

Introduction à Borland Together sous Borland Turbo Delphi

par [Pierre RODRIGUEZ \(Pedro\)](#)

Date de publication : 2 octobre 2006

Dernière mise à jour :

La réorientation de Borland au niveau du développement de ses produits nous a appris qu'ils comptaient se concentrer dorénavant sur les logiciels gérant le cycle de vie des applications (ALM). C'est pourquoi l'accent est mis particulièrement sur ces programmes. Comment faciliter la programmation en modélisant? C'est ce que l'on va pouvoir étudier grâce à Turbo Delphi édition Explorer et Together.

- I - Introduction
- II - Introduction à la modélisation
- III - Présentation de Together
- IV - Quelques fonctionnalités de Together
 - IV.1 - Ajouter le support de la modélisation avec Together
 - IV.2 - De la modélisation au code: LiveSource
 - IV.3 - Créer une classe avec Together
 - IV.3.a - Les propriétés
 - IV.3.b - Les méthodes
 - IV.4 - Créer une classe dérivée d'une classe ancêtre
- V - Conclusion

I - Introduction

Le 5 septembre 2006, le groupe de développeurs d'outils de Borland (devco) met en téléchargement sur leur site [les éditions Turbo](#). Cette gamme est mono langage et se divise en 4 versions différentes: Turbo Delphi, Turbo Delphi for .NET, Turbo C++, Turbo C#. Ces versions sont elles-mêmes scindées en 2 éditions: l'édition Explorer et l'édition Professionnelle.

L'édition qui nous intéresse est Turbo Delphi Explorer puisqu'elle est gratuite et Together y est fourni.

II - Introduction à la modélisation

La modélisation est une notion qui, de nos jours, prend une place très importante dans le processus de développement.

L'idée est simple: dessiner un problème dans une syntaxe précise, uniformisée et donc, compréhensible par tout le monde. En d'autres termes, la modélisation est une "mise à plat", une représentation abstraite.

Il est possible de modéliser toutes sortes de problèmes.

En programmation, la modélisation permet de structurer son programme (classes, interface, etc.) à l'aide de graphiques.

Que ce soit sous forme graphique ([UML](#)) ou bien en vue modèle, cela permet de concevoir et de structurer le programme de façon rigoureuse.

Le gros avantage est de faciliter la programmation objet: on peut visualiser et composer très facilement les différentes interactions entre les classes par exemple; ce qui est bien plus difficile à réaliser en code pur.

Un autre avantage, celui d'aider au *back engineering*: le fait de partir d'une source complète pour comprendre le fonctionnement du programme. On peut aisément comprendre le casse-tête que cela peut être de comprendre le fonctionnement d'un programme écrit par quelqu'un d'autre seulement avec le code...

La visualisation graphique ne fait évidemment pas tout mais donne une vue d'ensemble très pratique dans cet exercice.

L'intérêt est d'avoir également, avec un rapide coup d'œil, une visualisation de la structure du programme.

Il existe beaucoup d'autres avantages à utiliser la modélisation.

Pour plus de renseignements, veuillez consulter les sites suivants de la section Conception de [developpez.com](#):

- [Section conception de developpez.com](#)
- [Section UML de developpez.com](#)
- [La FAQ UML de developpez.com](#)
- [Tutoriel : Introduction à UML par Laurent Piechocki](#)
- [Tutoriel : Introduction à UML et à Together sous BDS 2006 par Merlin](#)
- [Article : Pourquoi modéliser par GOLLOT Erik](#)

On peut également noter qu'il existe des pages qui présentent Together dans l'aide intégrée de Turbo:

ms-help://borland.bds4/bds4guide/html/tgNET_Part.htm

Il existe aussi des livres traitant du sujet qui pourront vous être grandement utile également:

- [Introduction à UML \(Sinan Si Alhir, traduit par Alexandre Gachet\)](#)
- [L'orienté objet \(H. Bersini et I. Wellesz\)](#)

III - Présentation de Together

Together est un outil de modélisation intégré à Turbo. Il contient la plupart des fonctionnalités que l'on peut espérer d'un tel outil.

- Synchroniser le modèle et le code source du projet,
- Générer la documentation de l'application,
- Prendre en compte la qualité dès le modèle grâce aux audits et métriques,

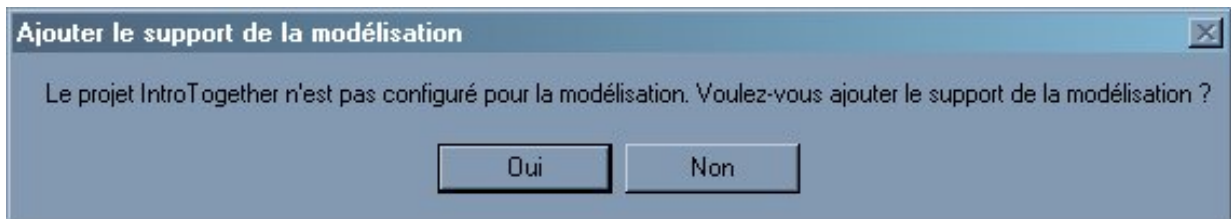
Together permet ainsi d'avoir 3 visualisations différentes de chaque projet:

- La hiérarchie du modèle présentée dans la vue Modèle.
- La représentation graphique du modèle dans la vue Diagramme.
- Le code source (pour les projets d'implémentation)

IV - Quelques fonctionnalités de Together

IV.1 - Ajouter le support de la modélisation avec Together

En cliquant sur l'onglet **Vue modèle** du **Gestionnaire de projet**, vous accédez à la vue modélisée de votre projet en cours. Si votre projet n'est pas relié à Together, un message vous propose de le faire:



Message de liaison à Together

Vous pouvez également activer ou désactiver à tout moment le support de Together en cliquant sur **Projet\Support Together...** Une boîte de dialogue apparaît alors:



Boîte de liaison à Together

Cochez le(s) projet(s) qui doivent être supportés par Together et validez.

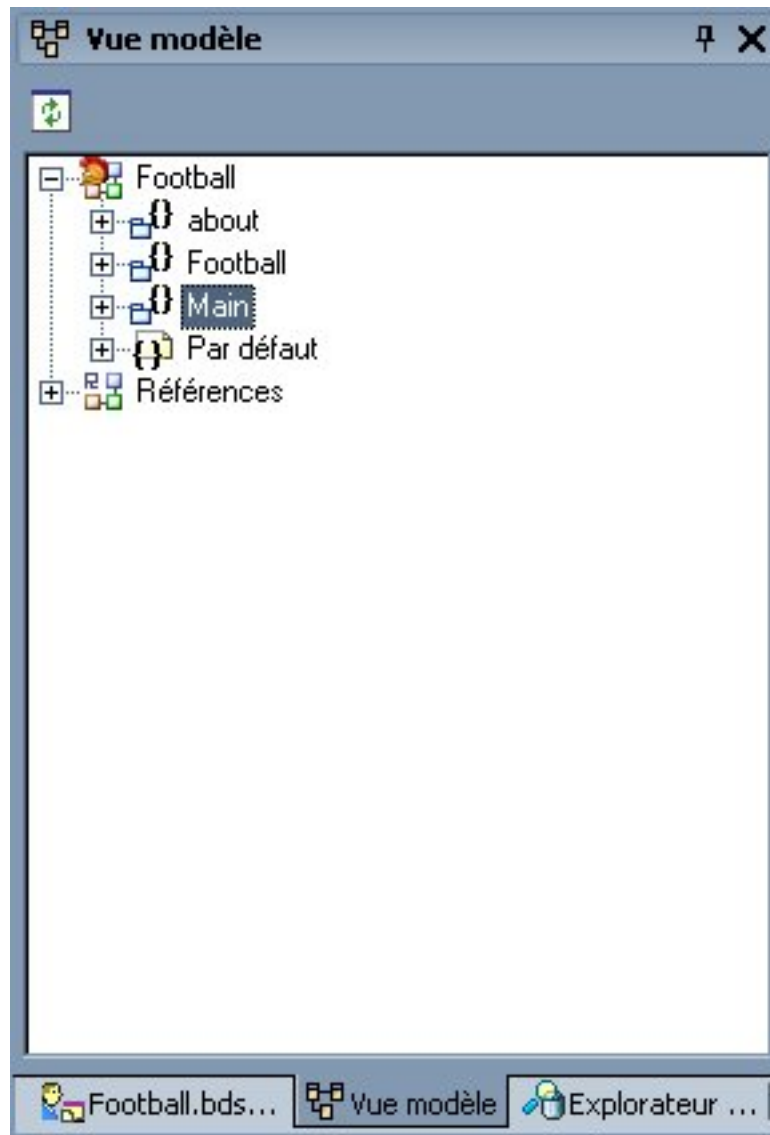
IV.2 - De la modélisation au code: LiveSource

LiveSource est la fonctionnalité la plus importante de Together. Elle permet la modélisation d'un projet soit par le code, soit avec une vue hiérarchique du modèle (Vue modèle), soit avec une vue graphique en diagramme (UML).

Pour mieux comprendre, ouvrons un projet de démonstration fourni avec Turbo Delphi.

Choisissons le projet Football.bdsproj. Il se trouve dans [Répertoire d'installation de Turbo]\Demos\DelphiWin32\VCLWin32\Football.

Une fois le projet ouvert, connectons-le avec Together. Nous avons donc l'onglet *Vue modèle* qui donne ceci:



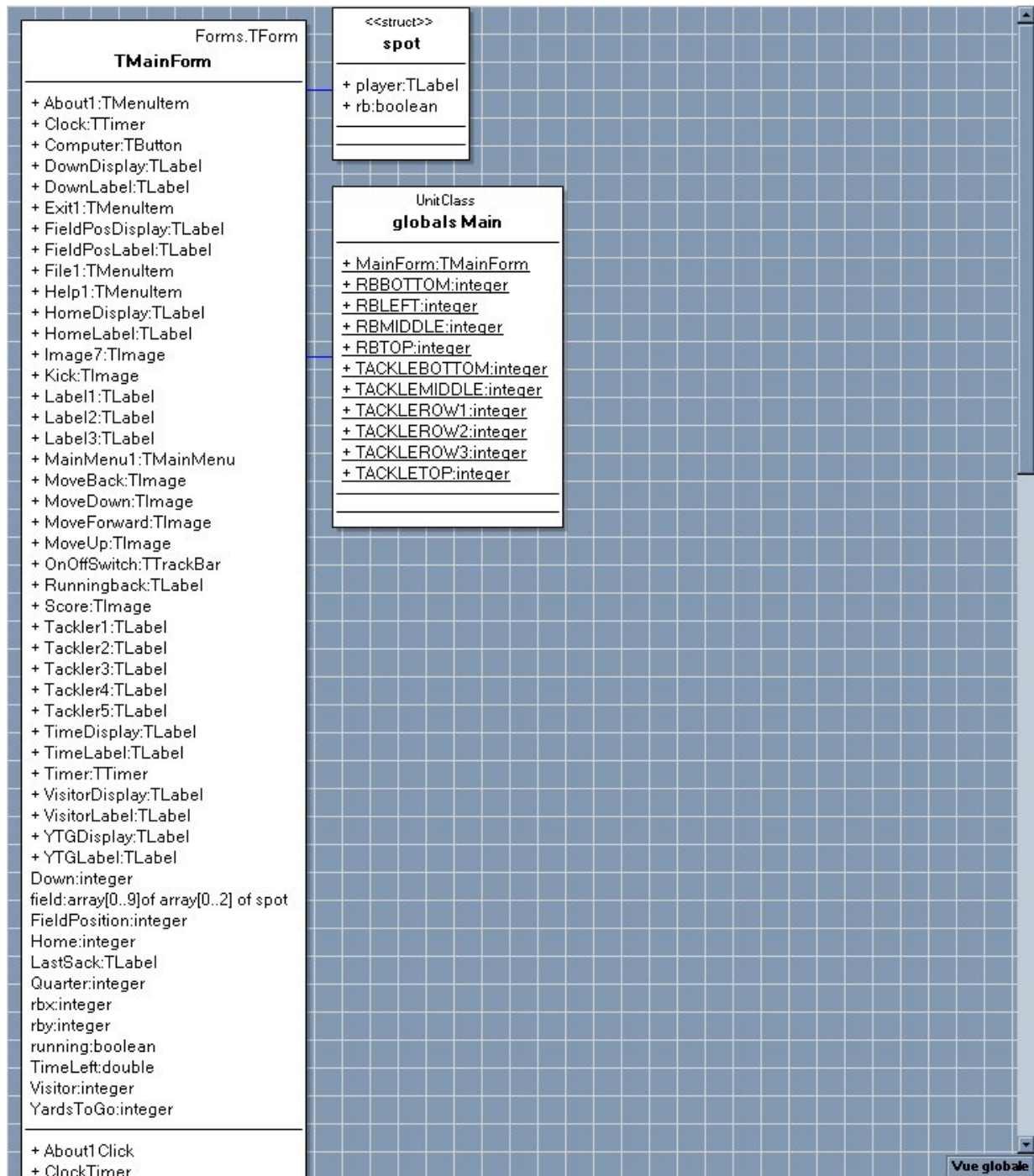
Le projet Football dans la vue modèle

On peut remarquer que le noeud **Football** contient 4 sous-noeuds:

- about: Représente la fiche about
- Football: représente le projet Football
- Main: représente la fiche principale main
- Par défaut

Cet onglet est une des 3 visualisations possibles fournies par Together. On peut y ajouter des membres/classes/méthodes, les déplacer, les couper, les copier, les coller, bref, on peut les manipuler de toute sortes de façons.

En double-cliquant sur Main dans la vue modèle, par exemple, on peut afficher le diagramme correspondant à la fiche Main dans une vue UML:



Vue UML de la fiche

Chaque élément de cette vue UML a les mêmes comportements que les noeuds de la vue modèle: déplacement, copie, coupe, collage, renommage, etc sont possibles.

On peut naviguer facilement entre les 3 visualisations en utilisant le menu contextuel:

- Atteindre la définition: Affiche la définition dans le code de l'objet sélectionné
- Synchroniser la vue modèle: Met à jour la vue modèle et sélectionne l'objet
- Sélectionner dans le diagramme: Sélectionne l'objet dans la vue UML

