsque le bouton est pressé il reste enfoncé et affiche une autre bitmap, s'il est pressé à nouveau il réaffiche le bitmap d'origine.
Pour traiter ce cas, j'ai rajouté des fonctions supplémentaires à la classe **CToolBarEx.**

```
//------------------------------------------------------------------
int CToolBarEx::UpdateButton(int nIDBt)
{
      struct CUSTOMBT custom;

      if(!m_mapCustomBt.Lookup(nIDBt,custom)) return -1;

      int rnImage;
      if(GetButtonImage(nIDBt,rnImage))
    {

      rnImage=(rnImage==custom.nIndiceNew?custom.nIndiceOrg:custom.nIndiceN
ew);

        SetButtonImage(nIDBt,rnImage);

            // met le bouton enfoncé dans le cas du deuxieme etat.
            SetButtonPressed(nIDBt,rnImage==custom.nIndiceNew);
            return (rnImage==custom.nIndiceNew);
    }
      return -1;
}
//------------------------------------------------------------------
void CToolBarEx::SetButton(int nIDBt,UINT nBitmapID,const char *szNextText)
{
      struct CUSTOMBT custom;

      int nNumButtons=GetToolBarCtrl().GetImageList()->GetImageCount();
      custom.nIndiceNew=AddBitmapToolBar(nNumButtons,nBitmapID);

      custom.strNewTxt=szNextText;

      GetButtonImage(nIDBt,custom.nIndiceOrg);

      m_mapCustomBt.SetAt(nIDBt,custom);
}
//------------------------------------------------------------------
BOOL CToolBarEx::SetButtonImage(int nIDBt,int nImage)
{

    TBBUTTONINFO tbbi;
    tbbi.dwMask = TBIF_IMAGE;
    tbbi.cbSize = sizeof tbbi;

    BOOL b=GetToolBarCtrl().GetButtonInfo(nIDBt, &tbbi );
    if(b)
    {
        tbbi.iImage=nImage;
        b=GetToolBarCtrl().SetButtonInfo(nIDBt,&tbbi);
        GetToolBarCtrl().RedrawWindow();
    }
    return (b && b!=-1);
}
```

```
//-----------------------------------------------------------------
BOOL CToolBarEx::GetButtonImage(int nIDBt,int &rnImage)
{
   TBBUTTONINFO tbbi;
   tbbi.dwMask = TBIF_IMAGE;

   tbbi.cbSize = sizeof tbbi;

   BOOL b= GetToolBarCtrl().GetButtonInfo(nIDBt, &tbbi );

   rnImage=tbbi.iImage;


   return (b && b!=-1);
}
//-----------------------------------------------------------------
BOOL CToolBarEx::AddBitmapToolBar(int nNumButtons,UINT nBitmapID )
{
   return GetToolBarCtrl().AddBitmap(nNumButtons,nBitmapID);
}
//-----------------------------------------------------------------
int CToolBarEx::GetButtonCount()
{
    // nombre de boutons dans la toolbar separateur compris.
    int nct=GetToolBarCtrl().GetButtonCount();
    int nCount=0;
    TBBUTTON tb;
    for(int i=0;i<nct;i++)
    {
        GetToolBarCtrl().GetButton(i,&tb);
            // si l'indice commande du bouton est valide ok on a bien un
bouton.
        if(tb.idCommand) nCount++;
    }
    return nCount;
}
//-----------------------------------------------------------------
BOOL CToolBarEx::SetButtonPressed(int nIDBt,bool bPressed/*=true*/)
{
   int nState=TBSTATE_ENABLED ;
   if(bPressed) nState|=TBSTATE_PRESSED;

   return GetToolBarCtrl().SetState(nIDBt,nState);
}
```

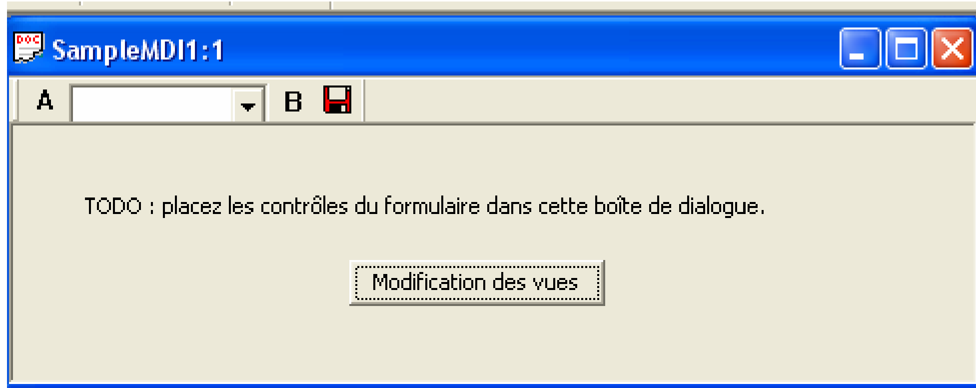
Le principe est d'insérer dans la barre d'outils le bitmap correspondant au nouvel état.
Ce que l'on pourra faire par exemple à partir de la vue dans **OnInitialUpdate** :

```
// Accès à la frame (parent de la classe fenêtre).
CChildSampleView * pChild=static_cast<CChildSampleView
*>(GetParentFrame());
// ajout du bitmap en fin de toolbar .
pChild->m_ToolBar.SetButton(ID_BUTTONADD,IDB_BITMAPNOSAVE,"Fermé");
```

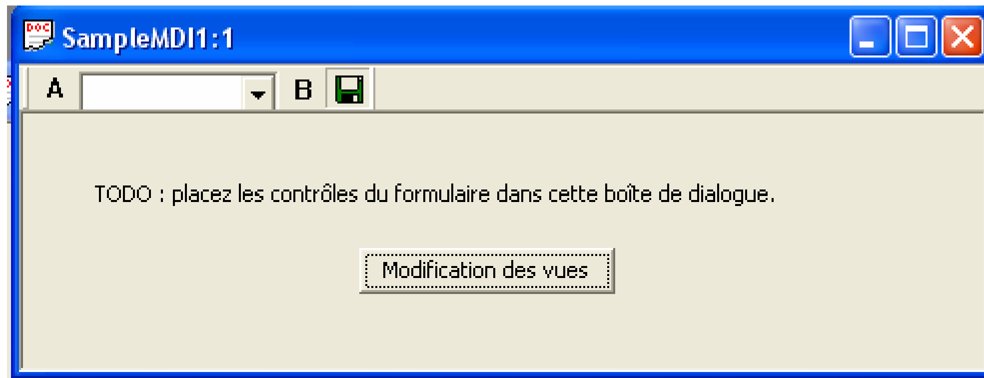
Sur le message de commande on mettra à jour le bouton comme suit:

```
CChildSampleView * pChild=static_cast<CChildSampleView
*>(GetParentFrame());
pChild->m_ToolBar.UpdateButton(ID_BUTTONADD);
```

II-F-5-3-a. Le résultat :



Le deuxième état, le bouton reste pressé :

