# LiveBindings de A à … Écrire un composant observable I

Par Serge Girard

Date de publication : 14 avril 2019

CONFIRMÉ

Au cours de mon **introduction aux LiveBindings** j'avais utilisé un **VCL.TTrackBar** et en particulier sa propriété ***Position***. Nous y avions découvert que ce composant n'était pas observable et je vous avais alors proposé un contournement pour résoudre ce problème : l'utilisation d'un lien non géré et l'instruction **Notify.**

L'objectif de ce tutoriel est d'apprendre à rendre une propriété d'un composant observable et même d'ajouter d'autres propriétés que nous pourrons lier.

En complément sur Developpez.com

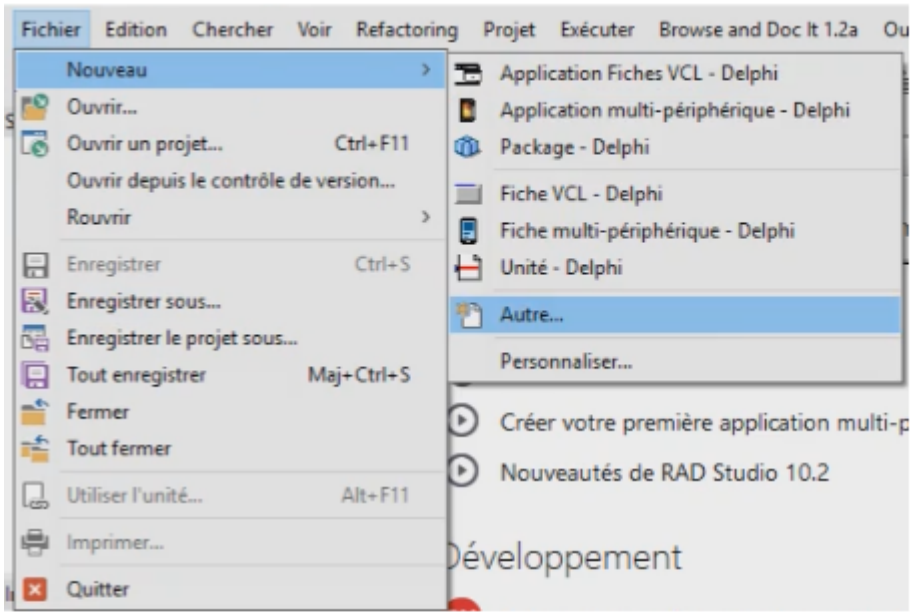
- Création de composants de Sébastien Doeraene

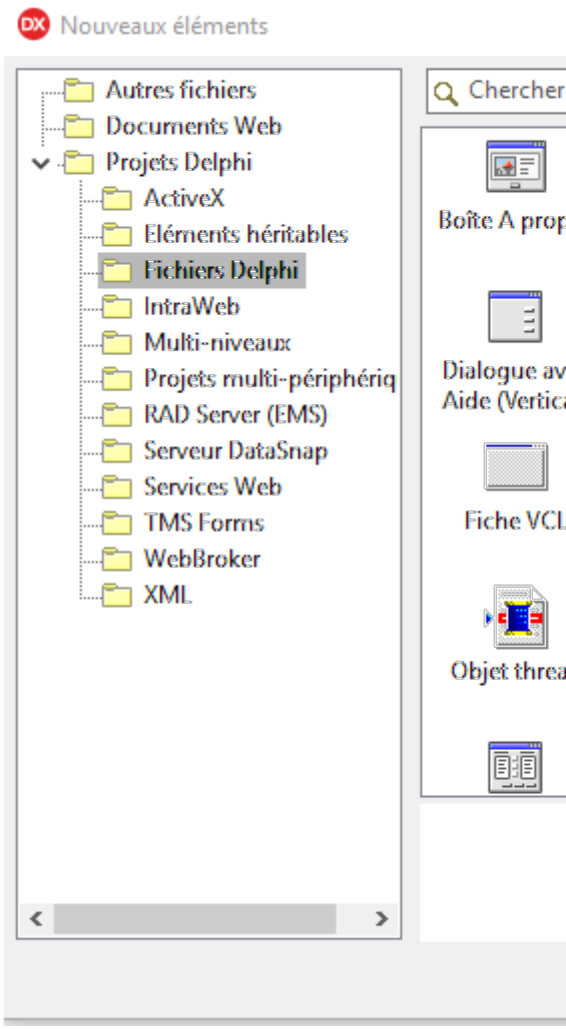
# I - Introduction

L'objectif premier de ce tutoriel n'est pas de construire des composants de A à Z, ce qui est parfaitement décrit dans d'autres tutoriels comme ceux rassemblés **ici** par **Sébastien Doeraene**. Il s'agit plutôt d'améliorer un composant existant en héritant de celui-ci comme décrit dans **la partie II de ce tutoriel**.

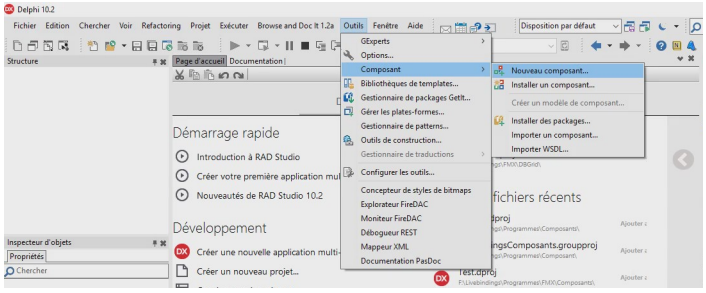
La tâche est facilitée par l'expert de création de composant, invocable de deux manières :

**Fichier/Nouveau/Autre…**

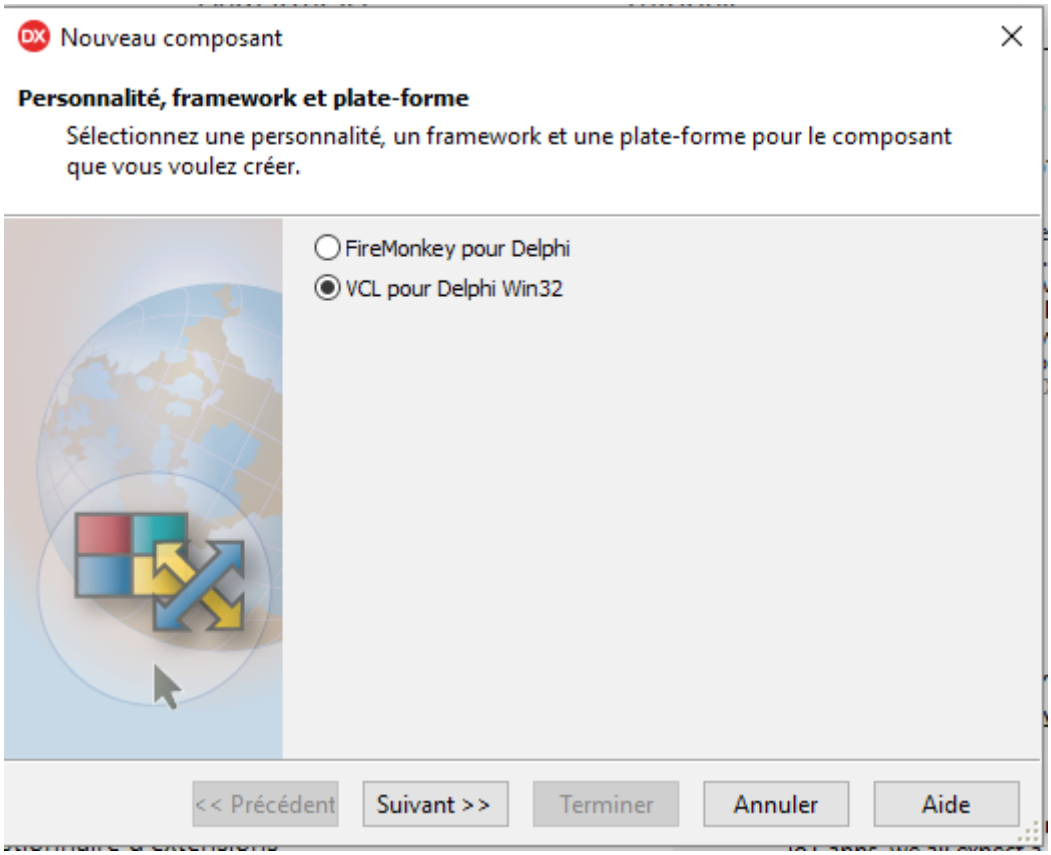



ou **Outils/Composants/Nouveau composant…**



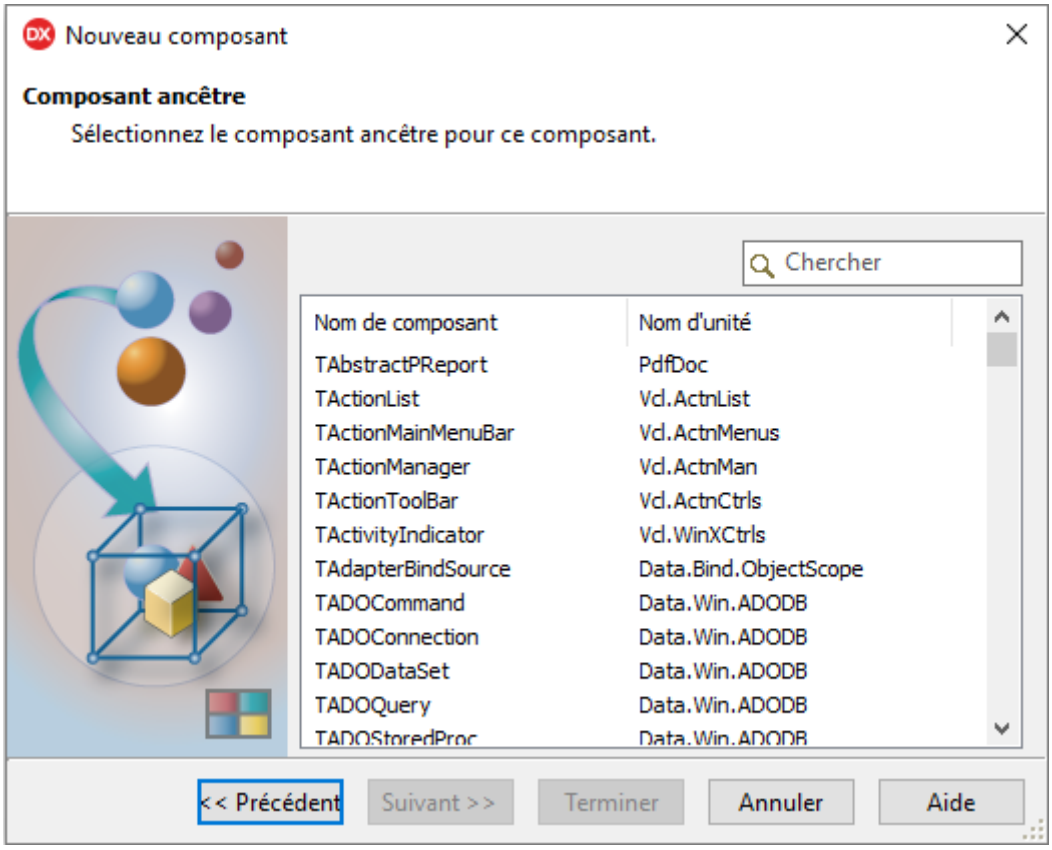
Nous devons ensuite indiquer quel sera le framework cible :




*debutzoom[F:\Livebindings\Article 4 Composant\Images\NouveauComposantVCL.PNG]*

*Il est à noter que si un framework est déjà chargé, autrement dit si un projet est déjà ouvert, cette information n'est pas demandée. Le composant créé sera du même environnement que le projet.*

Pour ajouter l'observateur, nous ne pouvons (devons) pas modifier le composant d'origine : il faut donc créer une nouvelle classe, descendante de ce composant.




*debutzoom[F:\Livebindings\Article 4 Composant\Images\ancetre.png]*

La dernière étape permet de fournir à l'expert les informations nécessaires à la construction du code source, à savoir :

- le nom du composant (classe) ;
- la palette dans laquelle il va se trouver ;
- le nom de l'unité source à générer.


*À propos du nom de l'unité : il est tout à fait possible de nommer cette dernière en la préfixant par le framework (exemples : **VCL.mounite.pas** ou **FMX.monunite.pas**), et ce à la manière très pratique utilisée par Embarcadero.*

*Tant que cela reste un composant interne à votre organisation, cela ne pose aucun problème. Toutefois, Embarcadero ne recommande pas, voire pas du tout, l'utilisation de ces préfixes pour des composants que vous pourriez distribuer.*

## II - Rendre un composant actif

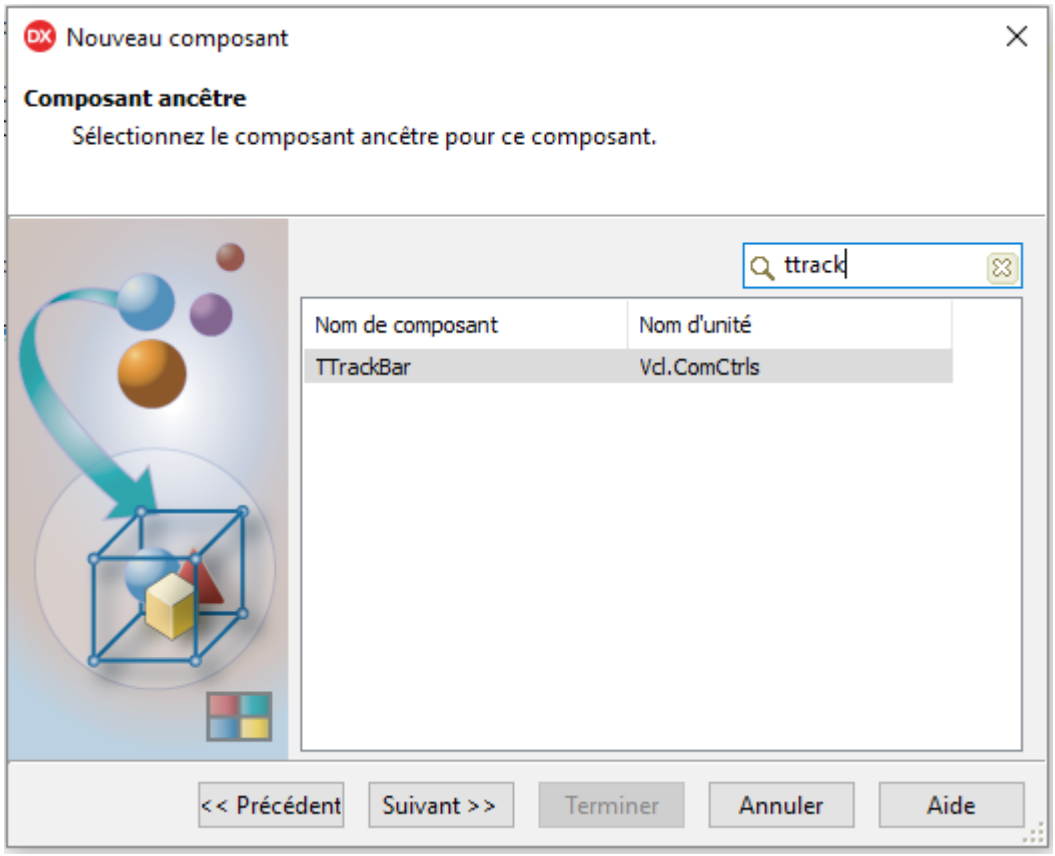
Comme un tutoriel existe déjà dans la documentation d'Embarcadero (**création de composant activé**), nous allons simplement étudier le déroulement de cette création d'un composant activé.