IV.3 - Créer une classe avec Together

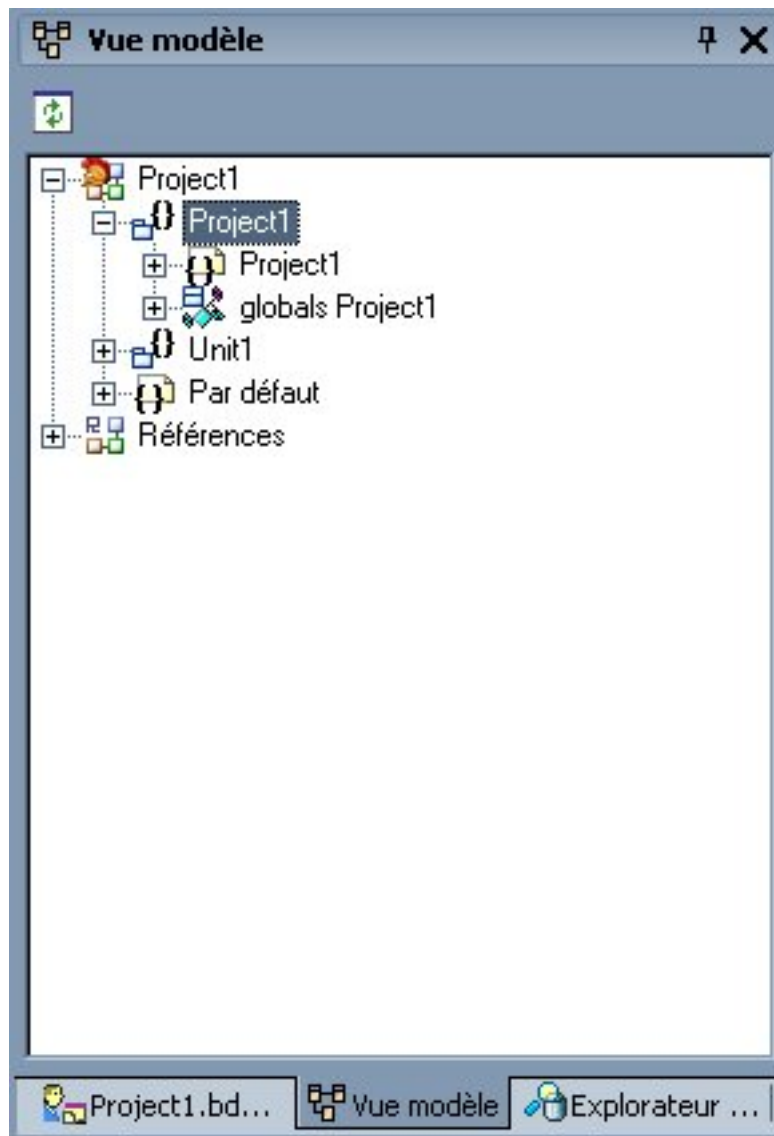
Rentrons dans le vif du sujet et créons une **classe** avec Together.

La classe que nous allons créer chargera un fichier texte puis l'insérera dans un TMemo.

Voici comment nous allons structurer cette classe:

- Une propriété FileName qui chargera le fichier dès qu'elle sera renseignée
- Une **méthode** AppendToMemo qui ajoutera le contenu du fichier dans le TMemo
- Une propriété Count qui renvoie le nombre de lignes chargées.

Créons un nouveau projet Win32 (**FichierNouveau...Application Fiches VCL - Delphi pour Win32**) et connectons-le à Together.



Projet vierge dans Together

Ajoutons maintenant une unité vierge dans le projet en cliquant sur **FichierNouveau..Unité - Delphi pour Win32**.

Il est nécessaire de mettre à jour la vue modèle pour continuer. Pour cela, on clique sur le petit icône



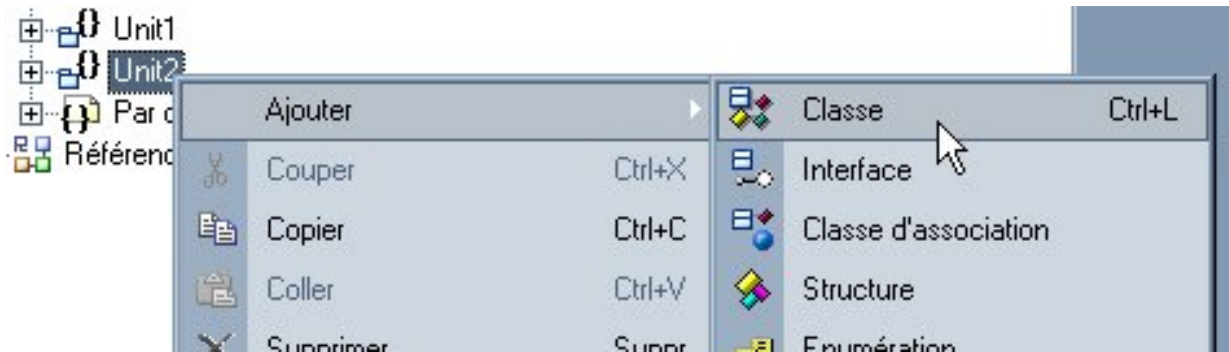
Actualiser la vue modèle

au dessus de la vue modèle. Un nouveau noeud portant le nom de l'unité apparaît.

Comme vous avez pu le voir, le support ou la mise à jour de Together avec votre projet nécessite l'enregistrement de chaque fichier du projet.

IV.3.a - Les propriétés

Créons donc notre classe: Cliquez droit sur le nouveau noeud de l'unité puis, dans le menu contextuel, cliquez sur **Ajouter\Classe**.