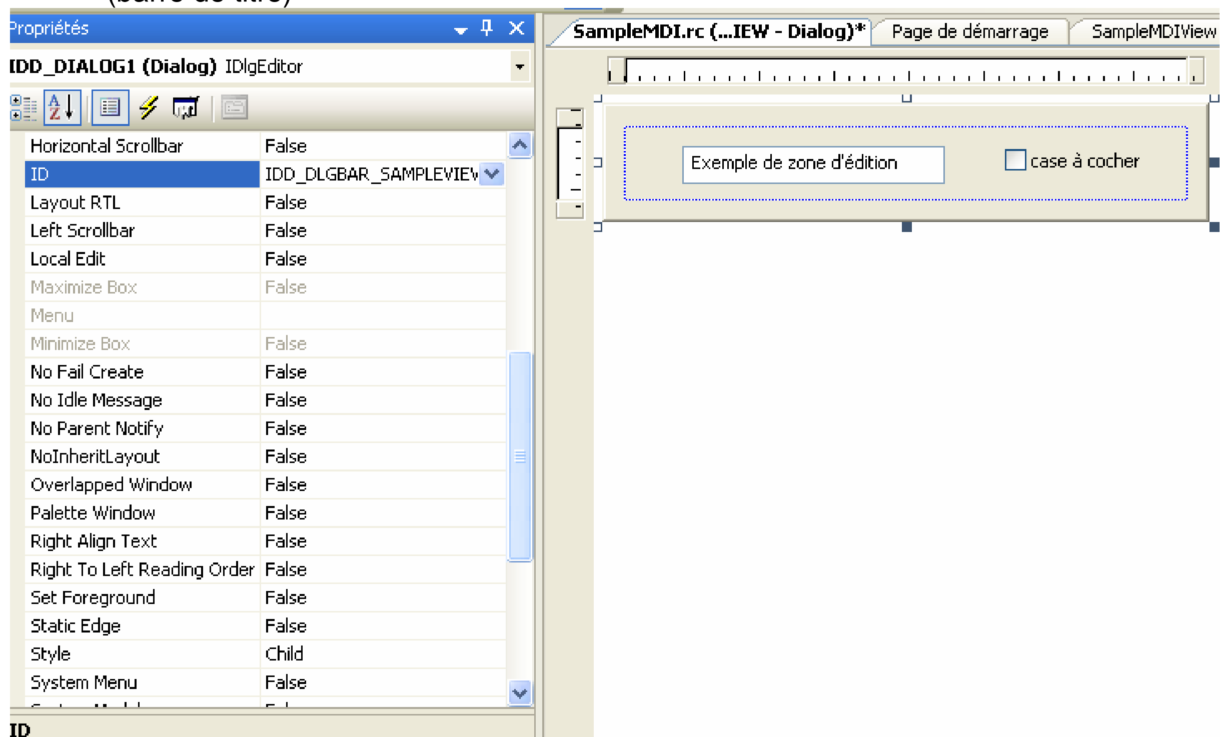


II-G. Mise en place d'une barre de dialogue dans un cadre MDI.

Les barres de dialogues sont semblables à des boites de dialogues, elles permettent de disposer un ensemble de contrôle, sur sa surface de la même manière qu'une boite de dialogue standard.
Vous l'avez compris maintenant, tous les ajouts visant à enrichir une vue sont à effectuer dans la classe cadre MDI.

Pour mettre en place une barre de dialogue Il faudra :

- Créer une nouvelle boite de dialogue dans l'éditeur de ressources
  Changer son nom en IDD_DLGBAR_xxxx pour bien les distinguer des boîtes de dialogues classiques.
  Mettre le style : « Child » à la place de « popup » et enlever l'option « title bar » (barre de titre)



- Générer avec l'aide de l'assistant une nouvelle classe dérivée de **CDialog** (on n'a pas le choix, il n'y a pas de **CDialogBar** dans le sélecteur de classes !).
  On fera donc un clic droit sur la fenêtre pour appeler l'option Ajouter une classe.

Assistant Classe MFC - SampleMDI

Bienvenue dans l'Assistant Classe MFC

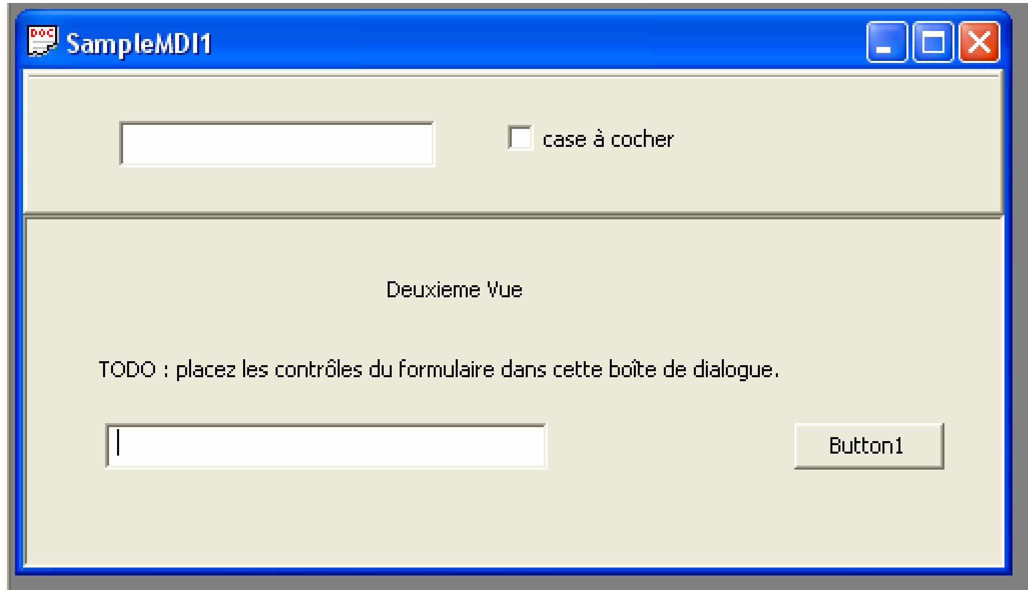
Il nous faudra donc modifier le constructeur de la classe générée comme suit :

```
class CDlgBarSampleView : public CDialogBar
{
// Construction
public:

CDlgBarSampleView ();

// standard constructor on enlève la référence à l'id et la classe parent
```

- Il nous reste à Initialiser la **CDialogBar** dans la classe héritée de **CMDIChildWnd** (pour notre exemple la classe **CChildSampleView**) associée à la vue dans le cas d'un projet MDI ou dans la **CMainFrame** dans le cas d'un projet SDI.

```
// MDI
int CChildSampleView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1)
    return -1;
    // TODO: Add your specialized creation code here
    // dans le header de la classe child : CDlgBarSampleView m_wndDlgBar;
    if (!m_wndDlgBar.Create(this, IDD_DLGBAR_SAMPLEVIEW,
        CBRS_TOP|CBRS_TOOLTIPS|CBRS_FLYBY, IDD_DLGBAR_SAMPLEVIEW))
    {
      TRACE0("Failed to create DlgBar\n");
      return -1;      // Fail to create.
   }
   return 0;
}
```

Voir dans la documentation MSDN les différentes options pour le placement de la fenêtre, dans notre cas elle est placée en haut (**CBRS_TOP**).
Le résultat :



Pour résumer:
La barre d'outils comme la barre de dialogue s'initialise dans le message **WM_CREATE** du cadre fille MDI.
Nous obligeant à définir une classe héritée de **CMDIChildWnd** pour chaque vue à personnaliser.