En résumé, par rapport à une création classique (d'avant les **Livebindings**), deux exigences doivent être remplies :

- le contrôle doit implémenter le support des observateurs, car **LiveBindings** dépend des observateurs pour l'abonnement aux notifications des contrôles ;
- vous devez recenser le nom de la valeur du contrôle. Ce dernier est utilisé par les composants **LiveBindings** pour générer des expressions qui obtiennent et définissent la valeur du contrôle.

### II-A - Le nouveau composant TObservableTrackBar

Je vais dériver le composant d'origine **TTrackBar** afin d'ajouter l'observateur. Nous devons donc créer une nouvelle classe, descendante de ce composant.




## II-A-1 - Ajouter un attribut

La première étape est d'ajouter un attribut **ObservableMember** (ligne 10) juste avant la déclaration de cette nouvelle classe. Cet attribut sera utilisé par **LiveBindings** pour générer les expressions.

Partie interface

```
1.  unit VCL.TrackBarObservable;
2.
3.  interface
4.
5.  uses
6.    Vcl.ComCtrls, System.Classes, WinApi.Messages, WinApi.CommCtrl, Vcl.Controls;
7.
8.  type
9.
10.   [ObservableMember('Position')] // Attribut, identifie le nom de la valeur à observer
11.   TObservableTrackBar = class(TTrackBar)
12.   private
13.     procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
14.     procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
15.     procedure ObserverToggle(const AObserver: IObserver; const Value: Boolean);
16.   protected
17.     // déclarées dans System.Classes elles doivent être écrasées (override)
18.     function CanObserve(const ID: Integer): Boolean; override;
19.     procedure ObserverAdded(const ID:  Integer; const Observer: IObserver);  override;
20.   end;
```

## II-A-2 - Rendre observable

La seconde étape consiste à indiquer que le composant peut être observé. C'est le rôle de la fonction **CanObserve** de la ligne 18 de la partie interface. Sans elle, la modification du composant n'a pas lieu d'être.

CanObserve
```
function CanObserve(const ID: Integer): Boolean;
begin
 case ID of //est-ce que la destination et la source sont compatibles ?
    TObserverMapping.EditLinkID,
    TObserverMapping.ControlValueID:
      Result := True;
  else
    Result := False;
  end;
end;
```

## II-A-3 - Ajouter un observateur

La troisième étape ajoute la procédure **ObserverAdded** et, pour la méthode d'articulation associée, **ObserverToggle**. Ces deux méthodes sont en fait écrites pour le concepteur visuel de **LiveBindings**.

Observer
```
procedure TObservableTrackBar.ObserverAdded(const ID: Integer; const Observer: IObserver);
begin
  if ID = TObserverMapping.EditLinkID then
    Observer.OnObserverToggle := ObserverToggle;
end;

procedure TObservableTrackBar.ObserverToggle(const AObserver: IObserver; const Value: Boolean);
var
  LEditLinkObserver: IEditLinkObserver;
begin
  if Value then
  begin
// désactive le composant si le champ associé est en écriture seule
    if Supports(AObserver, IEditLinkObserver, LEditLinkObserver) then
      Enabled := not LEditLinkObserver.IsReadOnly;
  end else
    Enabled := True;
end;
```

## II-A-4 - À quoi servent ces deux dernières étapes ?

Pour nos programmes qui vont utiliser ce composant, la réponse est catégorique : les deux dernières étapes ne serviront à rien ! Nous pouvons même nous passer de ces deux procédures puisque, contrairement à la procédure **CanObserve** qui est indispensable, notre composant fonctionnera quand même sans elles ! Ce n'est qu'au niveau du concepteur visuel de liaisons, donc dans l'EDI, que ces méthodes sont utilisées.

*Pour vous en convaincre, commentez ces deux dernières méthodes puis réinstallez le composant. A priori, le comportement est identique !*

## II-A-4-a - Comment cela fonctionne ?

Vous avez certainement remarqué que les méthodes **CanObserve** et **ObserverAdded** ont toutes deux un argument **ID.** C'est cet identifiant qui permet d'obtenir les informations de type et dans le cas d'**ObserverAdded** de faire le lien avec la méthode **ObserverToggle**.

*Pour imager davantage :c'est ce qui fait que le concepteur visuel accepte ou refuse une liaison entre deux composants, voire entre deux propriétés de ceux-ci, et détermine le type de flèche qui en résulte (bidirectionnelle ou non).*

## II-A-5 - Notifier les changements

La dernière étape créera les méthodes (lignes 13 et 14 de la partie interface) qui vont surveiller les valeurs, en l'occurrence la position du curseur du **TrackBar**, de façon à alerter l'observateur. Comme le **TrackBar** peut être horizontal ou vertical, il nous en faut une par mode.

Ces méthodes écrasent celles de la classe parente. Ce qui est à retenir, c'est que la méthode **TLinkObservers.ControlChanged** permet d'indiquer qu'il y a eu un changement.

surveillance
```
1. // le code est identique pour ces deux procédures
2. var
3.   LPosition: Integer;
4. begin
5.   LPosition := Position; // récupère la position précédente
6.   inherited; // appel de la méthode parente
7.   if LPosition <> Position then  // vérifie s'il y a eu déplacement
8.      TlinkObservers.ControlChanged(Self); // notifie le changement
9. end ;
```

## II-B - Recenser le composant

Pour pouvoir utiliser ce composant, il faut bien sûr l'installer ! C'est l'objectif de la procédure **register** qui permet d'ajouter le composant à la palette. Mais il faut aussi recenser ce dernier pour le concepteur visuel. La propriété du nom de valeur du composant (dans notre cas 'Position') doit être signalée en utilisant la méthode **RegisterObservableMember**.

Recensement composant
```
unit VCL.TrackBarObservableReg;

interface

procedure Register;

implementation

uses
  System.Classes, VCL.TrackbarObservable, Data.Bind.Components;


procedure Register;
begin
  RegisterComponents('LiveBindings Samples', [TObservableTrackBar]);
end;

initialization

Data.Bind.Components.RegisterObservableMember(TArray<TClass>.Create(TObservableTrackBar), 'Position', 'DFM');

finalization

Data.Bind.Components.UnregisterObservableMember(TArray<TClass>.Create(TObservableTrackBar));

end.
```

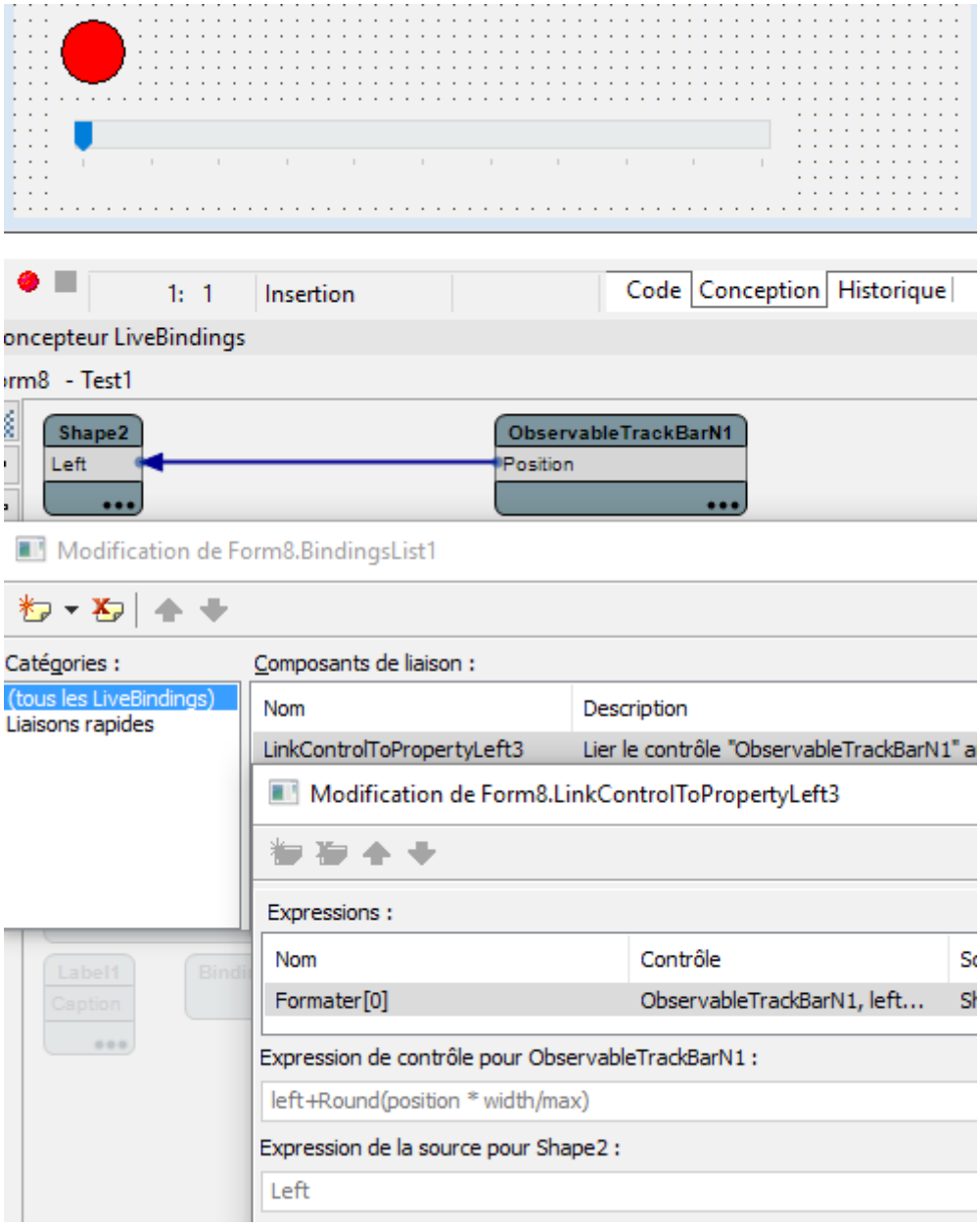
Reste à installer le composant : ce sera chose faite en créant un nouveau package ou en ajoutant les sources à un paquet déjà existant.

Vous remarquerez que j'indique **les** sources et non **le** source. En effet le tutoriel de la documentation en présente deux et non un seul, la partie recensement du composant étant séparée de la partie source de la nouvelle classe.

