

LiveBindings de A à ... Écrire un composant observable II

Par [Serge Girard](#) 

Date de publication : 17 mars 2022

TOUT PUBLIC

Lors du tutoriel précédent (**LiveBindings de A à ... Écrire un composant observable**) je m'étais arrêté à une liaison unidirectionnelle ou, du moins, était-ce mon objectif.

Le fait de tester un composant qui soit aussi bien VCL que FMX rendait le truc un peu plus ardu.

Cette fois je reprend l'ouvrage avec un objectif plus simple : une simple Diode qui sera déclinée par la suite, mais surtout uniquement pour la plateforme FMX.

I - Objectif : créer un composant TDiode pour FMX.....	3
II - Création du paquet.....	4
II-A - Preamble.....	8
II-B - Recensement des propriétés à proposer.....	9
II-C - Codage.....	11
II-D - Ajout de la partie LiveBindings.....	12
III - Installation et Tests.....	13
III-A - Installer le composant.....	13
III-B - Programme de test.....	15
III-C - Corrections nécessaires sur les liaisons.....	17
III-C-1 - Première solution.....	17
III-C-2 - Solution.....	18
IV - Extension, rendre la propriété BrightPos accessible.....	18
IV-A - Les valeurs X et Y de la propriété BrightPos.....	19
IV-B - Test final.....	20
V - Un composant plus orienté données de la vie entreprise.....	22
V-A - Le composant DiodeDB.....	24
V-B - Test rapide.....	25
VI - Touche finale.....	26
VII - Conclusion.....	28

I - Objectif : créer un composant TDiode pour FMX

En soit dessiner une diode dans une forme consiste simplement à poser un **TCircle**. S'il ne s'agit que de cela vous pourriez me dire qu'il n'est pas nécessaire d'en faire un composant.

C'est totalement vrai, même si quelques propriétés sont cachées (vous en découvrirez bientôt quelques unes) il n'est pas impossible d'y accéder par code.

De même s'il ne s'agit que de jouer sur la couleur de remplissage, et dans le cas d'un diode cela se résume souvent à ça, une fois de plus un peu de code suffit.

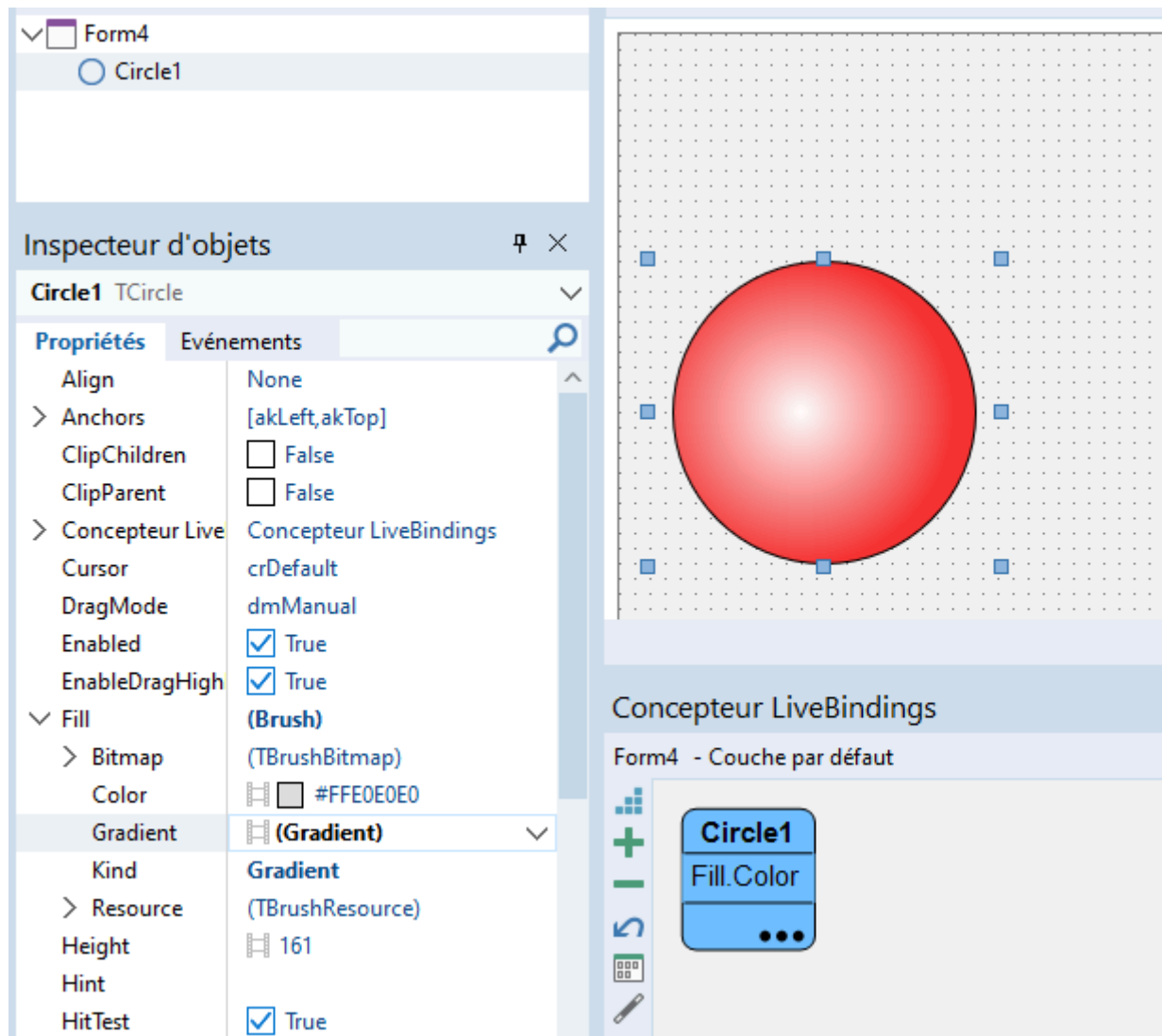
Pourquoi alors m'embarquer dans cette galère ? D'autres membres du forum DVP ont déjà proposé leurs solutions à l'exemple de **Gouyon** et des **ses composants d'affichages** et, bien sûr des sociétés tierces connues peuvent aussi le proposer.