II-G-1. Problèmes typiques avec la barre de dialogue

II-G-1-a. Les boutons ne sont pas actifs par défaut dans une barre de dialogue.

Effectivement par défaut un bouton dans une **CDialogBar** n'est pas actif.
Ce problème est référencé sur MSDN : INFO: CDialogBar Button Enabled When Command Handler Present
Pour se rendre compte du problème, rajouter un bouton dans la barre de dialogue.

Pour résoudre le problème deux solutions:

- On veut traiter le message dans la classe vue associée:
  Il faut déplacer (couper) à la main le message **ON_BN_CLICKED** et sa fonction de la **CDialogbar** pour les placer dans la gestion des messages de la classe fenêtre (on ne peut pas le faire directement par l'assistant, l'identifiant du bouton n'apparaissant pas dans la liste).

- On veut traiter le message dans la **CDialogBar** :
  On rajoutera un message de type
  ON_UPDATE_COMMAND_UI dans la **CDialogBar** pour activer le bouton (voir note)

```
BEGIN_MESSAGE_MAP(CMyDlgbar, CDialogBar)
//{{AFX_MSG_MAP(CMyDlgbar)
ON_BN_CLICKED(IDC_BUTTON2, OnButton2)
//}}AFX_MSG_MAP
ON_UPDATE_COMMAND_UI(IDC_BUTTON2, OnUpdateButton2)
```

```
END_MESSAGE_MAP()

// CMyDlgbar message handlers
void CMyDlgbar::OnUpdateButton2(CCmdUI* pCmdUI)
{
       // TODO: Add your command update UI handler code here
}
void CMyDlgbar::OnButton2()
{
     // TODO: Add your control notification handler code here
      AfxMessageBox("click bouton");
}
```

Note: La commande **ON_COMMAND_UI** est une ligne de code à rajouter manuellement dans le message map

II-G-1-b. Cacher et faire apparaître une barre de dialogue.

La classe **CDialogBar** est dérivée de la class **CControlBar** de même que la classe **CToolBar** Ce qui va suivre s'appliquera donc à ces deux classes.

Pour commencer il faut déclarer la barre de dialogue ancrable (docking) avec la commande

```
m_wndDlgBar.EnableDocking(CBRS_ALIGN_ANY);
```

Ensuite :
l'affichage d'une barre de dialogue se gère avec la fonction :

```
CFrameWnd::ShowControlBar
void ShowControlBar( CControlBar* pBar, BOOL bShow, BOOL bDelay );
```

Il suffit de disposer du pointeur sur le cadre qui gère la barre de dialogue:

La barre de dialogue est sur la MainFrame :

```
CMainFrame *pFrame=static_cast<CMainFrame *>(AfxGetMainWnd());
// cacher la toolbar.
pFrame->ShowControlBar(&pFrame->m_wndToolBar,FALSE,FALSE);
```

La barre de dialogue est sur le cadre fille MDI :

```
// à partir de la vue :
CChildSampleView *pFrame=static_cast<CChildSampleView*>(GetParentFrame());
pFrame->ShowControlBar(&pFrame->m_wndToolBar,FALSE,FALSE);
```

Une autre fonction intéressante pour gérer à travers un menu disposant d'une coche :
La fonctionnalité de cacher / faire apparaître la barre de dialogue.

```
// gérer le mode SHOW/Hide sur la commande d'un menu .
pFrame->ShowControlBar(&pFrame->m_wndToolBar,
!pFrame->m_wndToolBar->IsWindowVisible() ,FALSE);
```

Dernière fonction pour savoir si la barre est flottante :

```
CControlBar::IsFloating
BOOL IsFloating( ) const;

pFrame->m_wndToolBar->IsFloating();
```

Pour finir voici l'explication du mécanisme qui gère l'affichage ou le masquage de la barre d'outils générée par le wizard d'application :
La barre d'outils ou la barre de statut générée par l'assistant possède un identifiant prédéterminé et connu par le Framework.

Regardons par exemple le prototype de la fonction de création de la barre d'outils:

```
CToolBar::CreateEx
virtual BOOL CreateEx(
CWnd* pParentWnd,
DWORD dwCtrlStyle = TBSTYLE_FLAT,
DWORD dwStyle = WS_CHILD | WS_VISIBLE | CBRS_ALIGN_BOTTOM,
CRect rcBorders = CRect( 0, 0, 0, 0 ),
UINT nID = AFX_IDW_TOOLBAR );
```

L'identifiant nID est fourni par défaut avec la valeur **AFX_IDW_TOOLBAR** que l'on retrouve dans cet extrait de code MFC lié à la **CFrameWnd**:

```
void CFrameWnd::OnUpdateControlBarMenu(CCmdUI* pCmdUI)
{
    ASSERT(ID_VIEW_STATUS_BAR == AFX_IDW_STATUS_BAR);
    ASSERT(ID_VIEW_TOOLBAR == AFX_IDW_TOOLBAR);
    ASSERT(ID_VIEW_REBAR == AFX_IDW_REBAR);

    CControlBar* pBar = GetControlBar(pCmdUI->m_nID);
    if (pBar != NULL)
    {
        pCmdUI->SetCheck((pBar->GetStyle() & WS_VISIBLE) != 0);
        return;
    }
    pCmdUI->ContinueRouting();
}

BOOL CFrameWnd::OnBarCheck(UINT nID)
{
    ASSERT(ID_VIEW_STATUS_BAR == AFX_IDW_STATUS_BAR);
    ASSERT(ID_VIEW_TOOLBAR == AFX_IDW_TOOLBAR);
    ASSERT(ID_VIEW_REBAR == AFX_IDW_REBAR);

    CControlBar* pBar = GetControlBar(nID);
    if (pBar != NULL)
    {
        ShowControlBar(pBar, (pBar->GetStyle() & WS_VISIBLE) == 0,
FALSE);
        return TRUE;
    }
    return FALSE;
}
```

Voilà ce n'est pas compliqué,

Il est recommandé de s'inspirer de ce code pour cacher/faire apparaître ses propres barres d'outils ou barres de dialogues.

II-G-1-c. Initialiser les contrôles de la barre de dialogues.