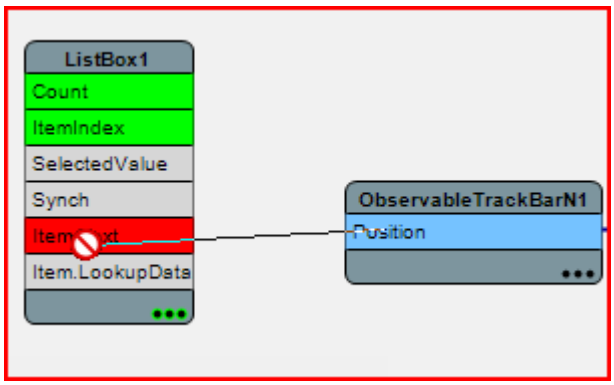
| | |
|---|---|
| | Or, si vous utilisez l'expert, vous n'obtenez qu'un seul source fusionnant à la fois la classe et le recensement !<br>Les deux modes fonctionnent, mais séparer le recensement du reste est ce que je recommanderais. |

Une fois le paquet installé, un programme test nous permettra de vérifier :

1 Que le concepteur visuel est fonctionnel.




2 Que le composant est observé (*cf.* mon introduction aux **LiveBindings**).

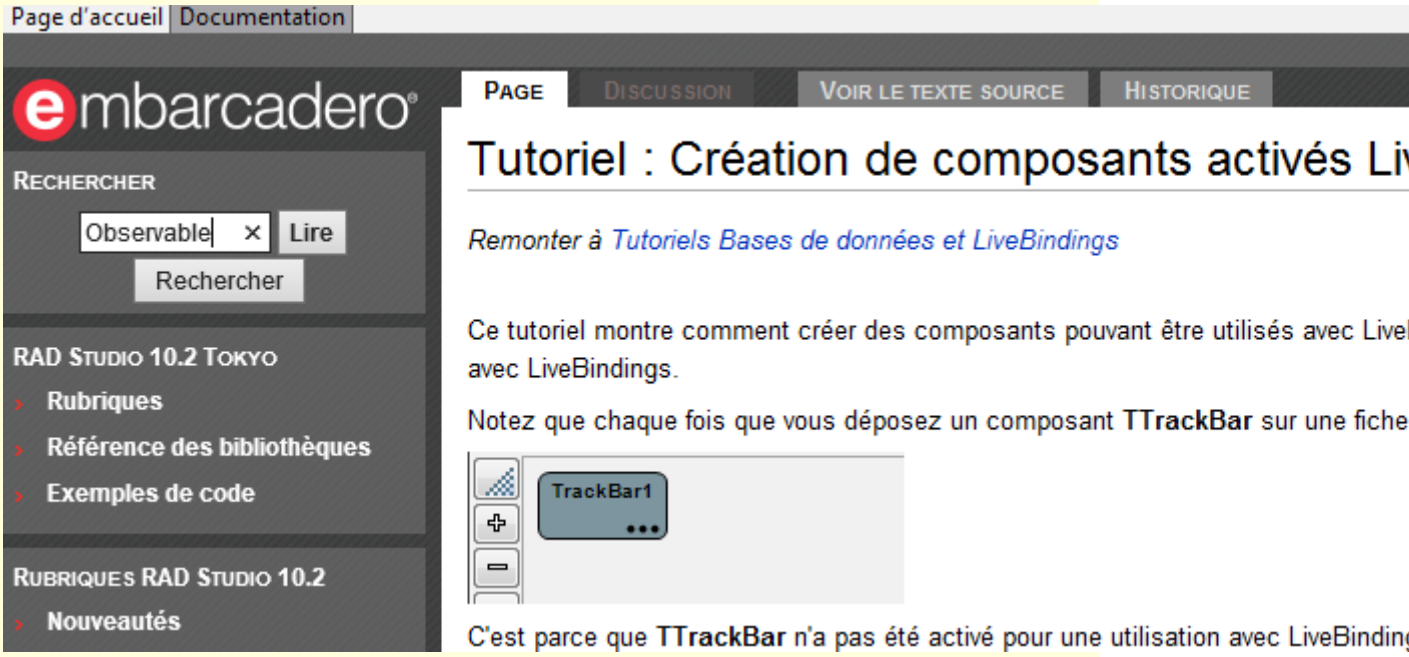



*Pour cette partie, je ne fournirai pas les sources. Je préfère vous inciter à suivre les étapes de la **documentation** à la lumière de ce chapitre.*

*Je vous incite même à afficher cette documentation dans votre EDI.*

*Recherchez « Observable » puis sélectionnez le tutoriel.*

*Vous pourrez alors facilement faire des opérations de copier-coller sans sortir de l'EDI.*




## III - Application à un autre composant

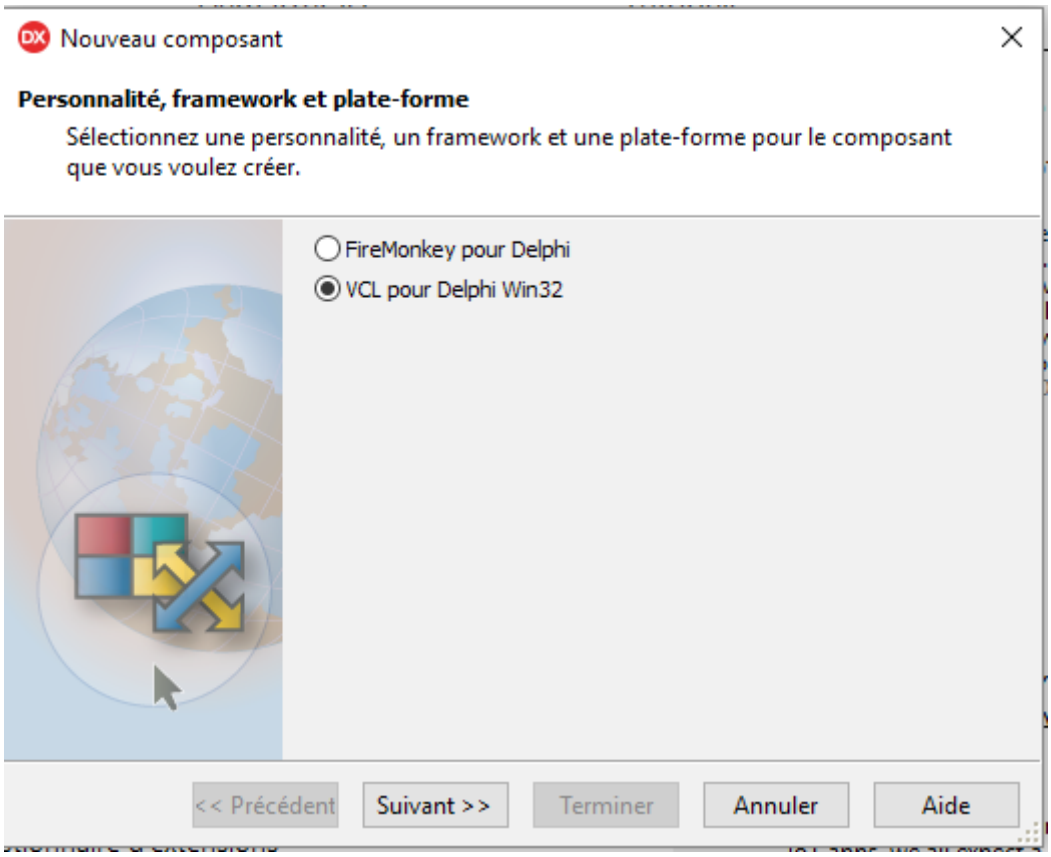
Remâcher le tutoriel de la documentation, c'est bien pour en comprendre le principe ; essayer d'appliquer ces principes à un autre composant va certainement être plus formateur.

Comme composant d'entraînement, je vais utiliser le **TImage**. Il est facile de lier la propriété **Picture** à une image contenue dans une colonne de type binaire (**blob**) d'une table. En revanche, il n'y a pas de propriété contenant le chemin vers une image et qui permettrait de charger celle-ci. Or, par souci de taille, sauf cas particulier, je préfère indiquer le chemin et le nom de l'image dans mes bases de données.

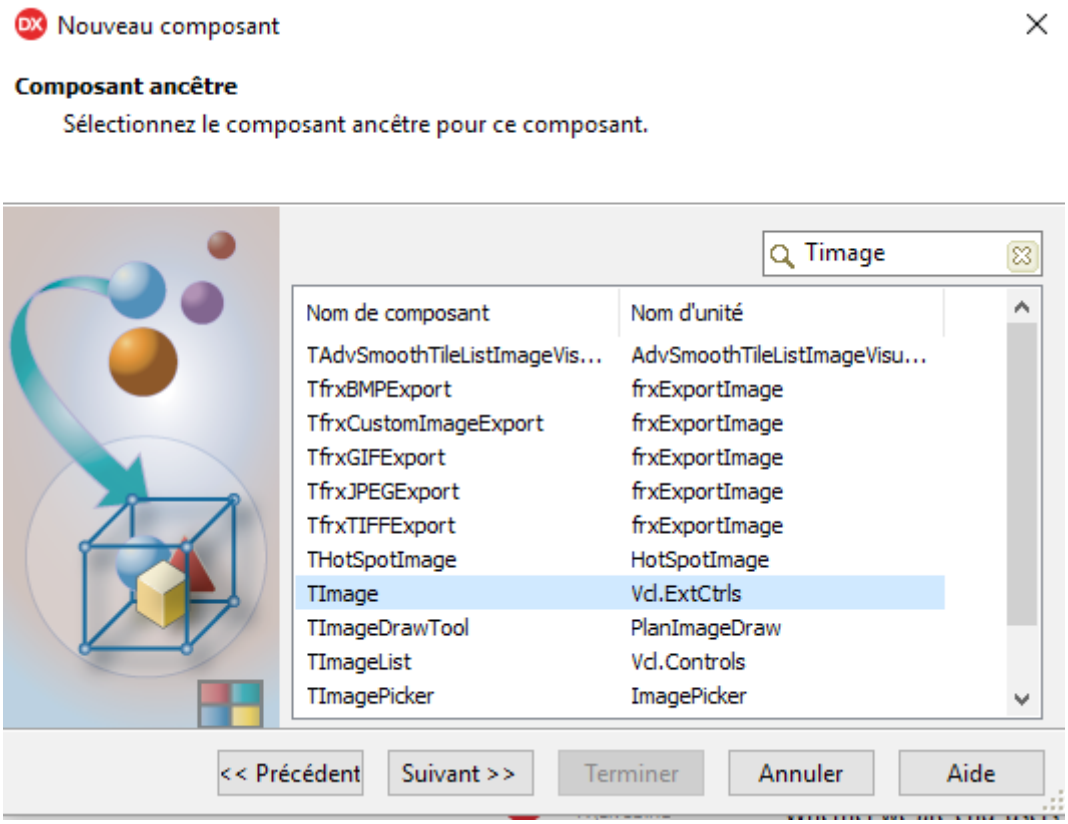
Le challenge est donc posé : ajouter une propriété qui contiendra le nom de fichier et faire en sorte que le moteur des **LiveBindings** la gère.

## III-A - Créer le composant

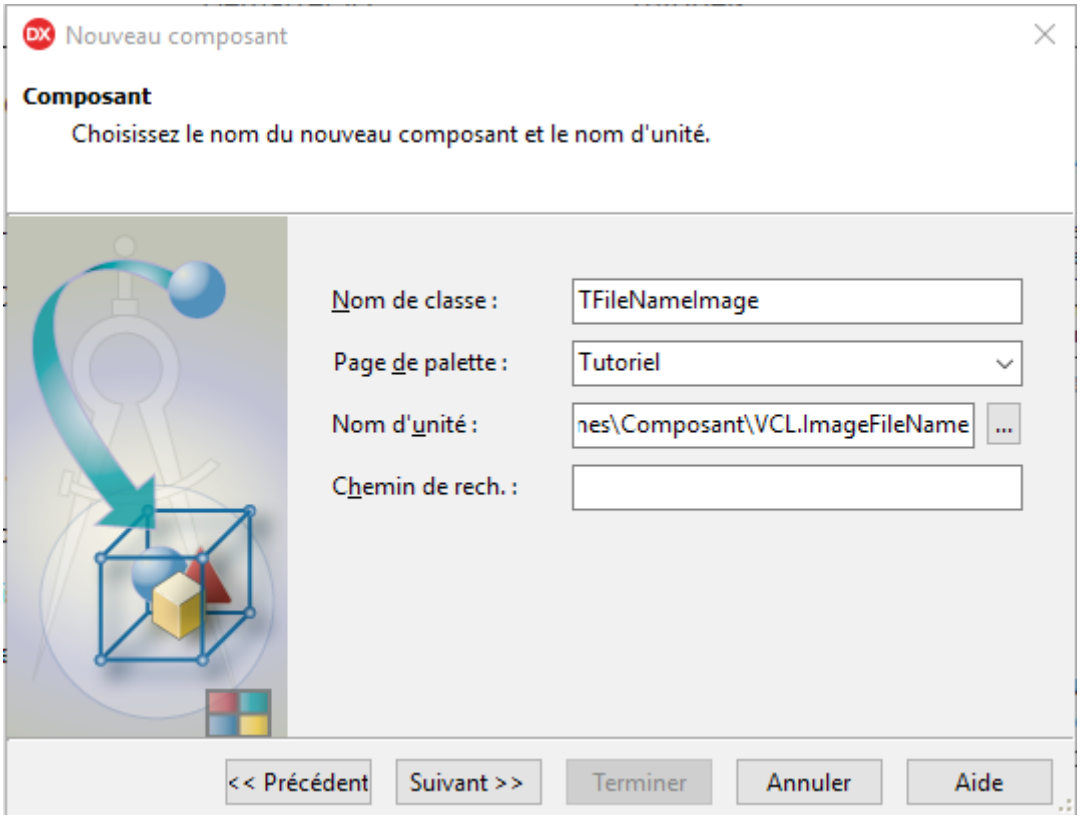
La création du composant s'opère à partir de l'expert :



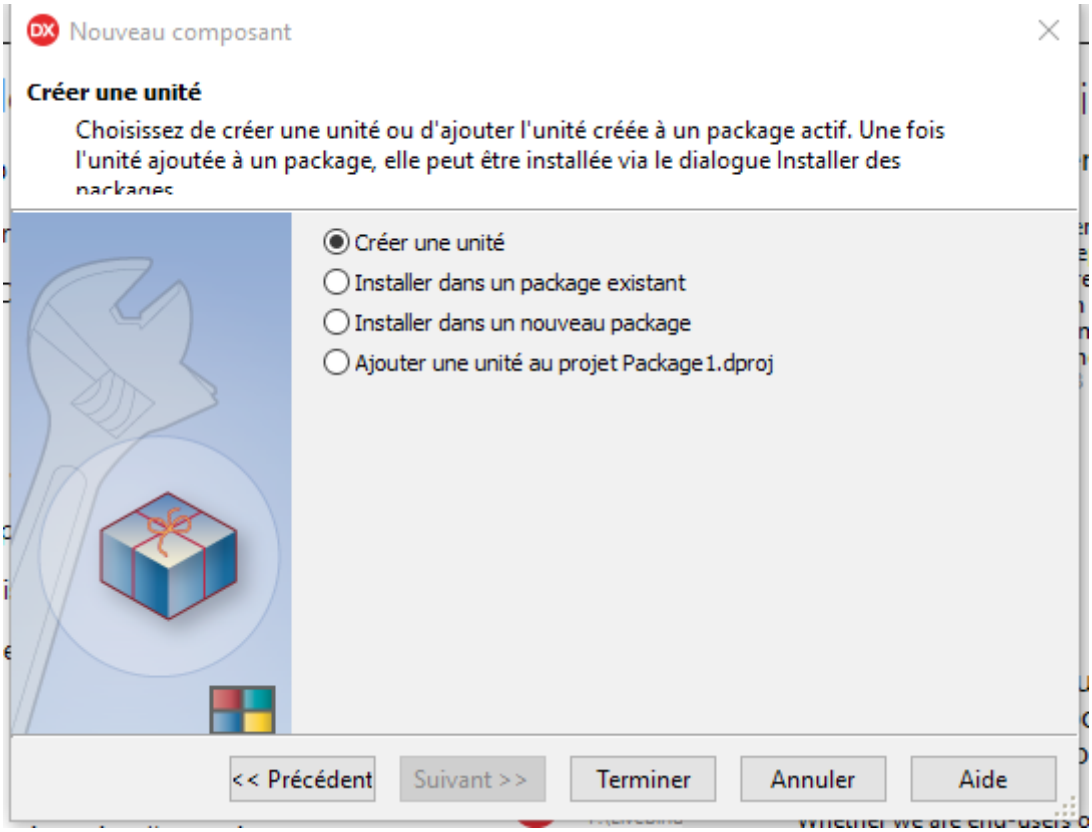

*debutzoom[F:\Livebindings\Article 4 Composant\Images\NouveauComposantVCL.PNG]*




*debutzoom[F:\Livebindings\Article 4 Composant\Images\ComposantVCL_2.PNG]*




*debutzoom[F:\Livebindings\Article 4 Composant\Images\ComposantVCL_3.PNG]*




*debutzoom[F:\Livebindings\Article 4 Composant\Images\ComposantVCL_4.PNG]*

J'obtiens un premier source qu'il me sera ensuite possible de scinder en deux unités distinctes, une pour la classe et une pour le recensement.