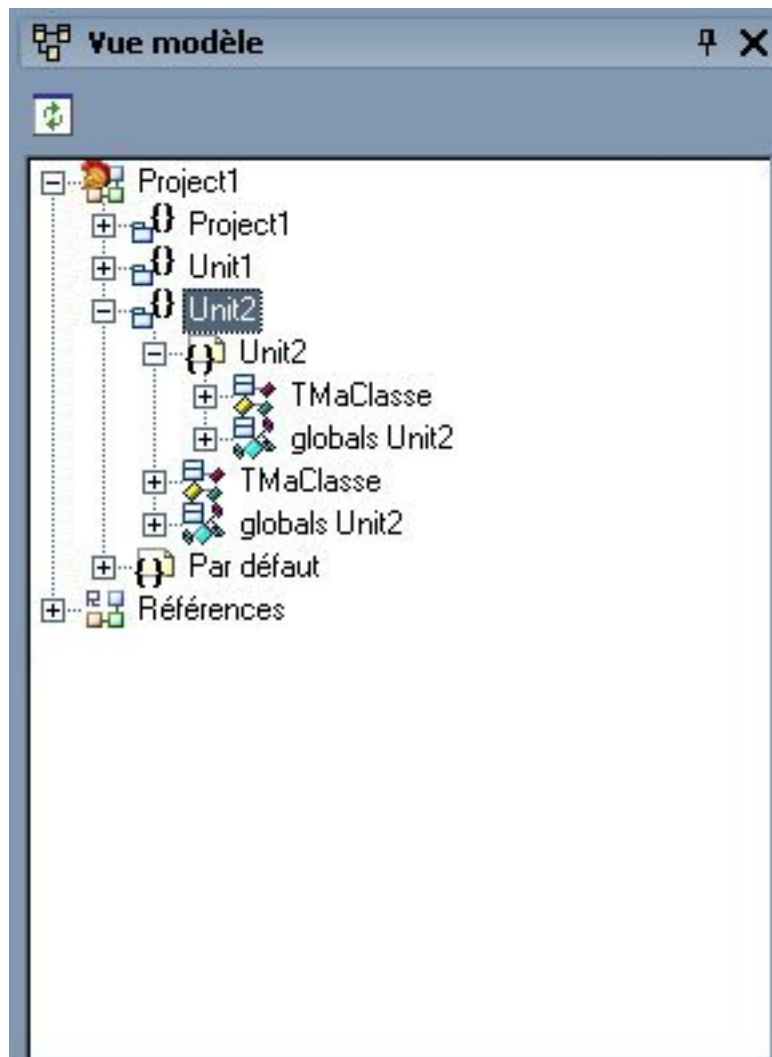


Ajouter une classe

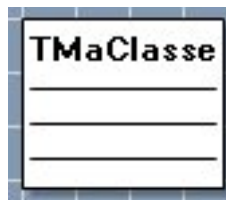
Together crée un nouveau noeud et vous permet de le nommer. Nommer et renommer ce noeud revient à nommer et renommer la classe qu'il représente. Nommons cette classe **TMaClasse**.

Si vous affichez le code de l'unité, vous verrez que Together a automatiquement ajouté la nouvelle classe.

En double-cliquant sur le noeud de la fiche contenant la nouvelle classe, vous pouvez afficher le diagramme représentant la classe.



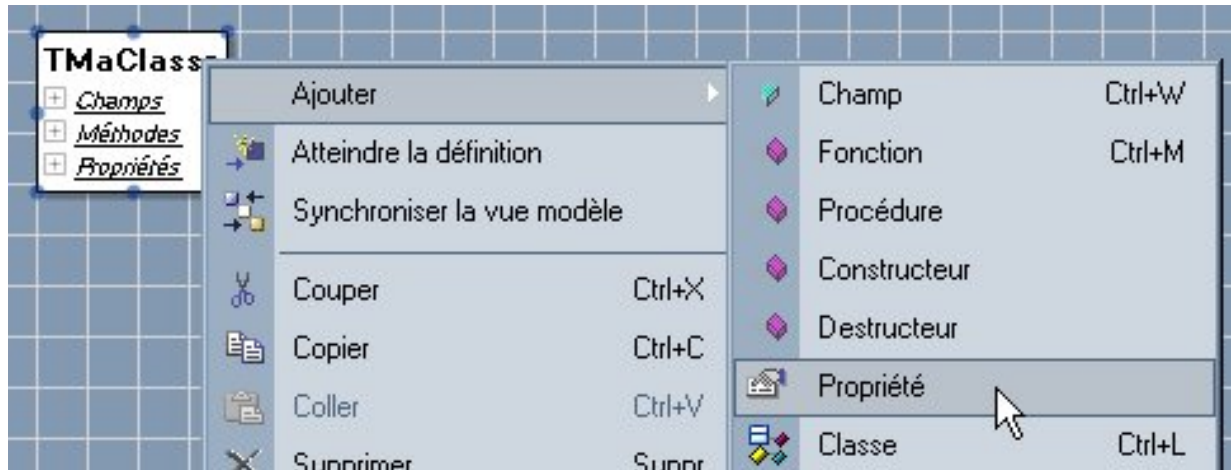
Double click sur le noeud de l'unité contenant la nouvelle classe (Ici: Unit2)



La classe TMaClasse dans la vue graphique

Définissons maintenant la propriété FileName.