Quelques raisons à cela :

- Le composant de base est simple et permet d'extraire les principes fondamentaux pour évoluer dans l'écriture de composants.
- Plusieurs propriétés ne sont pas utilisables par le concepteur de liens. Pour illustrer ce propos, l'image suivante montre que l'on peut lier la propriété **Fill.Color** d'un **TCircle** mais aucunement la couleur du gradient qui offre une rendu plus « 3D ». De même il faut que la liaison soit faite sur une donnée de type **TAlphaColor**.
- Souvent, les développeurs ne pensent pas à rendre le composant utilisable par LiveBindings, à l'exemple du **VCL.TTrackBar** et du **tutoriel d'Embarcadero** qui est à la base de cette série.



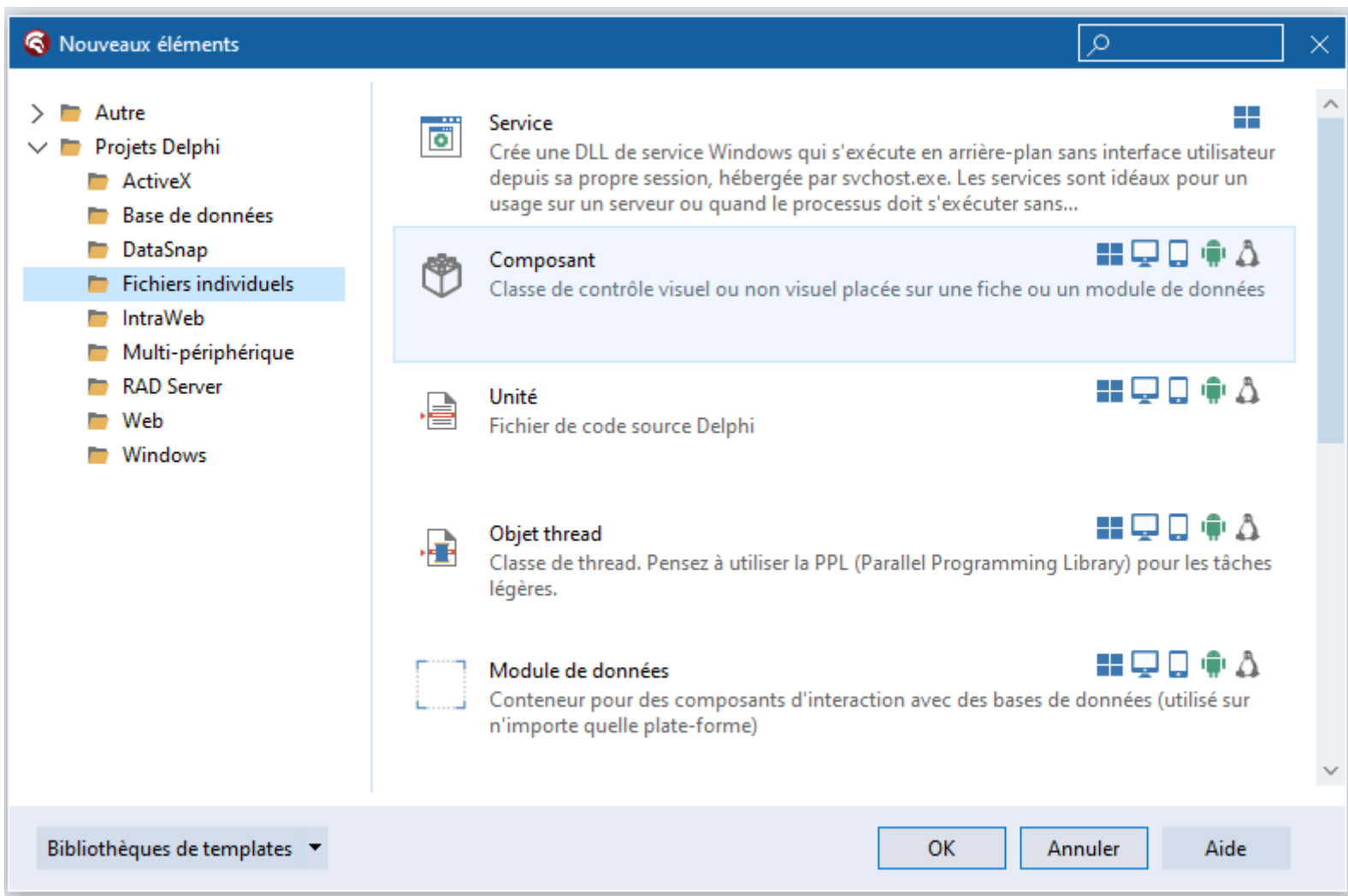
Les images qui suivront ont été prises en utilisant la dernière version de Delphi à ce jour soit Delphi 11.1 (Alexandria). Néanmoins la démarche peut très bien s'appliquer à des versions moins récentes comme la version Community (10.4.2). Un seul bémol, certaines versions, présentent quelques défauts (bogues) que j'essaierai de signaler.