*Cette scission n'est pas une obligation, mais gardez cette possibilité en mémoire, car elle nous sera très utile si notre choix se porte sur la création d'un composant mixte (VCL et FMX).*

Je suivrai les mêmes étapes que dans le premier chapitre : il me faut dériver la classe **TImage**, y ajouter la propriété, la mettre en attribut et rendre celle-ci observable.

interface
```
unit ImageFileVCL;

interface

uses
  Vcl.ComCtrls, System.Classes, WinApi.Messages, WinApi.CommCtrl,
  Vcl.Controls, VCL.ExtCtrls, System.SysUtils;
type
  [ObservableMember('NomFichierImage')]
  TFileImageVCL = class(TImage)
  private
    FNomFichierImage : String;
    procedure ObserverToggle(const AObserver: IObserver; const Value: Boolean);
  protected
    function CanObserve(const ID: Integer): Boolean; override;
    procedure ObserverAdded(const ID: Integer; const Observer: IObserver); override;
    procedure DrawImage(Value : String);
  published
    property NomFichierImage : string read FNomFichierImage write DrawImage;
  end;
```

La principale différence avec le chapitre **II.A.1** est donc l'ajout d'une propriété (**NomFichierImage**) et le traitement de celle-ci lorsqu'elle est changée (procédure **DrawImage**).

DrawImage
```
procedure TFileImageVCL.DrawImage(Value : String);
var OldValue : String;
begin
if FNomFichierImage<>Value then
 begin
  FNomFichierImage:=Value; // modifie la valeur de la propriété
  try
    Picture.LoadFromFile(Value); // charge l'image
  except
    Picture:=nil; // efface l'image si le fichier n'existe pas
  end;

  TlinkObservers.ControlChanged(Self); // notification du changement
 end;
end;
```

Il s'agit à présent de rendre observable la propriété et d'ajouter un observateur. La méthode est la même que lors des étapes 2 et 3 du chapitre précédent. Je résume ce qu'il faut faire :

- ajouter un attribut [ObservableMember('xxxx')] **II.A.1** ;
  rendre observable par l'ajout de la fonction **CanObserve** **II.A.2** ;
- ajouter l'observateur pour le concepteur visuel par l'intermédiaire des procédures **ObserverAdded** et **ObserverToggle** **II.A.3**

## III-B - Recenser le composant

Si nous nous en tenons au chapitre **II.B**, l'écriture du recensement du composant est simple.

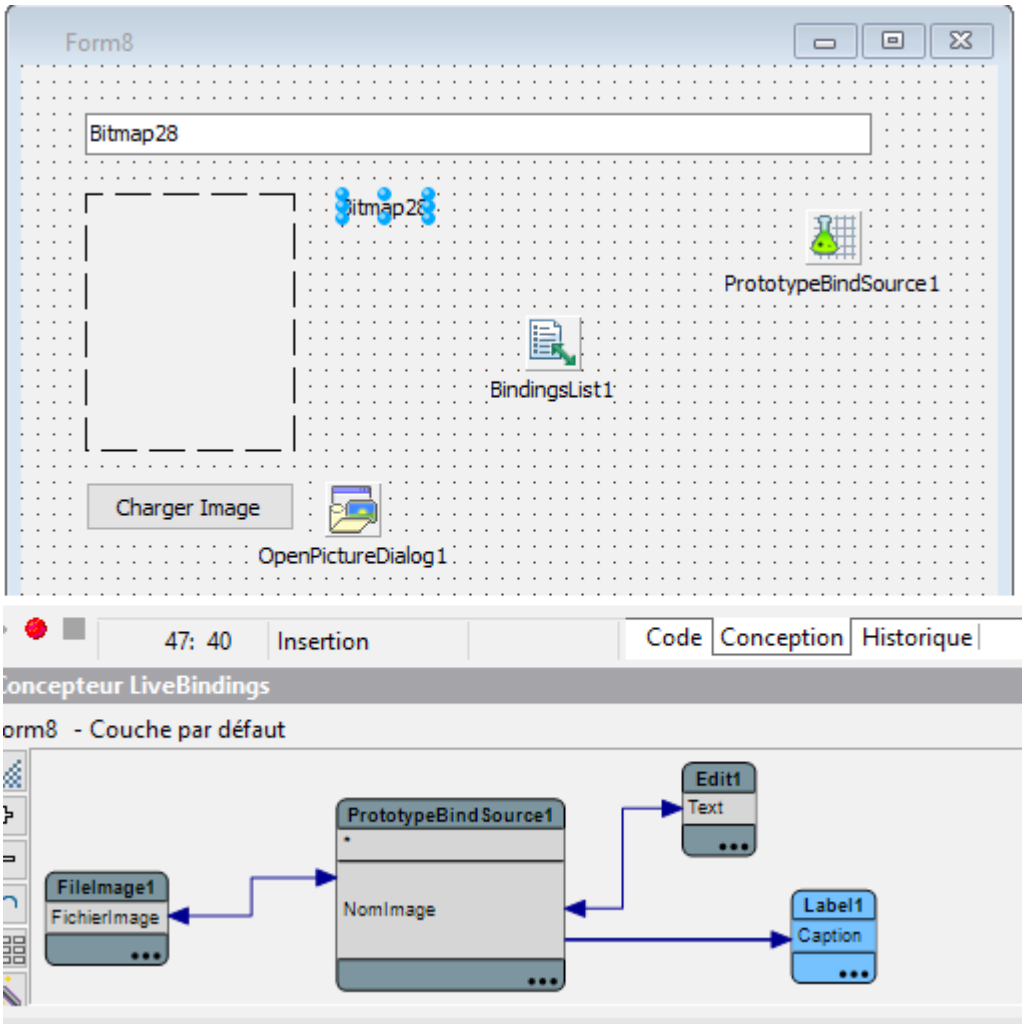
Recensement TFileImage
```
unit RegImageFile;

interface
uses System.Classes, System.SysUtils, VCL.Controls, VCL.Graphics,
     Data.Bind.Components,
     System.Bindings.Outputs, System.rtti, System.TypInfo,
     ImageFileVCL;

procedure Register;

implementation

procedure Register;
begin
  GroupDescendentsWith(TFileNameImageVCL, Vcl.Controls.TControl);
  RegisterComponents('Tutoriels', [TFileNameImageVCL]);
end;

initialization
 Data.Bind.Components.RegisterObservableMember(TArray<TClass>.Create(TFileNameImageVCL), 'NomFichierImage','DFM')

finalization
  Data.Bind.Components.UnregisterObservableMember(TArray<TClass>.Create(TFileNameImageVCL));
end.
```

## III-C - Programme test

Après avoir installé le composant dans notre palette (1) , il est aisé de créer un petit programme test.

*Trop impatient, j'ai mis la charrue avant les bœufs ! Le bon usage aurait été de tester le composant sans que celui-ci soit installé afin de faire un premier essai en le créant à l'exécution (**runtime**) et, en cas de satisfaction, installer ensuite le package. Toutefois il est assez délicat de créer des liens directement à l'exécution et tester les Livebindings sans installation, sans parler du fait que l'on ne pourrait y voir les impacts sur l'EDI.*

L'écran comprend quatre composants visuels : notre nouveau composant **TFileImage**, un bouton, une zone de saisie et un simple libellé. En composants non visuels, nous avons un **TOpenPictureDialog** que nous utiliserons en relation avec le bouton pour charger une image, un **TPrototypeBindSource** qui simulera notre accès à une base de données et pour les liaisons un **TBindingsList** pour établir les différentes liaisons, comme vous pouvez le voir sur l'image suivante.




Le seul code à écrire est celui concernant le comportement du bouton.

```
procedure TForm8.Button1Click(Sender: TObject);
begin
if OpenPictureDialog1.Execute then
  FileImage1.FichierImage:=OpenPictureDialog1.FileName;
end;
```

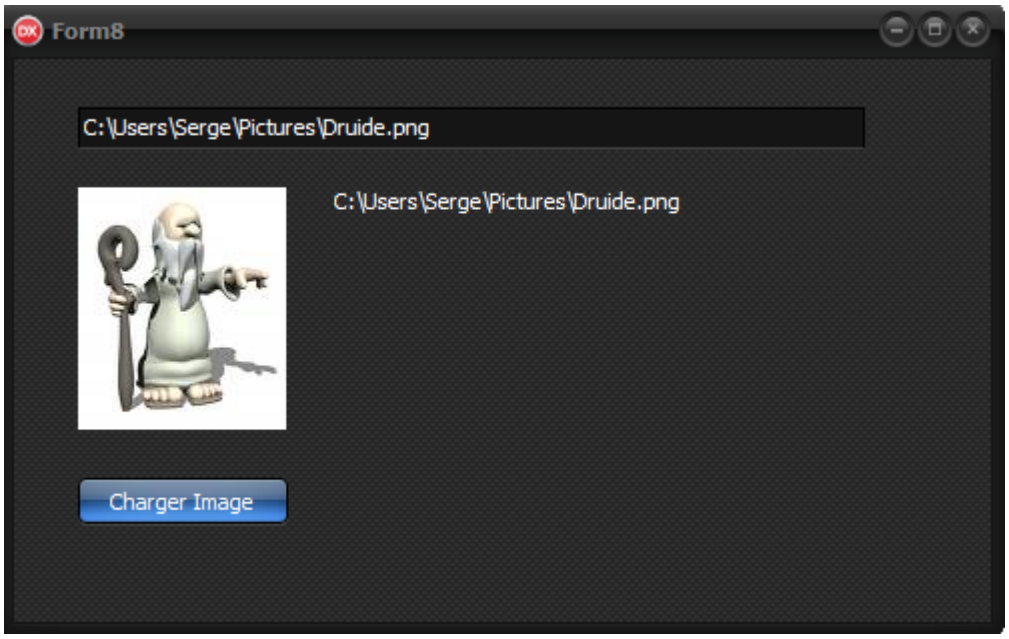
*Ce code sera avantageusement remplacé par :*

```
   if FileOpenDialog1.execute  then
```

```
    begin
      PrototypeBindSource1.Edit;
PrototypeBindSource1.InternalAdapter.FindField('NomImage').SetTValue(OpenPictureDialog1.FileName);
      PrototypeBindSource1.Post;
    end;
```

*Qui « joue » sur directement sur les données et ainsi s'applique à tous les liens.*

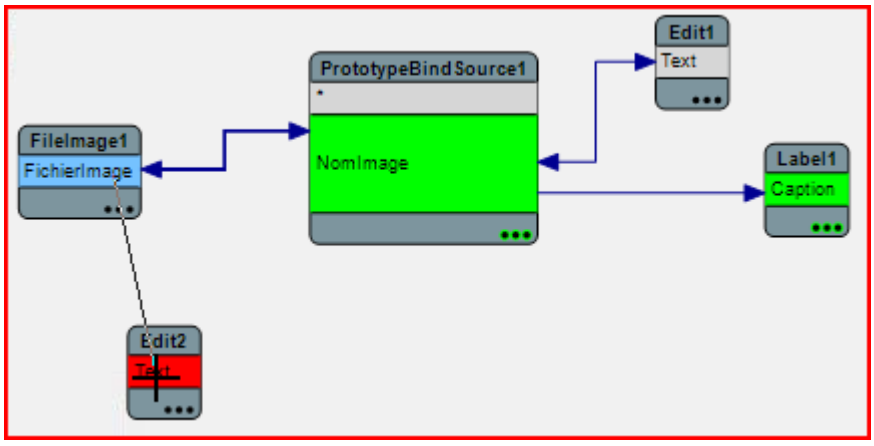
À l'exécution, le programme est conforme à notre attente : la saisie dans la zone de texte, une fois le changement de zone effectué, charge l'image et modifie le libellé.