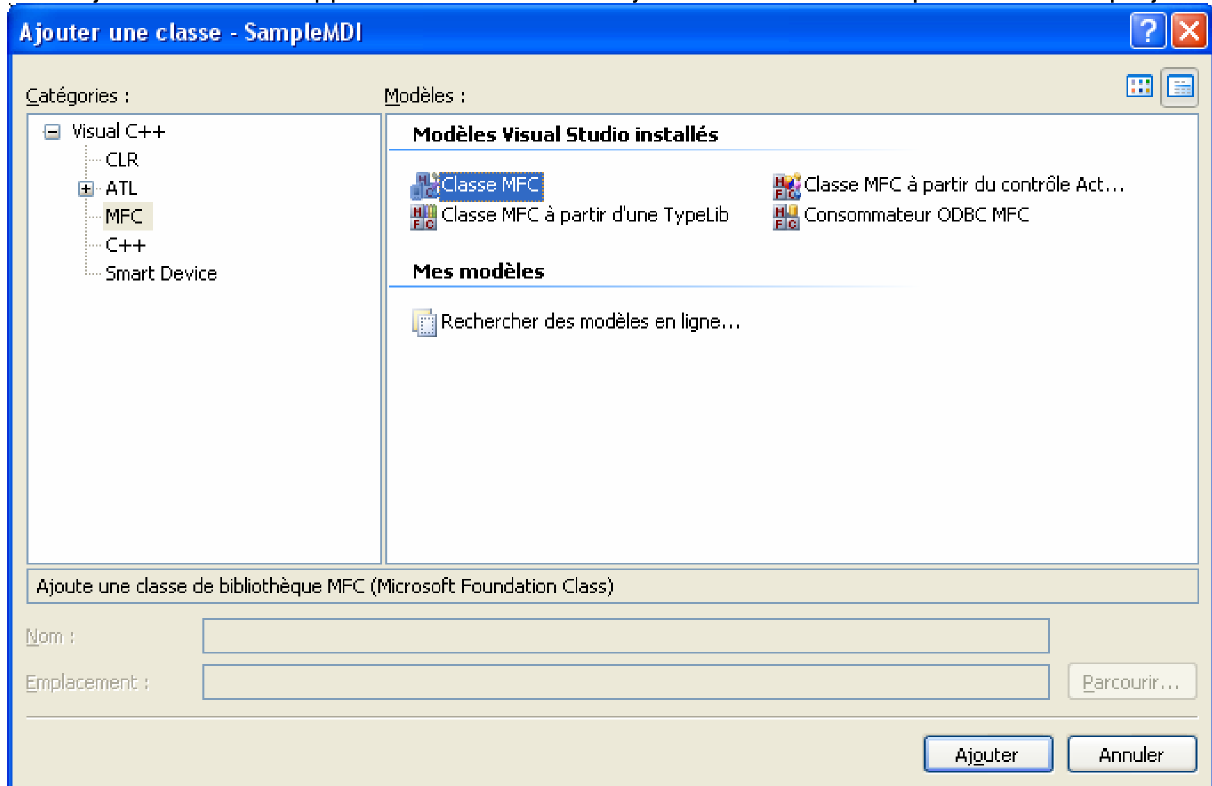
Une **CDialgBar** ne s'initialise pas malgré son apparence comme une **CDialog**, la fonction **OnInitDialog** n'est pas appelée (par le message **WM_INITDIALOG**).
L'endroit le plus sûr pour procéder à l'initialisation des contrôles c'est la vue, et plus exactement dans la fonction **OnInitialUpdate.**

```
void CSampleMDIView::OnInitialUpdate()
{
 CFormView::OnInitialUpdate();
 GetParentFrame()->RecalcLayout();
 ResizeParentToFit();

 CChildSampleView *pChild=static_cast<CChildSampleView*>(GetParentFrame());
 pChild ->m_DlgBar.OnInitDialog();
}
```

La barre de dialogue étant déclarée dans le cadre fille (child), il faut d'abord récupérer le pointeur sur le cadre MDI, puis appeler la fonction d'initialisation de la barre de dialogue.

II-H. Mise en place d'une barre de séparation dans un cadre MDI. (splitter).

La mise en place d'un splitter (ou rideau) est facile.
L'exemple qui suit montre comment intégrer deux fenêtres séparées par un splitter .
Etapes :
Il faut disposer d'une seconde fenêtre à générer avec l'aide de l'assistant au choix :**CFormView ,CScrollView ,CEditView** etc…
Pour l'exemple j'utiliserai la classe de base CEditView.
Pour ajouter la classe appelez la commande « ajouter une classe » à partir du menu projet.



Sélectionnez ensuite le modèle classe MFC.

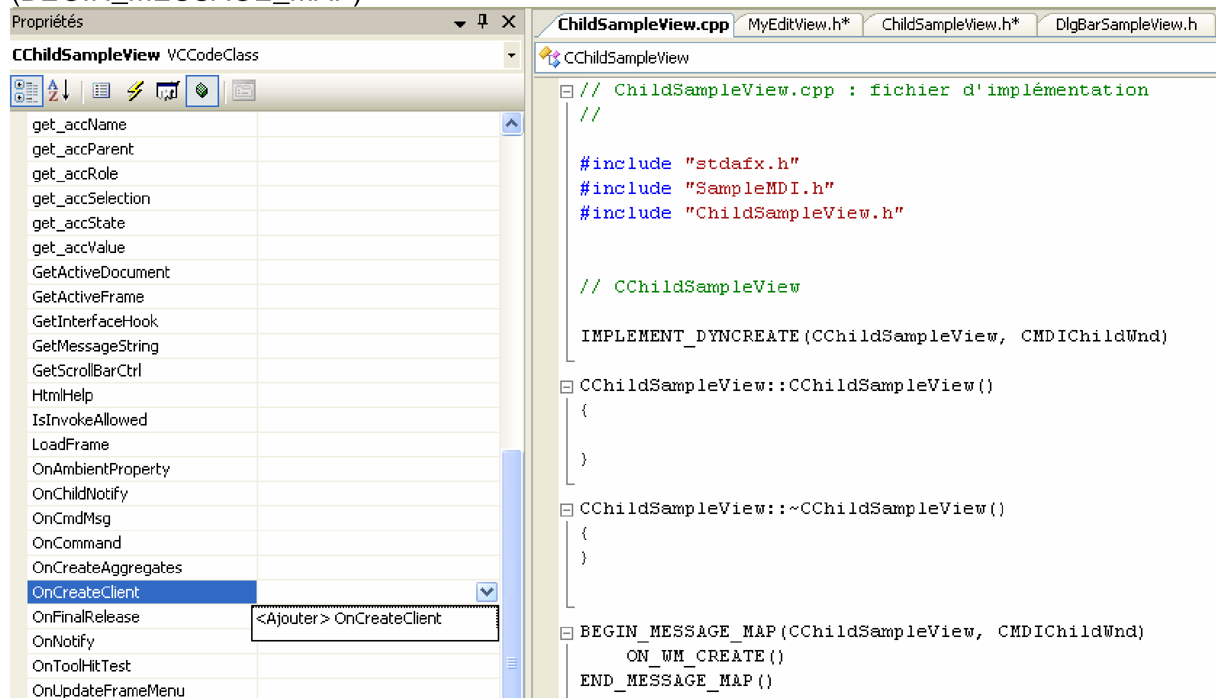
Après avoir sélectionné la classe de base **CEditView** indiquez le nom de votre classe.
La classe générée :

```
#pragma once
// Vue CMyEditView

class CMyEditView : public CEditView
{
	DECLARE_DYNCREATE(CMyEditView)

protected:
	CMyEditView();           // constructeur protégé utilisé par la
création dynamique
	virtual ~CMyEditView();

public:
#ifdef _DEBUG
	virtual void AssertValid() const;
#ifndef _WIN32_WCE
	virtual void Dump(CDumpContext& dc) const;
#endif
#endif

protected:
	DECLARE_MESSAGE_MAP()
};
```

il faudra ensuite modifier la classe cadre fille de la fenêtre principale pour rajouter notre initialisation :
Dans notre classe **CChildSampleView** on rajoutera la déclaration d'une donnée membre **m_wndSplitter** de la classe **CSplitterWnd** :

```
#pragma once

#include "ToolBarSampleView.h"
// Frame CChildSampleView

class CChildSampleView : public CMDIChildWnd
{
	DECLARE_DYNCREATE(CChildSampleView)
protected:
	CChildSampleView();           // constructeur protégé utilisé
par la création dynamique
	virtual ~CChildSampleView();
public:
CToolBarSampleView  m_ToolBar;
CSplitterWnd        m_wndSplitter;
protected:
	DECLARE_MESSAGE_MAP()
public:
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
};
```