TCircle

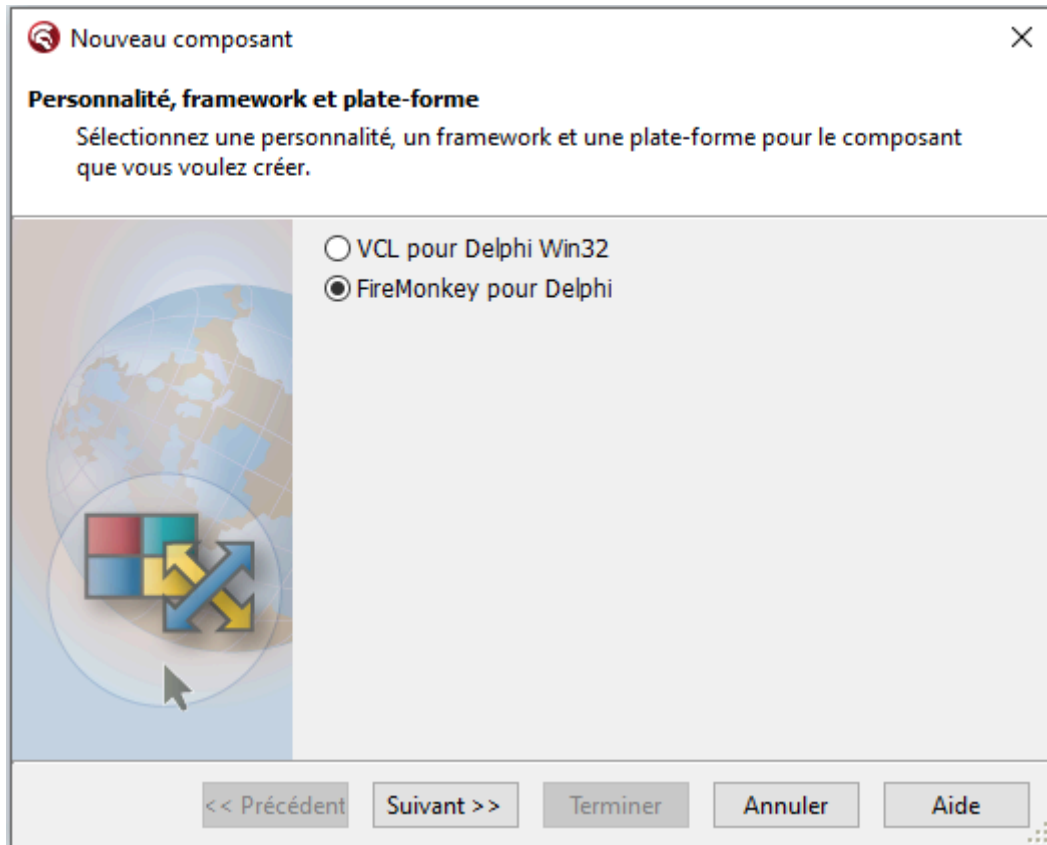
II - Création du paquet

Un développeur Delphi confirmé passera, très certainement, directement par le menu Fichier/Nouveau/Autres

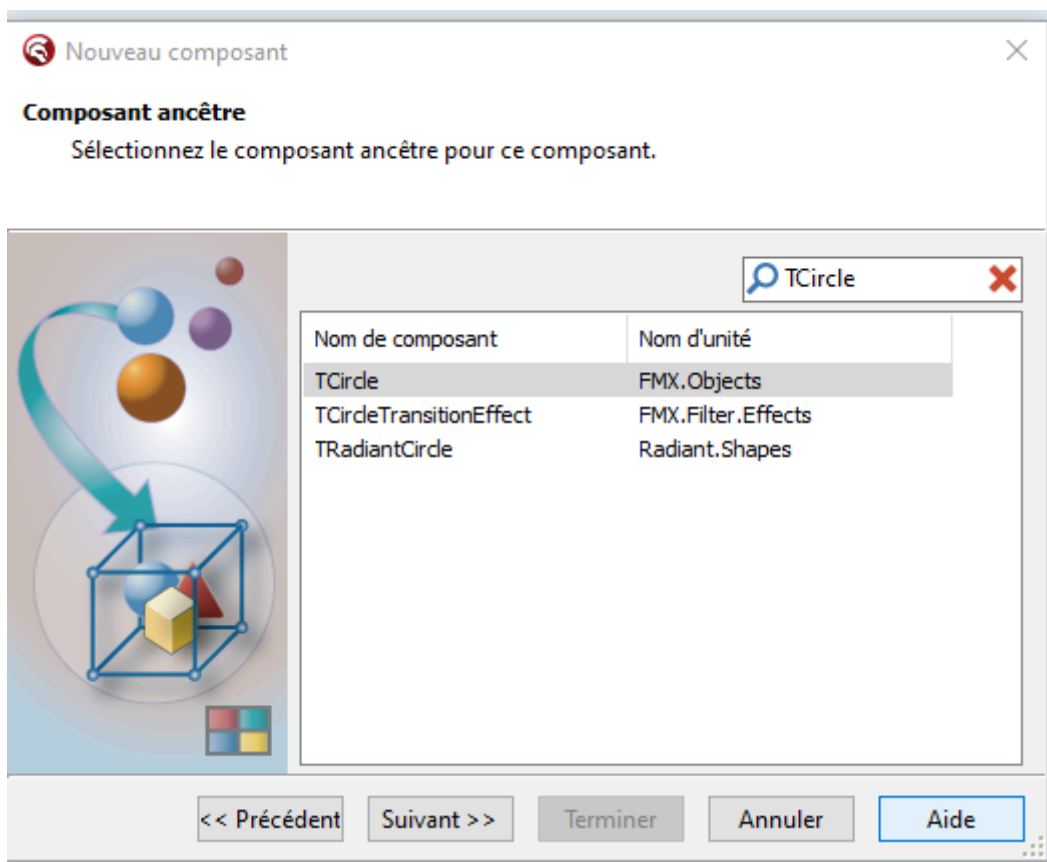


et commencera à écrire le/s unité/s nécessaires.


Un débutant préférera certainement utiliser le wizard (Composant/Nouveau composant) et suivre les 4 étapes




wizard étape 1



wizard étape 2


Nouveau composant

Composant
 Choisissez le nom du nouveau composant et le nom d'unité.