*J'ai utilisé un style (**Glossy**) pour avoir un rendu plus sympathique.*

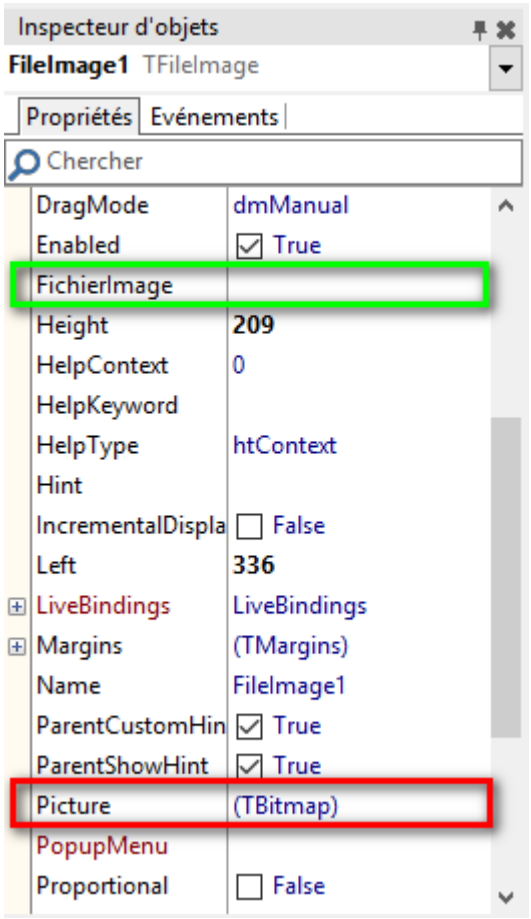
Cela pourrait donc nous suffire, toutefois il reste quelques difficultés à résoudre.

- Comme le montre l'image suivante, nous ne pouvons pas lier directement l'image à une zone de saisie (***Edit2***). Nous avons cependant la solution de passer par un **TProtypeBindSource**. Le champ ***NomImage*** est lié à la propriété ***FichierImage*** de mon composant, mais aussi à ma zone de saisie ***Edit1***.



*Gardez cette technique de l'utilisation d'un **PrototypeBindSource** en mémoire. En effet, passer par cet intermédiaire est un moyen facile de lier les propriétés de deux composants.*

- La nouvelle propriété **FichierImage** du composant ne nous permet pas de faire une recherche sur le disque.



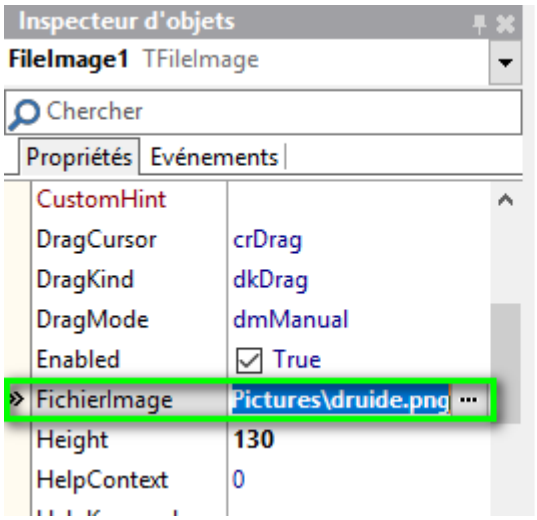

Pour corriger immédiatement ce dernier point, il faut changer le type de **FNomFichierImage** en **TFileName**, ce qui entraîne la correction de la propriété **FichierImage** et donc de la procédure **DrawImage**.

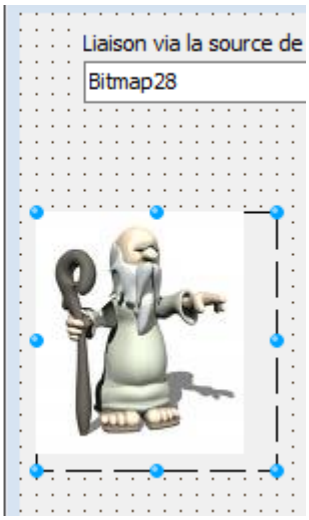
Corrections

```
uses
  Vcl.ComCtrls, System.Classes, WinApi.Messages, WinApi.CommCtrl,
  Vcl.Controls, VCL.ExtCtrls, System.SysUtils;
type
  [ObservableMember('NomFichierImage')]
  TFileNameImageVCL = class(TImage)
  private
    { Déclarations privées }
    FNomFichierImage : TFileName;
    FUseEvalShortcuts: Boolean;
    procedure ObserverToggle(const AObserver: IObserver; const Value: Boolean);
  protected
    { Déclarations protégées }
    procedure DrawImage(Value : TFileName); overload;
//    procedure DrawImage(Value : String); overload;
    function CanObserve(const ID: Integer): Boolean; override;
    procedure ObserverAdded(const ID: Integer; const Observer: IObserver); override;
  public
    { Déclarations publiques }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
```

Corrections

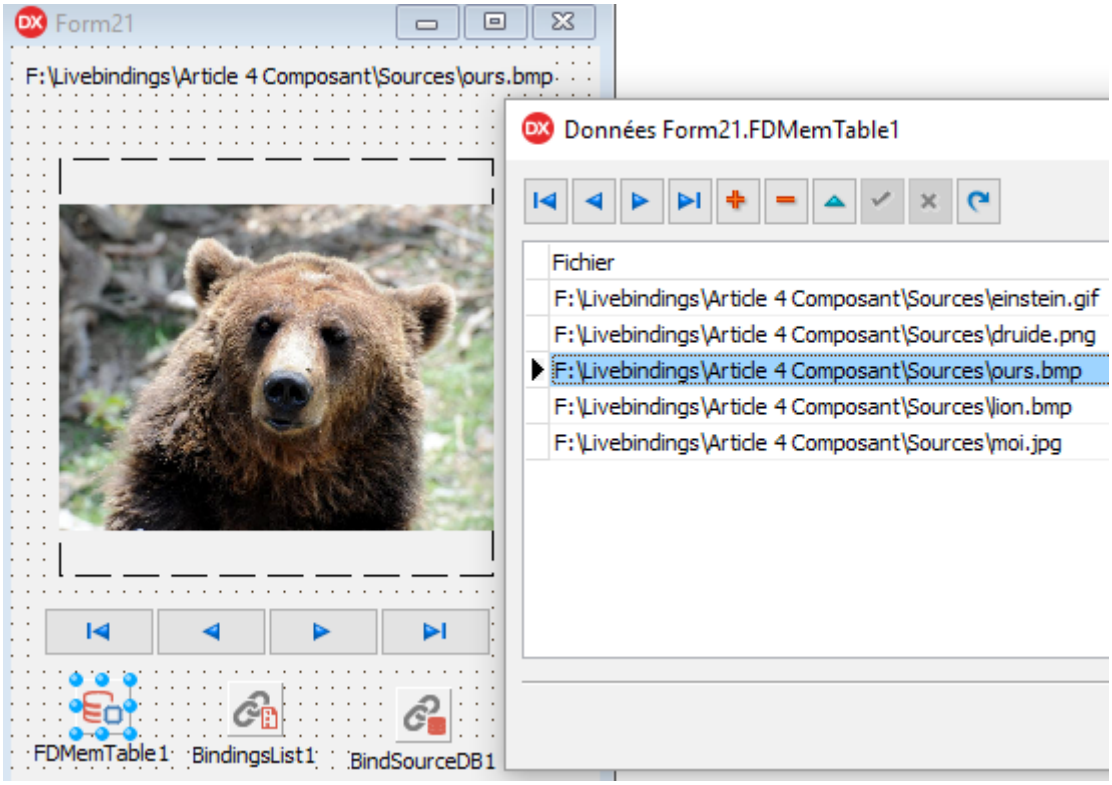
```
22.   published
23.     { Déclarations publiées }
24.     property NomFichierImage : TFileName read FNomFichierImage write DrawImage;
25.     property UseEvalShortcuts: Boolean read FUseEvalShortcuts;
26.   end;
```



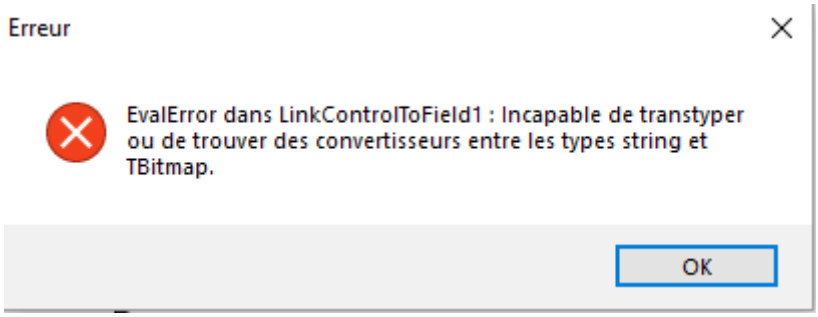

## III-C-1 - Test en situation (liaison avec une table)

Le test précédent n'impliquait pas de données *stricto sensu*. Il est par conséquent temps de faire un test un peu plus poussé. Pour cela, je vais écrire un nouveau programme qui va utiliser une table en mémoire que je remplirai grâce à un petit jeu d'essai.




*Programme test, design*

Premier écueil, dès la tentative de mise en place du lien j'obtiens l'erreur suivante :




*Erreur de conversion*

Pourtant ce composant semblait fonctionner !

Que s'est-il passé ? Le type de lien est en cause, ce n'est pas un lien de type **TLinkPropertyToField** mais un **TLinkControlToField** qui a été créé par le concepteur visuel.

*Je vous rappelle que les **LiveBindings** utilisent les **RTTI**. Malheureusement, aucun mécanisme ne permet d'indiquer qu'une chaîne de caractères fait référence à un fichier image !*

Pour résoudre ce problème, il me faut ajouter ce mécanisme, à savoir un convertisseur.

*Je n'avais pas encore parlé de cette nouveauté dans ma série sur les LiveBindings !*

*Plus d'informations dans la documentation **Wiki Embarcadero**.*

Où placer ce code ? Comme nous avons déjà eu affaire à ce genre de fonction **registerxxxx**, il semble évident de placer cette fonction dans une section **initialization** tout comme il ne faudra pas oublier de dé-recenser celle-ci dans la partie **finalization**. Par contre, ce n'est pas dans le fichier de recensement du composant (qui ne fait qu'ajouter notre composant à l'EDI Delphi), mais dans l'unité de notre classe que cette partie doit être incluse.

Convertisseur

```
Type
// …
const Convertisseur : String = 'NomFichierVersTPicture';
      CetteUnite : String = 'ImageFileVCL';
      Description : String = 'Récupère l''image';

procedure RegisterConverter;
procedure UnRegisterConverter;


implementation

//…


procedure RegisterConverter;
begin
if not TValueRefConverterFactory.HasConverter(convertisseur)
 then begin
  TValueRefConverterFactory.RegisterConversion(TypeInfo(String),TypeInfo(TPicture),
    TConverterDescription.Create(
      procedure(const InValue: TValue; var OutValue: TValue)
      var LOutObject : TObject;
```

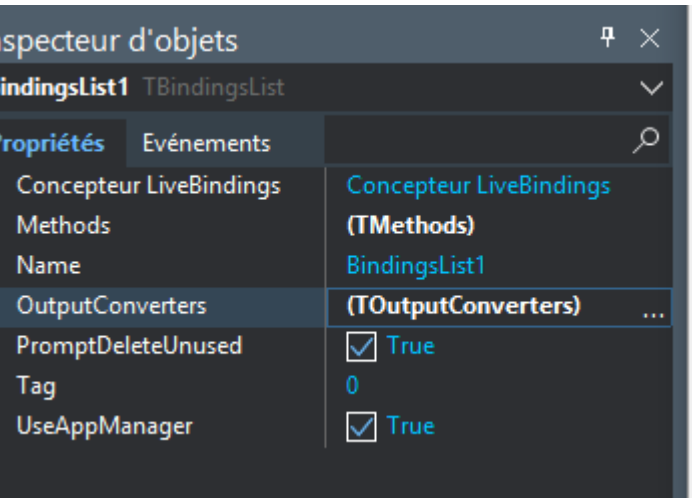
Convertisseur

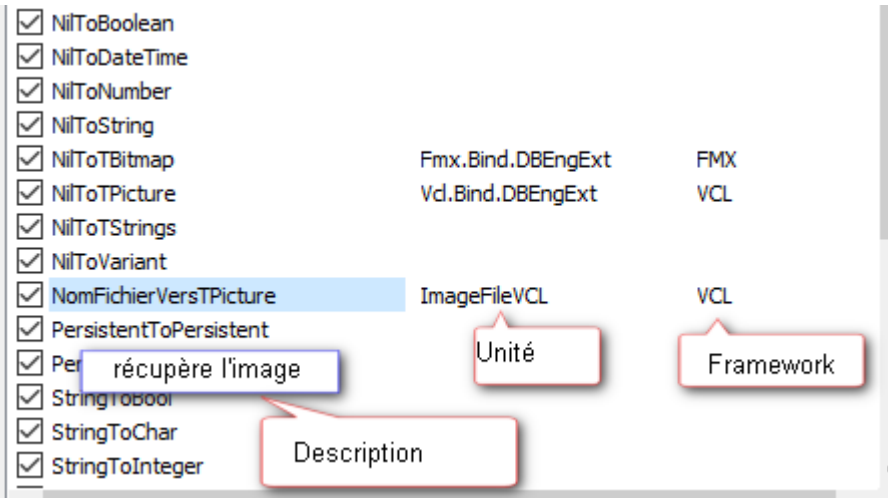
```
      begin
          LOutObject := OutValue.AsObject;
          Assert(LOutObject <> nil);
          Assert(LOutObject is TPicture);
            try
              TPicture(LOutObject).LoadFromFile(InValue.ToString)
            except
              TPicture(LOutObject).Graphic := nil;
            end;
      end,
      Convertisseur,  // id du convertisseur 'NomFichierVersTPicture'
      Convertisseur,  // nom du convertisseur 'NomFichierVersTPicture'
      CetteUnite,     // nom de l'unité 'ImageFileVCL'
      True,
      Description,   // description 'Récupère l'image'
      VCL.Controls.TControl // VCL
      )
    );
 end;
end;

procedure UnRegisterconverter;
begin
 TValueRefConverterFactory.UnRegisterConversion(Convertisseur);
end;

initialization
  registerconverter;
finalization
  unregisterconverter;
```

Question de style, plutôt que de coder directement le convertisseur au sein de la partie initialisation, j'ai préféré créer deux procédures (**RegisterConverter, UnRegisterConverter**).

J'ai également ajouter plusieurs constantes alors que j'aurais pu inscrire directement ces valeurs. En fait j'ai préféré rester au plus près de ce qui se trouve dans les sources Delphi.