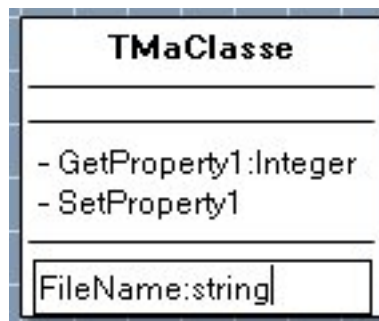
Sur le diagramme, cliquez sur le rectangle intitulé TMaClasse puis cliquez droit. Dans le menu, cliquez sur

Ajouter\Propriété:*Ajouter une propriété*

Together ajoute alors un nouvel élément à TMaClasse et vous permet de le nommer. Saisissez donc:

FileName:string

sans le point virgule.

*Saisie de la propriété*

Together ajoute alors automatiquement un "Getter" et un "Setter" qui sont respectivement GetFileName et SetFileName. Si vous êtes habitués au développement de composants, vous remarquerez que Together respecte à la lettre les conventions en la matière inhérentes au langage utilisé.

Si vous regardez maintenant le code, vous verrez que Together a généré la définition et l'implémentation de cette propriété, Getter et Setter compris:

```
TMaClasse = class
strict private
  procedure SetFileName(val : string);
  function GetFileName : string;

public
  property FileName : string read GetFileName write SetFileName;
end;
```

On peut remarquer que si l'on renomme le setter et/ou le getter dans la vue modèle ou dans la vue UML, la définition de la propriété n'est pas modifiée. Autrement dit, les méthodes n'ont pas de lien entre elles lors de la modélisation. Si une propriété dépend de méthodes, la modification de ces méthodes ne modifiera pas la déclaration de la propriété.

Par contre, la suppression d'une propriété dans ces vues supprime également les méthodes set et get dont elle dépend.

IV.3.b - Les méthodes

Ajoutons maintenant la méthode AppendToMemo. Cette procédure devra être publique et aura en paramètre, le TMemo à remplir. Elle aura pour déclaration:

```
procedure AppendToMemo(const Memo: TMemo);
```

Ouvrons donc le menu contextuel avec un click droit dans le rectangle de TMaClasse sur le label TMaClasse.



Label TMaClasse

En effet, le fait de cliquer sur ce label permet d'ajouter tous les types de membres possibles à la classe (variable, propriété, procédure, fonctions, etc.). Si l'on clique droit sur une fonction existante, par exemple, l'item **Ajouter** du menu contextuel devient **Ajouter une fonction** sans laisser la possibilité de créer autre chose.

Sélectionnons **AjouterProcédure**. De la même façon que la propriété FileName, nommez la méthode en AppendToMemo.

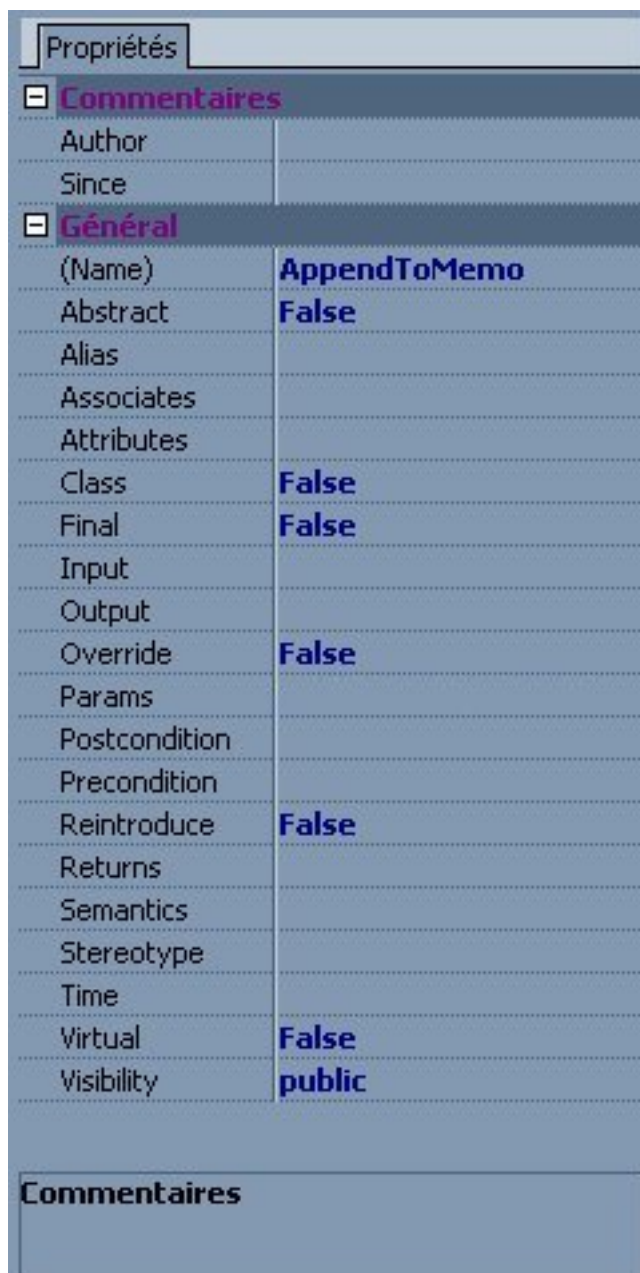
Together ajoute alors la nouvelle procédure dans le diagramme et dans le code.