Avec l'aide de l'assistant générez la fonction virtuelle **OnCreateClient** :
Pour cela placez-vous dans le source au niveau de la boucle des messages (BEGIN_MESSAGE_MAP)



Et appuyez sur la touche Alt+Entrée pour faire apparaître le volet des propriétés comme ci-dessus.

Initialisation du Splitter :

```
BOOL CChildSampleView::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
// TODO : ajoutez ici votre code spécialisé et/ou l'appel de la classe de base
      if (!m_wndSplitter.CreateStatic(this, 1, 2))return FALSE;

      if (!m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(CSampleMDIView), CSize(100, 100), pContext) ||
            !m_wndSplitter.CreateView(0, 1, RUNTIME_CLASS(CMyEditView), CSize(100, 100), pContext))
      {
            m_wndSplitter.DestroyWindow();
      return FALSE;
      }
      return TRUE;
      //return CMDIChildWnd::OnCreateClient(lpcs, pContext);
}
```

La première ligne **CreateStatic** initialise le splitter avec une ligne et deux colonnes ce qui correspond ici à deux fenêtres séparées verticalement par le splitter.
Les lignes suivantes correspondent aux classes fenêtres associées au splitter.
Dans la fonction **CreateView** on spécifie la ligne et colonne de la fenêtre, sa surface d'affichage initial, et bien sûr la signature (runtime) de la classe fenêtre.
C'est tout.

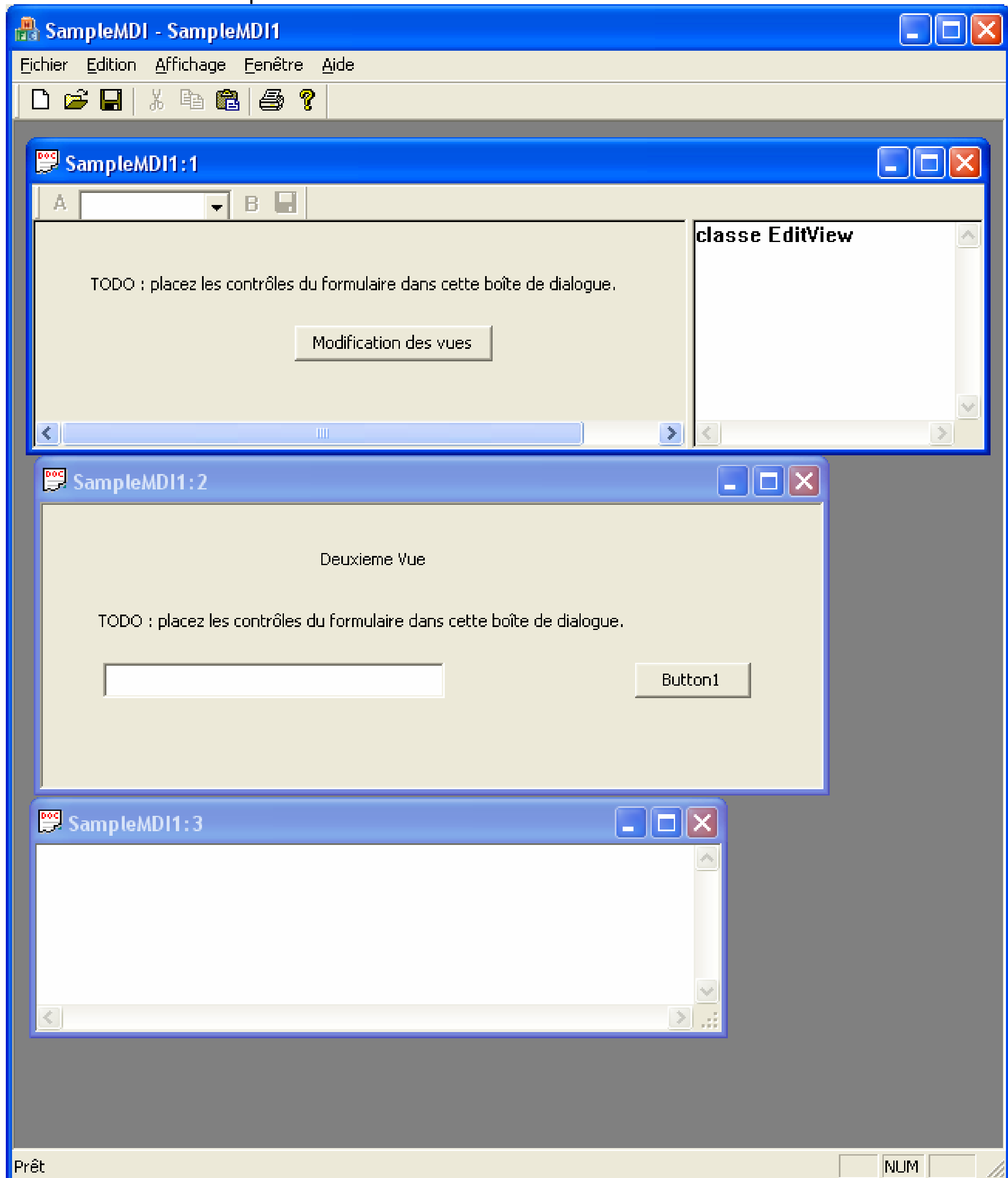
Note : Une des deux classes doit être celle qui est déclarée dans la fonction **InitInstance** de la classe d'application :

```
// Extrait InitInstance de CSampleMDIApp dérivée de CWinApp.

      CMultiDocTemplateEx *pDocTemplate = new
CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc)  ,
            RUNTIME_CLASS(CChildSampleView)     , // frame enfant
MDI personnalisé
        RUNTIME_CLASS(CSampleMDIView)  ,IDD_SAMPLEMDI_FORM);
```

Il faudra aussi veiller à fournir les includes des fenêtres concernées dans le source de notre classe **CChildSampleView** .

```
// SampleMDIView.cpp : implémentation de la classe CSampleMDIView
//
#include "stdafx.h"
#include "SampleMDI.h"

#include "SampleMDIDoc.h"
#include "FormViewBis.h"
#include "SampleMDIView.h"

#include "ChildSampleView.h"
#include "MainFrm.h"
```

Le résultat de notre implémentation:



Les sources du projet : http://farscape.developpez.com/Samples/SampleMDI-1.zip

III. Simplifier l’ajout d’éléments dans le cadre de travail (Framework).

Nous l’avons vu dans les pages précédentes, l’ajout des éléments barres d’outils, dialogues et splitter est pénible avec les MFC.
Pour chaque ajout il faut définir une classe cadre fille spécifique avec le code d’initialisation du composant.
Il serait beaucoup plus agréable de pouvoir spécifier ces ajouts directement dans **InitInstance** sans devoir générer une classe spécifique à chaque fois.