Quel impact ? Le premier, est bien évidemment que le composant fonctionne, du moins pour les fichiers de type BMP, un petit second se trouve dans l'IDE lorsque l'on regarde la liste des convertisseurs proposés par le **TBindingsList**.

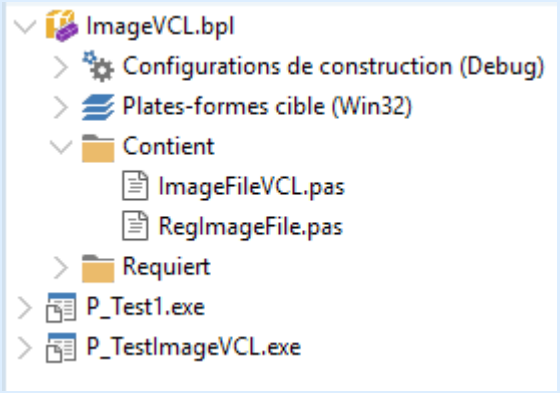



*Cette image montre l'utilité de fournir les informations, même les optionnelles que sont les paramètres **ADisplayName, AUnitName, ADescription et AFrameworkClass** du constructeur.*

Étrangement, seuls les fichiers de type BMP sont affichés ! Pour afficher tout type de fichier image (JPEG, PNG ou GIF), il faudra ajouter **Vcl.Imaging.JPEG, VCL.Imaging.PngImage, VCL.Imaging.GIF** à la liste des unités à utiliser.

*Les sources de cette première partie se trouvent dans l'archive téléchargeable à **cette adresse**.*



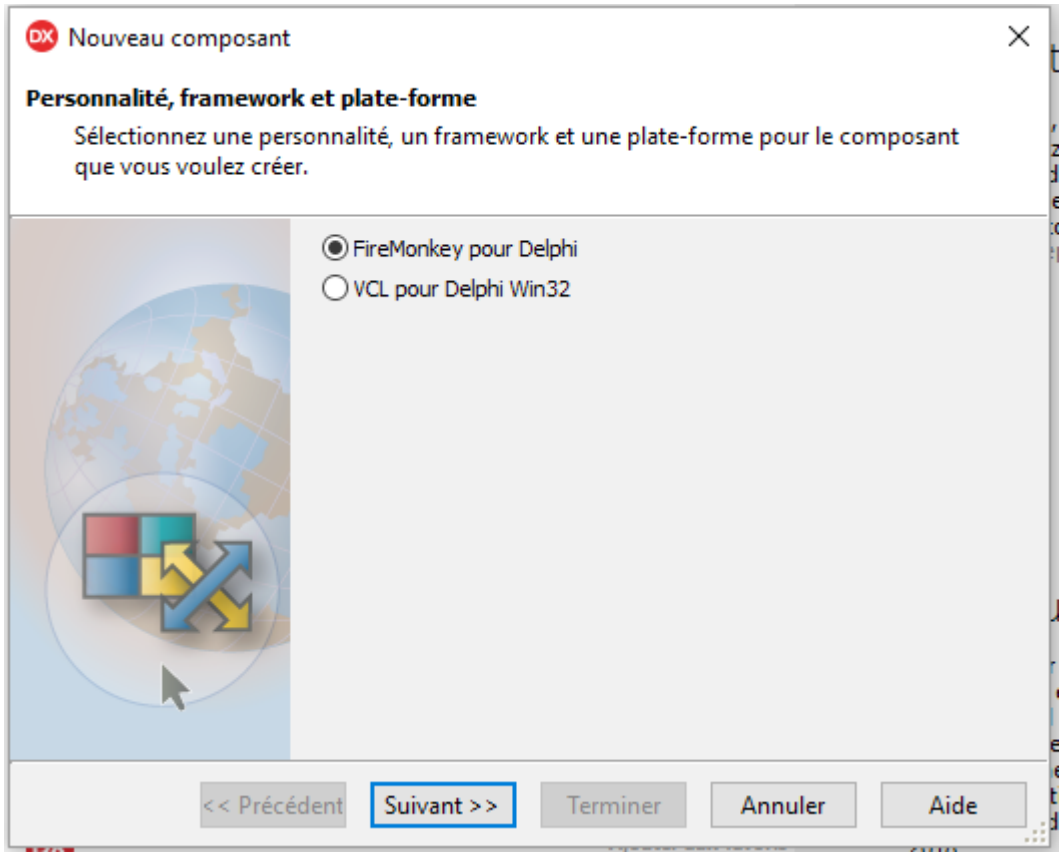

*Attention, il vous faudra certainement indiquer ou modifier le chemin du composant dans les options du projet.*

## III-D - Rédaction du même composant pour FireMonkey

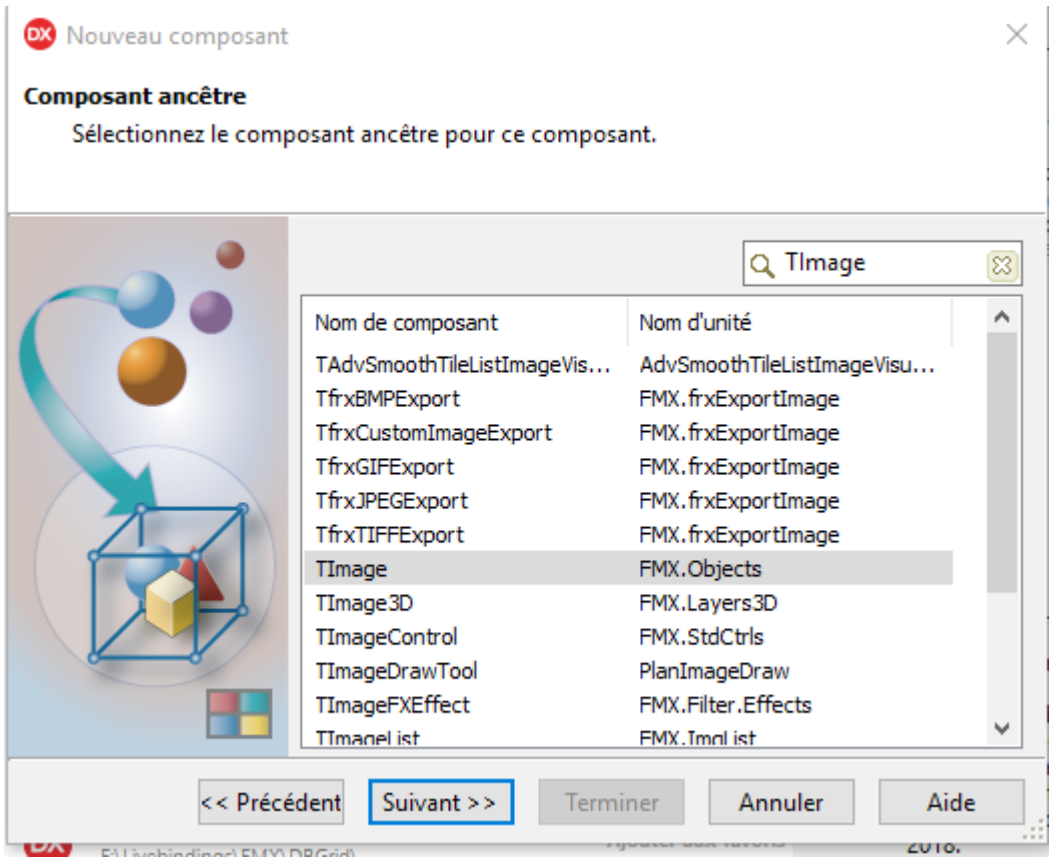
Ce qu'il y a d'intéressant avec l'exemple choisi, c'est que le composant n'existant ni en VCL ni en FMX, nous allons pouvoir aborder deux thèmes nouveaux : les différences entre les deux « cadres logiciels » (**frameworks**) et les techniques pour mixer les deux en un seul package.

### III-D-1 - Écriture du composant pour FiremonKey

Le point de départ est le même que pour le composant en VCL, mais cette fois-ci j'indique l'utilisation du framework FMX.




Je sélectionne ensuite le composant ancêtre **TImage**, comme pour la VCL, mais, cette fois, il est contenu dans l'unité FMX.Objects.



Généré par l'expert

```
unit FMX.FileNameImage;

interface

uses
  System.SysUtils, System.Classes, FMX.Types, FMX.Controls, FMX.Objects;

type
  TFileNameImage = class(TImage)
  private
    { Déclarations privées }
  protected
    { Déclarations protégées }
  public
    { Déclarations publiques }
  published
    { Déclarations publiées }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Tutoriels', [TFileNameImage]);
end;

end.
```

À quelles différences suis-je confronté ? La première, assez évidente, est dans la liste des **uses**. Elle est logique puisqu'il s'agit du framework FMX.

En revanche, tout ce qui concerne les **LiveBindings**, c'est-à-dire les procédures ou fonctions (**ObserverToggle, ObserverAdded, CanObserve),** sera strictement identique à ce qui a été déjà écrit pour le composant VCL.

Après m'être occupé du processus de recensement et avoir déclaré les différentes procédures et fonctions (simples copier/coller à partir de mon fichier source VCL), j'obtiens :

```
unit FMX.FileNameImage;

interface

uses
  System.SysUtils, System.Classes, FMX.Types, FMX.Controls, FMX.Objects;

type
  [ComponentPlatforms(pidWin32 or pidWin64 or pidAndroid)]
  [ObservableMember('NomFichierImage')]
  TFileNameImage = class(TImage)
  private
    { Déclarations privées }
     FNomFichierImage : TFileName;
     procedure ObserverToggle(const AObserver: IObserver; const Value: Boolean);
  protected
    function CanObserve(const ID: Integer): Boolean; override;
    procedure ObserverAdded(const ID: Integer; const Observer: IObserver); override;
    procedure DrawImage(Value : TFileName);
  published
    property FichierImage : TFileName read FNomFichierImage write DrawImage;
  end;


procedure RegisterConverter ;
procedure UnRegisterConverter ;

implementation
```

```
{ TFileNameImage }

function TFileNameImage.CanObserve(const ID: Integer): Boolean;
begin
 case ID of
    TObserverMapping.EditLinkID,
    TObserverMapping.ControlValueID:
      Result := True;
  else
    Result := False;
  end;
end;

procedure TFileNameImage.DrawImage(Value: TFileName);
begin
//
end;

procedure TFileNameImage.ObserverAdded(const ID: Integer;
  const Observer: IObserver);
begin
  inherited;
  if ID = TObserverMapping.EditLinkID then
    Observer.OnObserverToggle := ObserverToggle;
end;

procedure TFileNameImage.ObserverToggle(const AObserver: IObserver;
  const Value: Boolean);
var
  LEditLinkObserver: IEditLinkObserver;
begin
  if Value then
  begin
    if Supports(AObserver, IEditLinkObserver, LEditLinkObserver) then
      Enabled := not LEditLinkObserver.IsReadOnly;
  end else
    Enabled := True;
end;

procedure RegisterConverter;
begin
if not TValueRefConverterFactory.HasConverter('NomFichierImageToStringFMX')
 then begin
  TValueRefConverterFactory.RegisterConversion(TypeInfo(String),
    TypeInfo(TBitmap),
    TConverterDescription.Create(
      procedure(const InValue: TValue; var OutValue: TValue)
      var LOutObject : TObject;
      begin
          LOutObject := OutValue.AsObject;
          Assert(LOutObject <> nil);
          Assert(LOutObject is TBitmap);
            try
              TBitmap(LOutObject).LoadFromFile(InValue.ToString)
            except
              TBitmap(LOutObject).Clear(0);
             end;
      end,
      'NomFichierImageToStringFMX',
      'NomFichierImageToStringFMX',
      EmptyStr,
      True,
      EmptyStr,
      nil      // pas d'indication de framework
      )
    );
 end;
end;

procedure UnRegisterconverter;
begin
```

```
 TValueRefConverterFactory.UnRegisterConversion('NomFichierImageToStringFMX');
end;

initialization
  registerconverter;
finalization
  unregisterconverter;
end.
```

Reste à prendre en compte le changement de valeur de la propriété et charger l'image, ce qui est du ressort de la procédure **DrawImage**.

```
procedure TFileNameImage.DrawImage(Value: TFileName);
begin
if FNomFichierImage<>Value then
 begin
   FNomFichierImage:=Value;
  if FileExists(Value)
    then Bitmap.LoadFromFile(Value)
    else Bitmap:=nil;
  TLinkObservers.ControlChanged(Self);
 end;
end;
```

Une seule différence apparaît : ce n'est plus la propriété **Picture** comme en VCL qui est utilisée, mais **Bitmap**.

Je pourrais m'arrêter là, mais ce serait ignorer les spécificités de chaque cible :

- les noms de fichiers Windows sont insensibles à la casse ;
- le stockage des images Android doit se faire dans un emplacement spécifique.

C'est à ce moment que commencent à entrer en scène les directives de compilation et plus particulièrement **les conditions prédéfinies**.

Pour ce qui est de la sensibilité à la casse, je peux soit introduire une variable booléenne, soit utiliser plus directement les directives de compilation, la première méthode étant peut-être plus lisible que la seconde.

Comparez :

Première méthode
```
procedure TFileNameImage.DrawImage(Value: TFileName);
var Changed: Boolean;
begin
{$IFDEF MSWINDOWS}
Changed:=not SameText(FNomFichierImage,Value);
{$ELSE}
Changed:= FNomFichierImage<>Value;
{$ENDIF}
if Changed then
...
```

Seconde méthode
```
procedure TFileNameImage.DrawImage(Value: TFileName);
begin
if
{$IFDEF MSWINDOWS}
 not SameText(FNomFichierImage,Value)
{$ELSE}
  FNomFichierImage<>Value
{$ENDIF}
then begin
...
```

Pour ce qui est des emplacements de fichiers, ils vont beaucoup dépendre des conventions que vous voudrez mettre en place. Pour Android, les images seront certainement stockées dans un répertoire spécifique **/data/data/<ID application>/files** que l'on obtient facilement grâce à **System.IOUtils.TPath.GetDocumentsPath**. Pour Windows, les choix sont plus variés avec l'utilisation au choix du chemin complet dans la propriété et un lieu de stockage particulier !