Méthode AppendToMemo ajoutée

Toutefois, la procédure n'est pas complète: il manque le paramètre. Pour l'ajouter, nous allons utiliser l'inspecteur d'objet.

Tout comme des composants différents, les propriétés disponibles dans l'inspecteur d'objet sont différentes suivant le type d'objet auquel elles se réfèrent. Par exemple, une méthode a une propriété abstract qui n'existe pas pour les autres types de membres (propriété, champ, etc.) d'une classe. Ce qui est tout à fait normal.



Inspecteur d'objet de la méthode AppendToMemo

Lors de la conception de la fiche, cet outil est indispensable pour régler les paramètres des composants. Il l'est tout autant lors de la modélisation de votre classe.

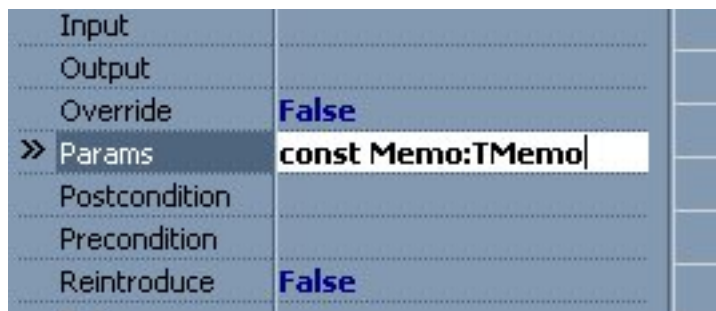
Vous pouvez spécifier pour la procédure AppendToMemo

- si c'est une méthode abstraite (Abstract)
- un alias (Alias)
- si c'est une procédure de classe (Class)
- si elle surcharge une autre méthode la méthode de son ancêtre (override)
- si elle surcharge et réintroduit des paramètres différents de la méthode de son ancêtre (reintroduce)
- si elle est virtuelle (Virtual)

- sa visibilité (Visibility)
- ses paramètres (Params)
- etc.

La propriété qui nous intéresse est la propriété Params. Spécifions donc à l'intérieur le paramètre tel que nous l'aurions saisi dans le code:

```
const Memo : TMemo
```



Spécification des paramètres de AppendToMemo

Le code est ainsi mis à jour.

IV.4 - Créer une classe dérivée d'une classe ancêtre

Certes, c'est bien joli de pouvoir créer des classes, mais c'est encore mieux si cette classe hérite d'une classe ancêtre.

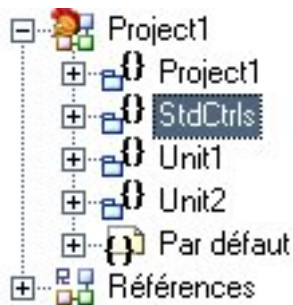
Pour cela, il existe une propriété de la classe: **extends**. Il faut comprendre "étendre" dans le sens que l'on veut étendre une classe existante qui sera la classe ancêtre.

Inspirons-nous de la [FAQ Delphi](#) et plus particulièrement: [Comment ne rentrer que des chiffres dans un TEdit ?](#) et créons une classe appelée TNumberEdit qui n'accepterait que les chiffres. Vous connaissez maintenant la méthode pour créer une classe: ajoutez donc une classe appelée TNumberEdit.

Dans l'inspecteur d'objet, vous avez le champ Extend. Il est impossible de taper directement le nom de la classe ancêtre. Pour ajouter cette classe, il faut tout d'abord ajouter au projet (**ProjetAjouter au projet...** ou Alt+F11) l'unité dans laquelle se trouve cette classe ancêtre.

Dans notre cas, la classe TEdit se trouve dans *[Répertoire de Turbo]\source\Win32\vc1\StdCtrls.pas*.

Une fois ce fichier ajouté au projet, vous pouvez voir qu'il s'est aussi ajouté à la vue modèle (hiérarchie) de Together.