C’est ce je vous propose de réaliser maintenant.

Avertissement : ce chapitre est d’un niveau un peu plus avancé, néanmoins les portions de code y figurant ont été étudiées tout au long des chapitres précédents.

III-A. Spécifier les éléments du cadre fille directement dans InitInstance

La classe **CDocumentTemplate** permet déjà de définir le contexte de travail de notre cadre fille MDI.
Il serait donc naturel d’envisager de faire de même pour les autres éléments composants le cadre fille à savoir barre d’outils et la barre de Dialogue.

Il suffira donc d’enrichir notre classe **CMultiDocTemplateEx** pour stocker les éléments permettant la création d’une barre d’outils ou de dialogue.

Phase suivante il faudra être capable d’accéder à ces informations dans le **OnCreate** de la MDIChild.
Malheureusement on ne dispose pas de l’accès à la classe **CMultiDocTemplateEx** dans cet objet.
Si on reprend les élements que nous avons étudiés plus haut nous savons que le traitement s’effectue dans la fonction **OpenDocumentFile** de la classe **CDocTemplate**.

En détaillant le code on trouve l’appel de la fonction **CreateNewFrame** qui permet de créer le cadre fille MDI.

```
virtual CFrameWnd* CreateNewFrame(CDocument* pDoc, CFrameWnd* pOther);
```

la solution devient donc simple:

il nous faut définir une classe héritée de **CMIChildWnd** qui pendra en charge la création des éléments de l’interface.
Cette classe aura accés au **CDocTemplate** par l’intemédiaire d’un pointeur initialisé dans la fonction **CreateNewFrame**.

III-A-1. Détails du code

Détail du code pour la classe **CMultiDocTemplateEx** :
J'ai rajouté trois nouvelles fonctions qui apparaissent en gras.

```
#pragma once
#include "ToolBarEx.h"
// CMultiDocTemplateEx extension de CMultiDocTemplate
class CMultiDocTemplateEx :public CMultiDocTemplate
{
public:
	CMultiDocTemplateEx( UINT nIDResource,
					CRuntimeClass* pDocClass,
					CRuntimeClass* pFrameClass,
					CRuntimeClass* pViewClass,
					UINT nID=0
					):
	CMultiDocTemplate(nIDResource,pDocClass,
				pFrameClass,
				pViewClass),m_pDocClass(pDocClass),

m_pFrameClass(pFrameClass),

m_pViewClass(pViewClass),m_nID(nID)
	{
		m_arDocTemplateEx.Add(this);
	}
	static CMultiDocTemplateEx *GetTemplateByRuntimeClass(CRuntimeClass
*pClass);
	static CMultiDocTemplateEx *GetTemplateByIdClass(UINT nID);

	CRuntimeClass* GetDocClass(){return m_pDocClass;}
	CRuntimeClass* GetFrameClass(){return m_pFrameClass;}
	CRuntimeClass* GetViewClass(){return m_pViewClass;}
	UINT GetnIDResource()		{return m_nID;}

	static INT_PTR GetCount(){return m_arDocTemplateEx.GetCount();}
	static CMultiDocTemplateEx* GetAt(int i)
	{
		VERIFY(i>=0 && i<GetCount());
		return m_arDocTemplateEx.GetAt(i);
	}
	bool CreateNewView(CMultiDocTemplate *pTemplateFrom,CView *pViewDest)
	{
		CFrameWnd * pFrame =pTemplateFrom->CreateNewFrame(pViewDest-
>GetDocument(),pViewDest->GetParentFrame());
		if(!pFrame) return false;
	    InitialUpdateFrame(pFrame,pViewDest->GetDocument());
	    return true;
	}

	virtual CFrameWnd* CreateNewFrame(CDocument* pDoc, CFrameWnd*
pOther);

	void AddToolBar(CRuntimeClass* pToolBar,UINT nid,DWORD dwStyle,CSize
SizeButton);
	void AddDialogBar(CRuntimeClass* pDlgBar,UINT nid,DWORD dwStyle);

public:
	struct INFOSBar
```

```
	{
		CRuntimeClass*	pRuntime;
		UINT		nid;
		DWORD		dwStyle;
		CSize		size;
	};

	static CArray<CMultiDocTemplateEx *,CMultiDocTemplateEx *>
m_arDocTemplateEx;
	CRuntimeClass* m_pDocClass;
	CRuntimeClass* m_pFrameClass;
	CRuntimeClass* m_pViewClass;
	unsigned int   m_nID;

	CArray<struct INFOSBar,struct INFOSBar>		m_ArrayDialogBar;
	CArray<struct INFOSBar,struct INFOSBar>		m_ArrayToolBar;
};
```

Détail du code pour les fonctions rajoutées :

Le passage important se résume au moment ou je mémorise l'adresse du Document Template dans le cadre MDI fille (lignes en gras).

```
void CMultiDocTemplateEx::AddDialogBar(CRuntimeClass* pDlgBar,UINT
nid,DWORD dwStyle)
{
	ASSERT(pDlgBar->IsDerivedFrom(RUNTIME_CLASS(CDialogBar)));
	struct INFOSBar bar;
	bar.pRuntime=pDlgBar;
	bar.nid=nid;
	bar.dwStyle=dwStyle;
	m_ArrayDialogBar.Add(bar);
}
//////////////////////////////////////////////////////////////////////////
void CMultiDocTemplateEx::AddToolBar(CRuntimeClass* pToolBar,UINT nid,DWORD
dwStyle,CSize SizeButton)
{
	ASSERT(pToolBar->IsDerivedFrom(RUNTIME_CLASS(CToolBarEx)));
	struct INFOSBar bar;
	bar.pRuntime=pToolBar;
	bar.nid=nid;
	bar.dwStyle=dwStyle;
	bar.size=SizeButton;
	m_ArrayToolBar.Add(bar);
}
//--------------------------------------------------------------------------
CFrameWnd* CMultiDocTemplateEx::CreateNewFrame(CDocument* pDoc, CFrameWnd*
pOther)
{
	if (pDoc != NULL)
		ASSERT_VALID(pDoc);
	// create a frame wired to the specified document

	ASSERT(m_nIDResource != 0); // must have a resource ID to load from
	CCreateContext context;
	context.m_pCurrentFrame = pOther;
	context.m_pCurrentDoc = pDoc;
	context.m_pNewViewClass = m_pViewClass;
	context.m_pNewDocTemplate = this;
```

```
	if (m_pFrameClass == NULL)
	{
	TRACE(traceAppMsg, 0, "Error: you must override
CDocTemplate::CreateNewFrame.\n");
		ASSERT(FALSE);
		return NULL;
	}
	CFrameWnd* pFrame = (CFrameWnd*)m_pFrameClass->CreateObject();
	if (pFrame == NULL)
	{
	TRACE(traceAppMsg, 0, "Warning: Dynamic create of frame %hs
failed.\n",
		m_pFrameClass->m_lpszClassName);
		return NULL;
	}
	ASSERT_KINDOF(CFrameWnd, pFrame);

	if(pFrame->GetRuntimeClass()-
>IsDerivedFrom(RUNTIME_CLASS(CMIChildWndEx)))
	static_cast<CMIChildWndEx *>(pFrame)->SetDocTemplate(this);

	if (context.m_pNewViewClass == NULL)
	TRACE(traceAppMsg, 0, "Warning: creating frame with no default
view.\n");

	// create new from resource
	if (!pFrame->LoadFrame(m_nIDResource,
		WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,   // default frame styles
			NULL, &context))
	{
		TRACE(traceAppMsg, 0, "Warning: CDocTemplate couldn't create a
frame.\n");
		// frame will be deleted in PostNcDestroy cleanup
		return NULL;
	}

	// it worked !
	return pFrame;
}
```