J'ai choisi un compromis : **TPath.GetDocumentsPath** pour Android alors que pour Windows il s'agira d'un chemin complet.

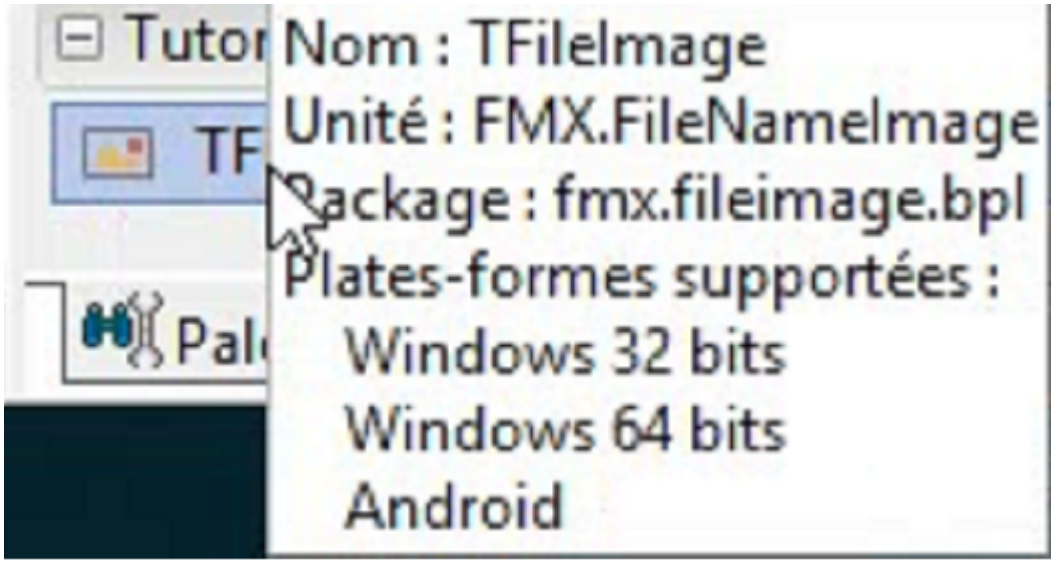
Pour prendre en compte ces conventions, je déclare la nouvelle variable locale **FullPathName**.

Code final

```
procedure TFileNameImage.DrawImage(Value: TFileName);
var Changed : Boolean;
    FullPathName : TFileName;
begin
{$IFDEF MSWINDOWS}
Changed:=Not SameString(FNomFichierImage,Value);
{$ELSE}
Changed:=FNomFichierImage<>Value;
{$ENDIF}
if FNomFichierImage<>Value then
 begin
  {$IFDEF MSWINDOWS}
    FullPathName:=Value;
  {$ELSE}
 FullPathName:=System.IOUtils.TPath.Combine(System.IOUtils.TPath.GetSharedPicturesPath,ExtractFilePath(Value));
  {$ENDIF}
  FNomFichierImage:=Value;
  if FileExists(FullPathName)
    then Bitmap.LoadFromFile(FullPathName)
    else Bitmap:=nil;
  TLinkObservers.ControlChanged(Self);
 end;
```

*Vous remarquerez que je ne tiens pas compte d'OSX ni d'iOS. C'est plus ou moins voulu : d'une part je n'ai aucun matériel pour tester ces deux environnements, d'autre part cela va me permettre d'introduire la notion de cibles de mon package FMX et ainsi de restreindre mon composant aux deux environnements testés.*

Pour limiter mon composant aux seules plates-formes prévues, j'ajoute l'instruction tout simplement `[ComponentPlatforms(pidWin32 or pidWin64 or pidAndroid)]` devant ma déclaration de classe. Passer la souris sur le composant dans la palette m'indiquera alors les plates-formes supportées (cf. **documentation**).




## III-D-2 - Recensement

Un dernier coup de collier est nécessaire avant de pouvoir utiliser notre nouveau composant : il faut écrire l'unité qui servira à recenser le composant, avec les quelques lignes que j'ai supprimées du source proposé par l'expert, en prévision du chapitre suivant **III.D.5**.

Recensement du composant

```
unit FMX.FileNameImageReg;

interface

procedure register;

implementation

uses System.Classes, FMX.FileNameImage, Data.Bind.Components;

procedure register;
begin
  RegisterComponents('Tutoriels', [TFileImage]);
end;

initialization
Data.Bind.Components.RegisterObservableMember(TArray<TClass>.Create(TFileImage), 'NomFichierImage', 'FMX');

finalization
Data.Bind.Components.UnregisterObservableMember(TArray<TClass>.Create(TFileImage));
end.
```

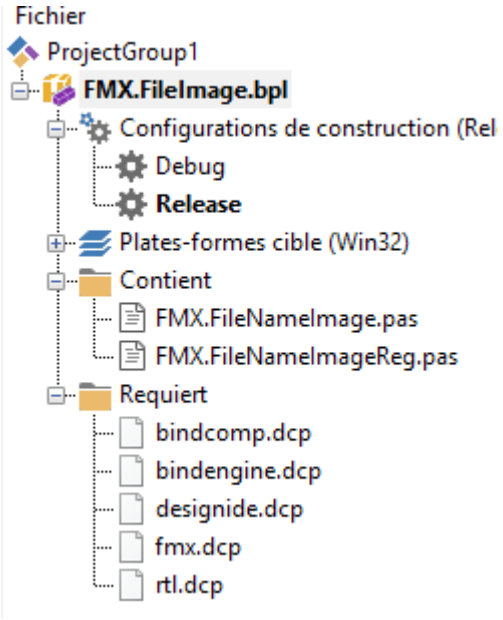
La seule différence notable avec le même composant VCL créé précédemment est la constante « FMX » au lieu de « DFM » en ce qui concerne la liaison.

## III-D-3 - Création du package

Ayant demandé de créer un source lors du déroulement de l'expert, mes sources ne sont, pour l'instant, pas installables en l'état : il me faut à présent créer un package.

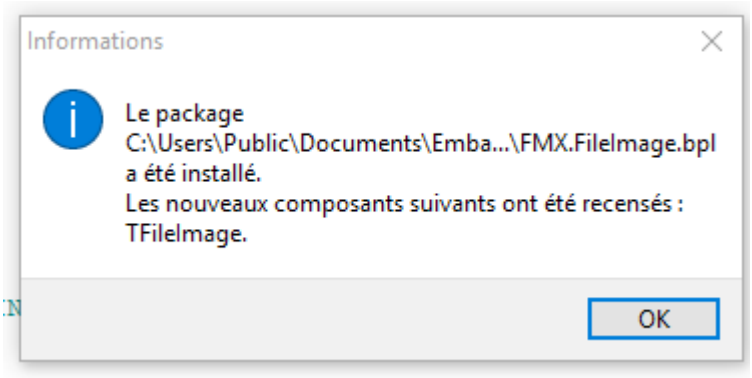
*Il faut choisir Fichier/Nouveau/Autres/Package depuis le menu principal pour cette création.*

J'y ajoute mes deux sources et les unités requises.



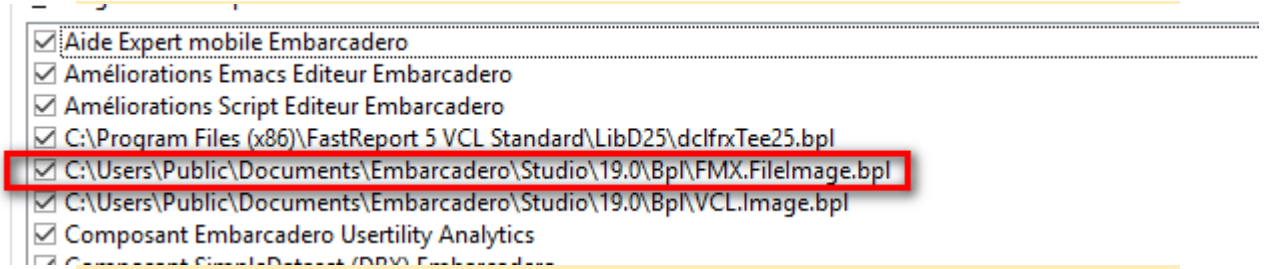

*Vous ne savez pas vraiment quelles unités sont requises ? Pas de problème ! Faites une première compilation, qui se révélera pleine d'erreurs d'inattention, mais qui ajoutera les unités requises.*

Une fois tout réglé, il suffit d'utiliser l'option d'installation du paquet.




*À force de faire des essais, il est possible que l'installation échoue.*

*Si vous êtes confronté à ce problème, désinstallez le package en passant par les options de menu Outils/Composants/Installer des packages.*

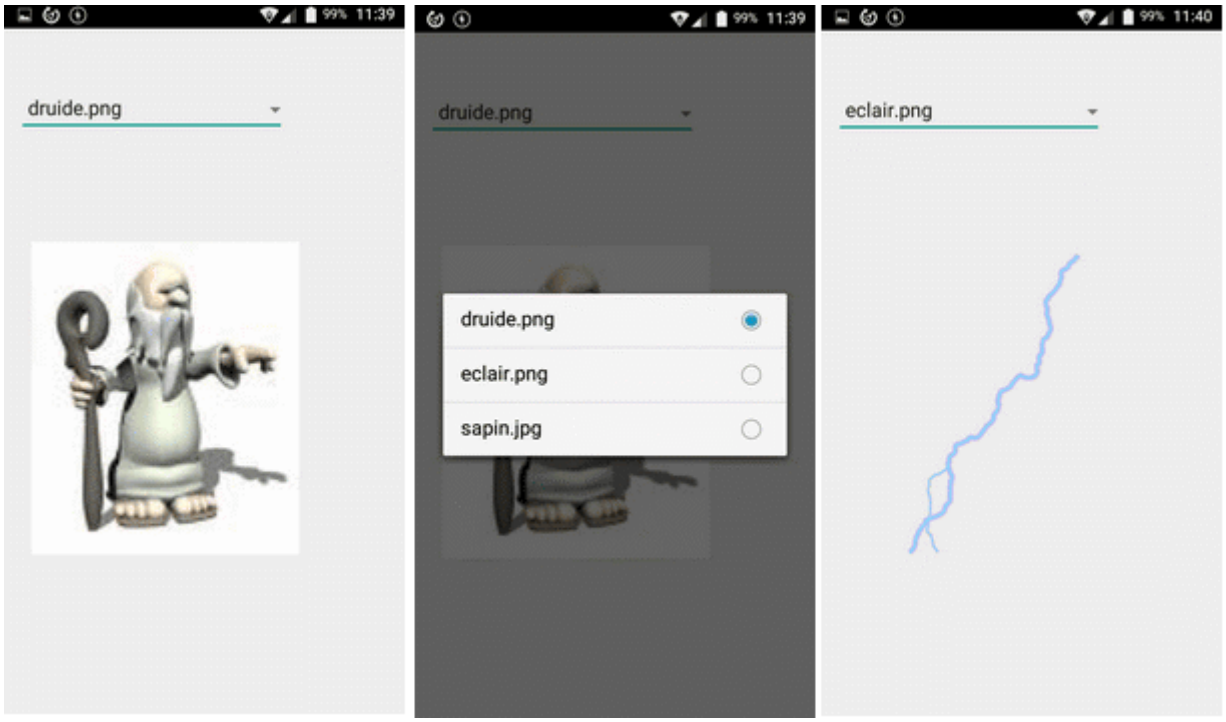



*Package déjà installé*

*Refaites alors une tentative.*

## III-D-4 - Utilisation

Je laisse libre cours à votre imagination pour l'utilisation du composant. Voici cependant ci-dessous quelques captures réalisées sur mon téléphone Android :




## III-D-5 - Deux cadres, un seul code source ?

La question se pose de savoir s'il est possible d'écrire un seul code pour les deux frameworks.

Est-ce possible ? Oui, et l'objet de ce chapitre est d'expliquer comment.

Est-ce judicieux ? La réponse est moins facile à donner. Comme vous le constaterez, il y aura quand même deux packages (**.bpl**), même si l'un est réduit à sa plus simple expression, et le code source du composant devra être truffé de directives de compilation voire, dans le cas de composants plus complexes, d'un nombre de déclarations et fonctions supplémentaires (pour les anglophones, je suggère de suivre **cette vidéo** afin de vous faire une idée du problème).

## III-D-5-a - VCL vs FireMonkey

Je ne vais pas entrer dans les détails, mais plutôt proposer une approche générale en quelques tableaux (**extraits de cette présentation CodeRage VI en 2011**).

Déjà, il y a beaucoup de points communs entre les contrôles de la VCL et ceux de FireMonkey.

| **Points communs** |
|---|
| Les deux héritent de **TComponent**. |
| Les deux ont une méthode **Paint**. |
| Les deux cadres ont des **TCanvas**. |
| Les deux cadres ont des **TBitmap.** |
| Les deux ont des méthodes similaires :<br><br>• **MouseDown**, **MouseUp** ;<br>• **SetBounds**. |
| Les deux ont des propriétés identiques ou très proches :<br><br>• **Color**,<br>• **Visible**,<br>• **TabOrder**… |

Mais les deux comportent aussi des différences qu'il faudra bien prendre en compte.

| **Différences** | VCL | FMX |
|---|---|---|
| Différentes unités | **VCL.Controls**, **VCL.Graphics** | **FMX.Types** pour les couleurs (**TAlphaColors**) **System.UITypes** |
| | **TCustomControl** | **TControl** |
| Types de coordonnées | **Integer** | **Single** |
| Forcer le dessin | | **Invalidate** inexistant |
| Focalisation | **Focused** | **IsFocused** |
| | Message **WM_VISIBLECHANGED** | Fonction **SetVisible** |
| | Fonction **MouseWheel** | Fonctions **DoMouseWheelUp** **DoMouseWheelDown** |
| **TCanvas** est très différent. | | |
| **TBitmap** est différent. | | |

| **Différences de TCanvas** | VCL | vs | FMX |
|---|---|---|---|
| Les coordonnées | **Integer** | devient | **Single** |
| Les couleurs | **TColor** | devient | **TAlphaColor** |
| La brosse | **Brush** | devient | **Fill** |
| Le crayon | **Pen** | devient | **Stroke** |
| Pour le dessin | **TBrush, TPen** | Sont fusionnés en | **TBrush** |

## III-D-5-b - Modifications à apporter

Si nous comparons les deux sources **VCL.FileImage.pas** et **FMX.FileImage.pas**, nous remarquons qu'ils comportent de nombreuses similitudes.