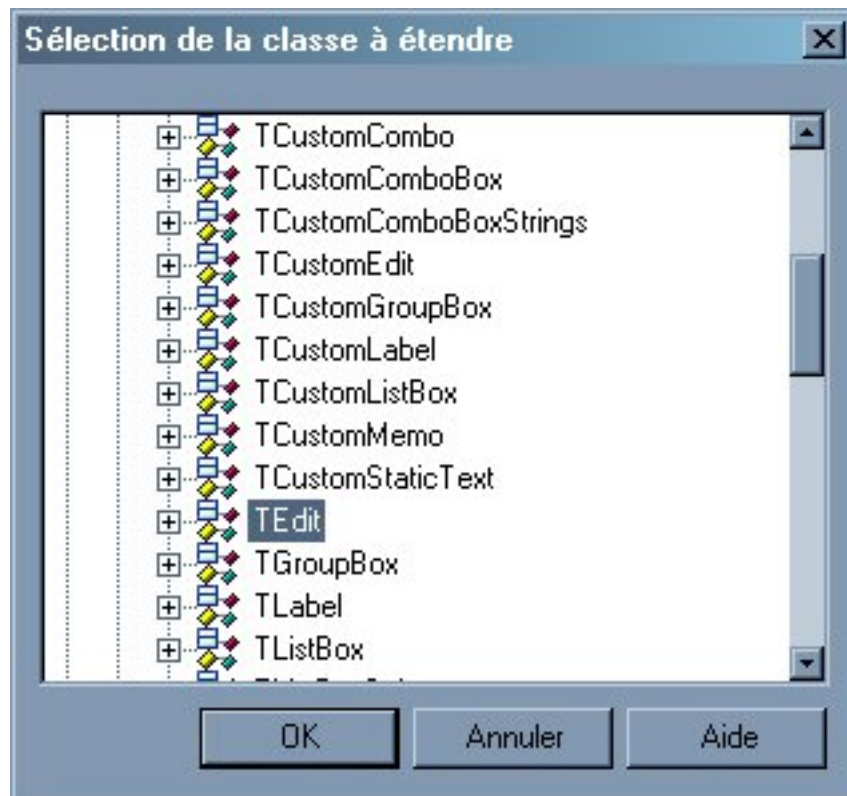
L'unité StdCtrls dans la vue modèle

On peut donc cliquer sur le petit bouton à droite de la propriété Extends de la classe TExtendedButton:



Bouton de sélection de la classe ancêtre

Ce bouton ouvre une boîte de dialogue contenant la vue modèle du projet en cours:



Sélection de la classe ancêtre dans l'unité StdCtrls

Sélectionnez comme le montre l'image ci-dessus, la classe TEdit contenue dans StrCtrls.pas.

Validez la boîte de dialogue et la vue UML, modèle ainsi que le code ont été mis à jour afin de refléter:

```
TNumberEdit = class(TEdit)
```

On peut également voir que dans la vue UML, une indication a été ajoutée afin d'informer que cette classe hérite d'une autre classe:



Indication de la classe ancêtre

Vous pouvez dès à présent surcharger, réintroduire et ajouter toutes sortes de méthodes héritées de TButton ou non. Pour qu'une méthode surcharge une autre, il suffit de mettre True à la propriété Override de cette méthode.

C'est là que le code fourni par la [FAQ Delphi](#) entre en jeu. On peut y lire qu'il faut utiliser l'événement OnKeyPress du TEdit. Dans notre cas, il nous faut donc surcharger la méthode KeyPress du TWinControl.

Pour cela, ajoutez donc une nouvelle procédure et appelez-la KeyPress. Dans l'explorateur d'objets, ajoutez:

```
var Key: Char
```

toujours sans le point-virgule. Finalement, passez sa propriété `Override` à `true` et changez sa visibilité à `Protected`. Retournez dans le code et recopiez le code fourni par la [FAQ Delphi](#):

```
if not (Key in ['0'..'9', DecimalSeparator, Chr(VK_BACK), Chr(VK_DELETE)]) then
  Key := #0;
if Key = DecimalSeparator then
  if Pos(DecimalSeparator, MyEdit.Text) <> 0 then
    Key := #0;
```

Voilà, vous disposez maintenant d'un TEdit personnalisé qui n'accepte que des chiffres en saisie.

Voici le code tel qu'il devrait être:

```
unit Unit2;

interface

uses
  ComCtrls, Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TNumberEdit = class(TEdit)
  protected
    procedure KeyPress(var Key: Char);override;
  end;

implementation

{ TNumberEdit }

procedure TNumberEdit.KeyPress(var Key: Char);
begin
  inherited;
  if not (Key in ['0'..'9', DecimalSeparator, Chr(VK_BACK), Chr(VK_DELETE)]) then
    Key := #0;
  if Key = DecimalSeparator then
    if Pos(DecimalSeparator, MyEdit.Text) <> 0 then
      Key := #0;
end;
end.
```

V - Conclusion

Etant donné la richesse de cet outil, chacune de ses fonctionnalités mériterait presque un tutoriel à part entière. Ce tutoriel n'est donc évidemment qu'une présentation sommaire de quelques possibilités de Together.

Les facilités qu'offre Together sont tout bonnement bluffantes, pour peu que l'on se penche sur l'outil. Il est de plus parfaitement intégré à Delphi ce qui est un atout indéniable.

Bref, pour finir, je ne pourrais que vivement conseiller l'utilisation de cet outil pour ces raisons:

- Un problème normalisé, précisé et compréhensible par tout le monde
- Une meilleure vue d'ensemble du projet
- Une restructuration très aisée grâce aux outils de refactoring
- Les autres avantages permis par la modélisation et exposés dans les liens donnés en introduction de ce tutoriel.
- Etc.

Pour ma part, je dois avouer avoir été séduit par l'efficacité de l'outil. Pour avoir créé des composants assez complexes, je peux dire que cet outil m'aurait évité bien des tests et des restructurations profondes de mon code. Il m'oblige à mieux penser POO (Programmation Orientée Objet) et ainsi, optimiser mon code au maximum.

Je tiens à remercier l'équipe Delphi de [developpez.com](#) et plus particulièrement Laurent Dardenne pour ses conseils plus qu'avisés et ses encouragements ainsi que Nip, Miles et Katyucha pour leurs corrections.

[Télécharger cet article \(fichier zip\).](#)

[Télécharger cet article au format pdf.](#)