Détails du code de la classe cadre fille **CMIChildWndEx :**
J'ai rajouté un ensemble de fonctions pour la gestion des barres d'outils,barre de dialogue et du splitter.

```
// Frame CMIChildWndEx

class CMIChildWndEx : public CMDIChildWnd
{
	DECLARE_DYNCREATE(CMIChildWndEx)
public:

	void SetDocTemplate(CMultiDocTemplateEx
*pTemplate){m_pDocTemplate=pTemplate;}
	INT_PTR GetToolBarCount(){return m_ToolbBarArray.GetSize();}
	INT_PTR GetDialogBarCount(){return m_DlgBarArray.GetSize();}
	CToolBarEx *GetToolBarAt(int nIndex);
	CDialogBar *GetDlgBar(int nIndex);

	void  SetSplitter(CSplitterWnd *pSlitter){m_pSlitter=pSlitter;}
	CSplitterWnd *GetSplitter(){return m_pSlitter;}

protected:
	CMIChildWndEx();           // constructeur protégé utilisé par la
création dynamique
	virtual ~CMIChildWndEx();

protected:
	CMultiDocTemplateEx                      *m_pDocTemplate;

	CArray<CToolBarEx*,CToolBarEx *> m_ToolbBarArray;
	CArray<CDialogBar *,CDialogBar *> m_DlgBarArray;
	CSplitterWnd                              *m_pSlitter;
	DECLARE_MESSAGE_MAP()
	virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
pContext);
public:
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
};
```

Le détail du code de la classe:

```
BOOL CMIChildWndEx::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
pContext)
{
	// TODO : ajoutez ici votre code spécialisé et/ou l'appel de la
classe de base
	CWinAppEx *pApp=static_cast<CWinAppEx*>(AfxGetApp());
	if(pApp->GetRuntimeClass()->IsDerivedFrom(RUNTIME_CLASS(CWinAppEx)))
	{
		if(pApp->OnChildCreateClient(this,lpcs,pContext)) return TRUE;
	}
	return CMDIChildWnd::OnCreateClient(lpcs, pContext);
}
//-----------------------------------------------------------------
int CMIChildWndEx::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
{
	if (CMDIChildWnd::OnCreate(lpCreateStruct) == -1)
		return -1;

	if(!m_pDocTemplate) return 0;

	// TODO:  Ajoutez ici votre code de création spécialisé
	struct CMultiDocTemplateEx::INFOSBar bar;
	CToolBarEx *pToolBar;
	for(int i=0;i<m_pDocTemplate->m_ArrayToolBar.GetSize();i++)
	{
	bar=m_pDocTemplate->m_ArrayToolBar.GetAt(i);
	pToolBar=static_cast<CToolBarEx *>(bar.pRuntime->CreateObject());
		if(!pToolBar)
		{
			TRACE("Failed to create toolbar:%d\n",bar.nid);
			return -1;	// fail to create
		}
		if(!bar.dwStyle)
		bar.dwStyle=WS_CHILD | WS_VISIBLE | CBRS_ALIGN_TOP|
CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC;
		if (!pToolBar->CreateEx(this, TBSTYLE_FLAT,bar.dwStyle) ||
			!pToolBar->LoadToolBar(bar.nid))
		{
			TRACE("Failed to create toolbar:%d\n",bar.nid);
			return -1;	// fail to create
		}
		pToolBar->OnInitToolBar();
		pToolBar->InitSizes(bar.size);
		pToolBar->SetWindowText(_T("Toolbar"));
		pToolBar->UpdateSizes();
		pToolBar->SetBarStyle(pToolBar->GetBarStyle() | CBRS_TOOLTIPS |
CBRS_FLYBY | CBRS_SIZE_DYNAMIC);

		// TODO: Delete these three lines if you don't want the toolbar
to
		//  be dockable
		EnableDocking(CBRS_ALIGN_ANY);
		pToolBar->EnableDocking(CBRS_ALIGN_ANY);
		DockControlBar(pToolBar);
		m_ToolbBarArray.Add(pToolBar);
	}
	CDialogBar *pDlgBar;
	for(int i=0;i<m_pDocTemplate->m_ArrayDialogBar.GetSize();i++)
	{
		bar=m_pDocTemplate->m_ArrayDialogBar.GetAt(i);
		pDlgBar =static_cast<CDialogBar *>(bar.pRuntime-
>CreateObject());
		if(!bar.dwStyle) bar.dwStyle=CBRS_TOP;
		if(!pDlgBar->Create(this,bar.nid,bar.dwStyle,bar.nid))
		{
			TRACE("Failed to create DlgBar:%d\n",bar.nid);
			return -1;
		}
		m_DlgBarArray.Add(pDlgBar);
	}
    return 0;
}
//---------------------------------------
CToolBarEx *CMIChildWndEx::GetToolBarAt(int nIndex)
{
```

```
        ASSERT(nIndex>=0 && nIndex<m_ToolbBarArray.GetSize());
        return m_ToolbBarArray.GetAt(nIndex);
}
//--------------------------------------
CDialogBar *CMIChildWndEx::GetDlgBar(int nIndex)
{
        ASSERT(nIndex>=0 && nIndex<m_DlgBarArray.GetSize());
        return m_DlgBarArray.GetAt(nIndex);
}
```

La fonction **OnCreateClient** redirige l'appel vers une fonction virtuelle de ma classe d'application **OnChildCreateClient** qui me permettra de traiter la mise en place du splitter .
La fonction **OnCreate** s'occupe de mettre en place les éventuelles barres de dialogues ou barres d'outils.