*Les différences se situent au niveau des unités à inclure et dans la procédure **DrawImage** où la propriété qui va charger l'image est **Picture** pour une composant VCL et **Image** pour FMX.*

Il est donc envisageable de se poser la question : pourquoi ne pas les fusionner et n'en faire qu'un seul ? Avantage ciblé ? Même code implique modifications répercutées.

Comment réussir à mixer les deux frameworks ? La réponse réside, une fois de plus, dans les directives de compilation, mais cette fois il va s'agir d'introduire une définition personnelle.

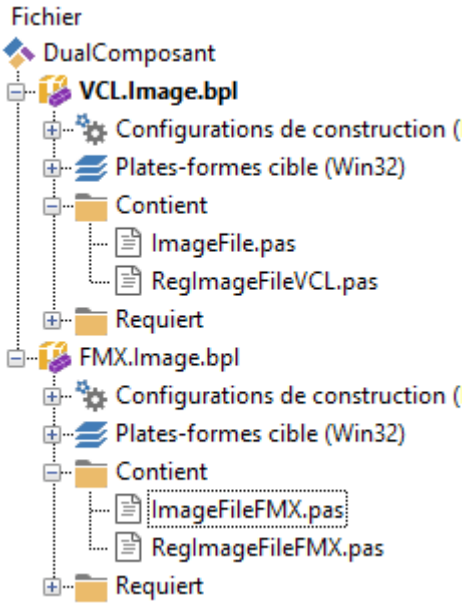
Pour fournir un exemple sans écraser les composants déjà écrits, je vais tout d'abord copier et renommer le fichier **FileImageVCL.pas** en **FileImage.pas**, mais aussi changer le nom de la classe.

FileImage

```
unit ImageFile;

interface

uses
  System.SysUtils, System.Classes, System.IOUtils,
  Vcl.Controls, Vcl.ExtCtrls,
  Data.Bind.Components;

type
  [ObservableMember('NomFichierImage')]
  TFileImage = class(TImage)
  private
    { Déclarations privées }
    FNomFichierImage : TFileName;
    procedure ObserverToggle(const AObserver: IObserver; const Value: Boolean);
  protected
    { Déclarations protégées }
    procedure DrawImage(Value : TFileName);
    function CanObserve(const ID: Integer): Boolean; override;
    procedure ObserverAdded(const ID: Integer; const Observer: IObserver); override;
  public
    { Déclarations publiques }
  published
    { Déclarations publiées }
    property NomFichierImage : TFileName read FNomFichierImage write DrawImage;
  end;


implementation


{ TFileImage }

function TFileImage.CanObserve(const ID: Integer): Boolean;
begin
  case ID of
    TObserverMapping.EditLinkID,
    TObserverMapping.ControlValueID:
      Result := True;
  else
    Result := False;
  end;
end;

procedure TFileImage.DrawImage(Value: TFileName);
```

FileImage
```
var FullPathName : TFileName;
begin
if not SameText(FNomFichierImage,Value) then
  begin
  FNomFichierImage:=Value;
  if FileExists(FullPathName)
    then Picture.LoadFromFile(Value)
    else Picture:=nil;
  TLinkObservers.ControlChanged(Self);
 end;
end;

procedure TFileImage.ObserverAdded(const ID: Integer;
  const Observer: IObserver);
begin
  if ID = TObserverMapping.EditLinkID then
    Observer.OnObserverToggle := ObserverToggle;
end;

procedure TFileImage.ObserverToggle(const AObserver: IObserver;
  const Value: Boolean);
var
  LEditLinkObserver: IEditLinkObserver;
begin
  if Value then
  begin
    if Supports(AObserver, IEditLinkObserver, LEditLinkObserver) then
      Enabled := not LEditLinkObserver.IsReadOnly;
  end else
    Enabled := True;
  end;

end.
```

Mon groupe de projet va être constitué ainsi :




Le fichier **FileImageFMX.pas** se réduira à quelques lignes :

FileImageFMX
```
unit ImageFileFMX;
{$DEFINE FMX}
{$I ImageFile.pas}
```

Je vais ensuite modifier le fichier **FileImage.pas** de façon à prendre en compte ma définition.

*Première ligne, il faut effacer le nom de l'unité s'il s'agit du composant FMX.*

*Dans les **uses**, il s'agit de préciser les unités à inclure.*

*Les dernières modifications concernent la procédure **DrawImage**.*

```
{$IFNDEF FMX} unit ImageFile; {$ENDIF}

interface

uses
  System.SysUtils, System.Classes, System.IOUtils,
  {$IFDEF FMX}
    FMX.Types, FMX.Controls, FMX.Objects,
  {$ELSE}
    Vcl.Controls, Vcl.ExtCtrls,
  {$ENDIF}
  Data.Bind.Components;

type
  [ObservableMember('NomFichierImage')]
  TFileImage = class(TImage)
  private
    { Déclarations privées }
    FNomFichierImage : TFileName;
    procedure ObserverToggle(const AObserver: IObserver; const Value: Boolean);
  protected
    { Déclarations protégées }
    procedure DrawImage(Value : TFileName);
    function CanObserve(const ID: Integer): Boolean; override;
    procedure ObserverAdded(const ID: Integer; const Observer: IObserver); override;
  public
    { Déclarations publiques }
  published
    { Déclarations publiées }
    property NomFichierImage : TFileName read FNomFichierImage write DrawImage;
  end;


implementation


{ TFileImage }

function TFileImage.CanObserve(const ID: Integer): Boolean;
begin
  case ID of
    TObserverMapping.EditLinkID,
    TObserverMapping.ControlValueID:
      Result := True;
  else
    Result := False;
  end;
end;

procedure TFileImage.DrawImage(Value: TFileName);
var FullPathName : TFileName;
begin
{$IFDEF FMX}
if FNomFichierImage<>Value then
 begin
  {$IFDEF ANDROID}
    FullPathName:=System.IOUtils.TPath.Combine(System.IOUtils.TPath.GetDocumentsPath,Value);
  {$ELSE}
    FullPathName:=Value;
```

```
  {$ENDIF}
{$ELSE}
if not SameText(FNomFichierImage,Value) then
  begin
{$ENDIF}
  FNomFichierImage:=Value;
  if FileExists(FullPathName)
    then {$IFDEF FMX}Bitmap.LoadFromFile(FullPathName)
         {$ELSE}     Picture.LoadFromFile(Value) {$ENDIF}
    else {$IFDEF FMX}Bitmap:=nil;
         {$ELSE}     Picture:=nil; {$ENDIF}
  TLinkObservers.ControlChanged(Self);
 end;
end;

procedure TFileImage.ObserverAdded(const ID: Integer;
  const Observer: IObserver);
begin
  if ID = TObserverMapping.EditLinkID then
    Observer.OnObserverToggle := ObserverToggle;
end;

procedure TFileImage.ObserverToggle(const AObserver: IObserver;
  const Value: Boolean);
var
  LEditLinkObserver: IEditLinkObserver;
begin
  if Value then
  begin
    if Supports(AObserver, IEditLinkObserver, LEditLinkObserver) then
      Enabled := not LEditLinkObserver.IsReadOnly;
  end else
    Enabled := True;
  end;
```

La partie convertisseur reste peut-être celle qui est la plus délicate mais pas plus que la procedure **DrawImage.** Il suffit de faire attention aux directives de compilation à indiquer.

```
const Convertisseur : String {$IFDEF FMX}= 'NomFichierImageToStringFMX';
                             {$ELSE}= 'NomFichierImageToString';
                             {$ENDIF}
      CetteUnite : String = 'ImageFile';
      Description : String = 'Affiche une image à partir d''un nom de fichier';

procedure RegisterConverter;
procedure UnRegisterConverter;


procedure RegisterConverter;
begin
if not TValueRefConverterFactory.HasConverter(convertisseur)
 then begin
  TValueRefConverterFactory.RegisterConversion(TypeInfo(String),
    {$IFDEF FMX} TypeInfo(TBitmap){$ELSE} TypeInfo(TPicture){$ENDIF},
    TConverterDescription.Create(
      procedure(const InValue: TValue; var OutValue: TValue)
      var LOutObject : TObject;
      begin
          LOutObject := OutValue.AsObject;
          Assert(LOutObject <> nil);
          {$IFDEF FMX}
            Assert(LOutObject is TBitmap);
            try
              TBitmap(LOutObject).LoadFromFile(InValue.ToString)
            except
              TBitmap(LOutObject).Clear(0);
             end;
          {$ELSE}
            Assert(LOutObject is TPicture);
            try
              TPicture(LOutObject).LoadFromFile(InValue.ToString)
```

```
            except
              TPicture(LOutObject).Graphic := nil;
            end;
          {$ENDIF}
      end,
      Convertisseur,
      Convertisseur,
      CetteUnite,
      True,
      Description,
      nil      // pas d'indication de framework
      )
    );
 end;
end;
```

*Une autre solution aurait été d'écrire deux procédures d'enregistrement différentes.*

**Alternative**

```
const ConvertisseurFMX : String = 'NomFichierImageToStringFMX';
      ConvertisseurVCL : String= 'NomFichierImageToString';
      CetteUnite : String = 'ImageFile';
      Description : String = 'Affiche une image à partir d''un nom de fichier';

procedure RegisterConverterVCL;
procedure RegisterConverterFMX ;
procedure UnRegisterConverter;
```

Reste ensuite à enregistrer les composants. Comme l'indique la structure de mon groupe de projet, j'ai préféré créer deux unités distinctes.

Pour la VCL, j'obtiens :

**RegImageFileVCL**

```
1. unit RegImageFileVCL;
2.
3. interface
4. uses System.Classes, VCL.Controls,
5.      Data.Bind.Components,
6.      ImageFile;
7.
8. procedure Register;
9.
10. implementation
11.
12. procedure Register;
13. begin
14.   GroupDescendentsWith(TFileImage, Vcl.Controls.TControl);
15.   RegisterComponents('Tutoriels', [TFileImage]);
16. end;
17.
18. initialization
19.
 Data.Bind.Components.RegisterObservableMember(TArray<TClass>.Create(TFileImage), 'NomFichierImage', 'DFM');
20.
21. finalization
22.  Data.Bind.Components.UnregisterObservableMember(TArray<TClass>.Create(TFileImage));
23. end.
```

Pour FMX, je propose :

**RegImageFileFMX**

```
1. unit RegImageFileFMX;
2.
3. interface
```

RegImageFileFMX
```
4.
5. procedure Register;
6.
7. implementation
8.
9. uses System.Classes, Data.Bind.Components,
10.      FMX.Controls,
11.      ImageFileFMX;
12.
13. procedure Register;
14. begin
15.   GroupDescendentsWith(TFileImage, Fmx.Controls.TControl);
16.   RegisterComponents('Tutoriels', [TFileImage]);
17. end;
18.
19. initialization
20. Data.Bind.Components.RegisterObservableMember(TArray<TClass>.Create(TFileImage), 'NomFichierImage', 'FMX');
21.
22. finalization
23.  Data.Bind.Components.UnregisterObservableMember(TArray<TClass>.Create(TFileImage));
24. end.
```

*Remarquez bien la première ligne de la procédure **Register** : la fonction **GroupDescendentsWith** est indispensable pour distinguer les frameworks !*

Une fois les deux packages installés, l'objectif est atteint !

Vous retrouverez les sources ainsi que les programme tests de ce chapitre en téléchargement **ici**.

## IV - Conclusion

À l'issue de ce tutoriel, nous savons comment créer un composant qui peut être lié par l'intermédiaire des **LiveBindings**. Cette première partie nous a permis de nous familiariser avec les méthodes supplémentaires (**ObserverToggle, ObserverAdded, CanObserve**) nécessaires au fonctionnement du concepteur visuel de liaisons de l'EDI.

En guise d'exercice, je peux vous proposer une amélioration possible pour le composant FMX. Le composant **TImage** du framework FMX offre la possibilité de charger différentes images dans sa propriété MultiresBitmap. Il doit donc être possible d'utiliser cette collection d'images pour afficher une image par défaut. En guise de piste de travail, je vous propose le code suivant :

Image par défaut
```
uses FMX.MultiresBitmap, FMX.PlatForm;

procedure TForm22.Button1Click(Sender: TObject);
var
  ScreenSvc: IFMXScreenService;
  SCale: Single;
begin
  if TPlatformServices.Current.SupportsPlatformService(IFMXScreenService,
    IInterface(ScreenSvc)) then
  begin
   Scale:=ScreenSvc.GetScreenScale;
  end;
  try
    Image1.Bitmap := Image1.MultiresBitmap.ItemByScale(Scale, false, true).Bitmap;
  finally
  end;
end;
```

Le but de l'exercice est d'intégrer ce code au composant. À vous de jouer !

Je remercie les bêta-testeurs de mon composant, et bien sûr l'équipe des correcteurs techniques et orthographiques de cet article (ma maison d'édition en quelque sorte).

Une autre amélioration possible serait d'ajouter la possibilité de charger l'image via FTP.

À vos claviers !

Je remercie les bêta-testeurs de mon composant **pprem**, **nabil74**, **nicolas.bbs**

et, bien-sûr, l'équipe des correcteurs techniques **gvasseur58** et orthographiques ??? de cet article (ma maison d'édition en quelque sorte).

1 : Pour le faire, il faut créer un package ou ajouter nos sources à un package déjà existant.