Enfin la déclaration des éléments d'interface au niveau de la classe d'application, directement au niveau de la classe template :

```
CMultiDocTemplateEx *pDocTemplate = new
CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc)  ,
            RUNTIME_CLASS(CMIChildWndEx)  , // frame enfant MDI personnalisé
            RUNTIME_CLASS(CSampleMDIView) ,IDD_SAMPLEMDI_FORM);
      pDocTemplate-
>AddToolBar(RUNTIME_CLASS(CToolBarSampleView),IDR_TOOLBARONE,0,CSize(16,15)
);
      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);

      pDocTemplate = new CMultiDocTemplateEx(IDR_SampleMDITYPE,
            RUNTIME_CLASS(CSampleMDIDoc),
            RUNTIME_CLASS(CMIChildWndEx), // frame enfant MDI personnalisé
            RUNTIME_CLASS(CFormViewBis),IDD_SAMPLEMDI_FORMBIS);
      pDocTemplate-
>AddDialogBar(RUNTIME_CLASS(CDlgBarSampleView),IDD_DLGBAR_SAMPLEVIEW,CBRS_T
OP);

      if (!pDocTemplate)return FALSE;
      AddDocTemplate(pDocTemplate);
```

L'exemple ci-dessus déclare une barre d'outils et une barre de dialogue, directement à la déclaration du document template.
Pour mon exemple des classes dérivées de **CDialogBar** ou **CToolBarEx** sont spécifiées et seront donc créées dynamiquement d'après la signature de leur classe …

Il reste à voir la création du splitter déporté dans la classe d'application :

```
//------------------------------------------------------
bool CSampleMDIApp::OnChildCreateClient(CMIChildWndEx
*pChild,LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
	CMultiDocTemplateEx *pDocTpl=static_cast< CMultiDocTemplateEx
*>(pContext->m_pNewDocTemplate);
	if(pDocTpl->GetnIDResource()==IDD_SAMPLEMDI_FORM)
	{
		CSplitterWnd *pSlitter= new CSplitterWnd;
		if (!pSlitter->CreateStatic(pChild, 1, 2))return FALSE;

		if (!pSlitter->CreateView(0, 0, RUNTIME_CLASS(CSampleMDIView),
CSize(100, 100), pContext) ||
			!pSlitter->CreateView(0, 1, RUNTIME_CLASS(CMyEditView),
CSize(100, 100), pContext))
		{
			pSlitter->DestroyWindow();
			return FALSE;
		}
		pChild->SetSplitter(pSlitter);
		return TRUE;
	}
	return FALSE;
}
```

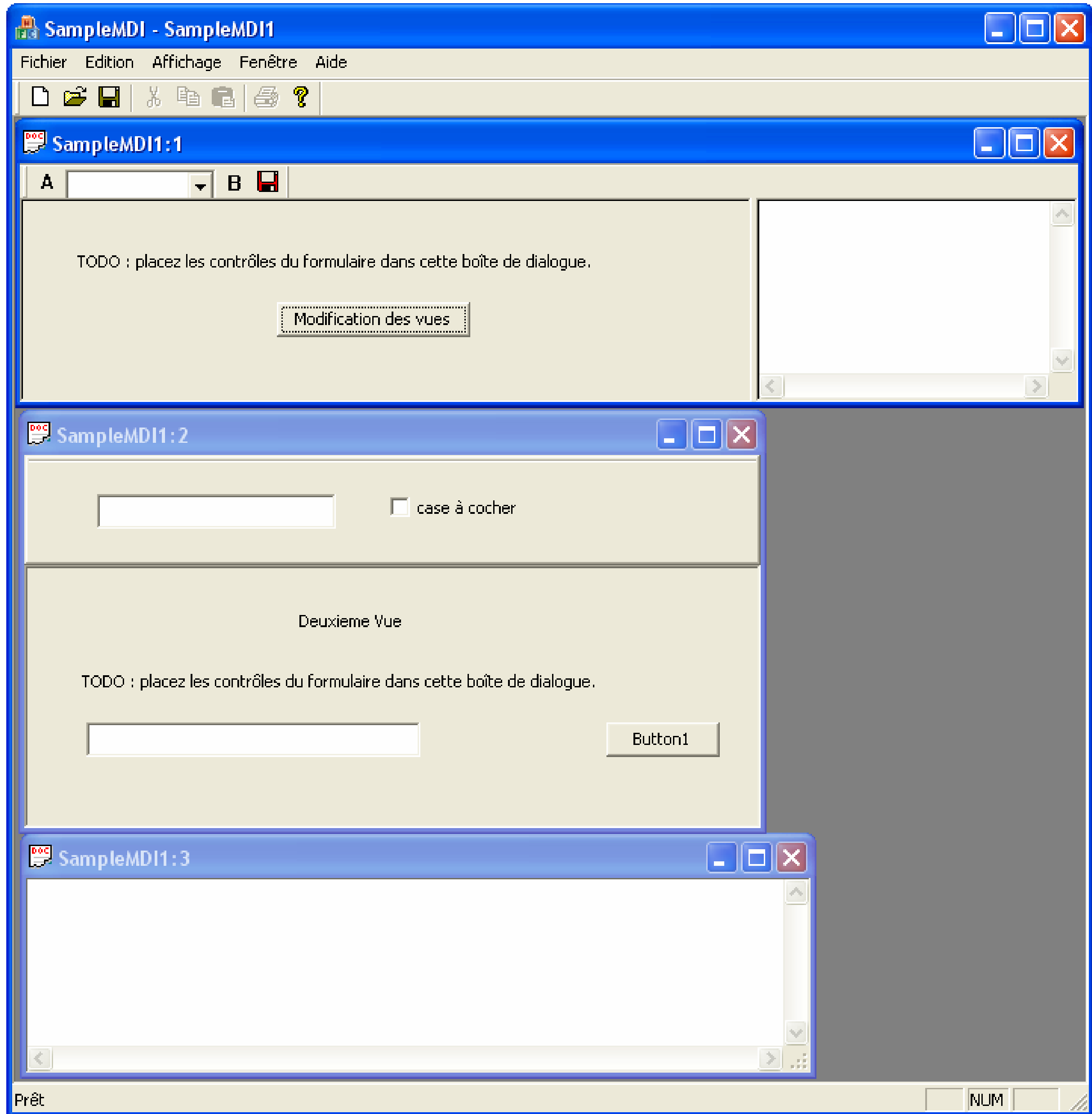
Le traitement est simple, notre classe Document template nous permet de savoir le type de formulaire créé pour faire notre traitement.

III-A-2. En Résumé :

Avec la redéfinition des trois classes principales de l'architecture MDI:
La classe **CMultiDocTemplate**
La classe D'application **CWinApp**
Et la classe **CMIChildWnd**
Je pense vous avoir démontré une simplification significative pour l'écriture et la mise en place d'une architecture MDI et de ses principaux composants graphiques.

Le projet et l'ensemble des sources le composant est disponible à l'emplacement suivant :
http://farscape.developpez.com/Samples/SampleMDI-2.zip

III-A-3. Résultat final :



Remerciements

Je remercie toute l'équipe du forum **Visual C++** pour sa relecture attentive du document.

Les sources présentées sur cette page sont libres de droits, et vous pouvez les utiliser à votre convenance.
Par contre, la page de présentation constitue une œuvre intellectuelle
Protégée par les droits d'auteurs. **Copyright © 2006 farscape**.
Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur.
Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 € de dommages et intérêts.
Cette page est déposée à la SACD.