Nom de classe :

Page de palette :

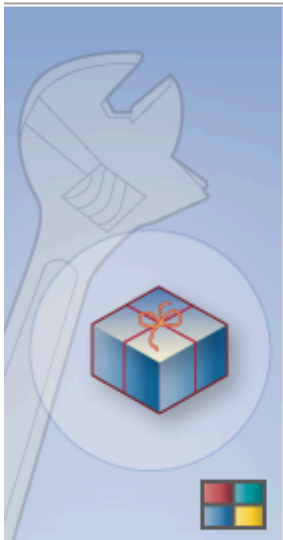
Nom d'unité :

Chemin de rech. :

wizard étape 3


Nouveau composant

Créer une unité
 Choisissez de créer une unité ou d'ajouter l'unité créée à un package actif. Une fois l'unité ajoutée à un package, elle peut être installée via le dialogue Installer des packages



☒ Créer une unité

☐ Installer dans un package existant

☐ Installer dans un nouveau package

☐ Ajouter une unité au projet Package1.dproj

wizard étape 4

avec l'avantage d'obtenir une première unité

Unité de départ

```
unit Lug.FMX.Diode;

interface

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

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

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('govel', [TDiode]);
end;

end.
```

II-A - Préambule

Il est temps de faire une parenthèse sur les noms de variables, propriétés et autres à utiliser.

Si pour un composant « maison » à usage unique les noms n'ont que peu d'importance, il n'en va pas de même au cas où l'objectif est de réutiliser ce dernier dans plusieurs programmes ou dans le cas d'un partage à large spectre (pour un public plus international), histoire de passer à la postérité ;-).

Une autre raison de bien choisir les noms de propriétés est que celles-ci seront proposées en ordre alphabétique dans l'éditeur de propriété.



Pas de stress, vous pouvez toujours par la suite utiliser le Refactoring ou les fonctions Rechercher/Remplacer

De fait le choix des noms commence dès celui de l'unité. Dorénavant les unités sont souvent préfixées du type de bibliothèque : VCL, FMX donc il serait bien de l'appliquer. L'équipe d'Embarcadero estime que ce n'est pas suffisant comme distinction et suggère (« fortement » ?) (1) que tout composant tiers ait un autre préfixe que ces deux-ci, histoire que le nom d'unité soit bien distinguée d'une unité « native ».

Ma future unité se nommera donc **Lug.FMX.Diode**.



Ici un peu de mon vernis de culture Celte entre en jeu. Lug ? Lug est un dieu celte, un polytechnicien (Salmidanach), à la fois dieu de la poésie, de la musique, du commerce, de la forge etc. il excelle en toute forme d'art. « Le Lugus (nom romanisé) gaulois est à la fois un dieu « lieur » par la magie, ainsi qu'un dieu « lié » avec des chaînes ». Ce nom me semblait donc tout indiqué.

Govel ? Traduction de forge en breton. Pas mal quand on sait le nombre de fois où j'ai remis au feu et à l'enclume mes composants.

II-B - Recensement des propriétés à proposer

C'est une étape importante, même si par la suite il se peut que vous apperceviez qu'il vous en faut d'autres, cela vous donnera quand même un bon point de départ.

Comment se décider ? J'avoue que ce n'est pas si évident que cela. Si l'objectif n'est qu'un changement de couleur comme montré dans la première image il me faut, évidemment une propriété en rapport avec la couleur. En définissant le gradient, je me suis aperçu que :

- je changeai le type de gradient (type radial),
- je nécessitai deux couleurs (celle de début et celle de fin qui simule un éclat plus brillant),
- un coup d'oeil au composant de Gouyon m'a convaincu d'ajouter une couleur supplémentaire pour « éteindre » la diode.

Donc déjà trois propriétés.

Le fait d'utiliser un gradient de type radial permet également de pouvoir déplacer le centre de celui-ci. Je vais en faire d'ores et déjà une propriété supplémentaire sans toutefois la rendre accessible, ce qui me permettra d'exposer par la suite un premier dérivé de mon composant.



Plusieurs versions de Delphi (X8 à 10.4.1)contiennent un bogue en ce qui concerne les fonctions de transformation radiale (**Gradient.RadialTransform**) vous pourriez donc avoir un effet un peu différent de celui souhaité avec ces versions.

Nommer ces propriétés va aussi être un peu « casse-tête ». Il est en effet assez agaçant d'aller les retrouver un peu partout dans l'inspecteur d'objet. Mon conseil essayez de nommer toutes les propriétés d'un même groupe avec un même préfixe. Par exemple, pour ce qui est des couleurs je vais utiliser le préfixe **color**, pour l'éclat le préfixe **bright**.

Pourquoi de l'anglais pour mes préfixes ? Pour les proposer à un plus large public, mais aussi parce que c'est le cas de toutes les propriétés du composant de base.



Pas de stress, le droit à l'erreur existe. Vous pourrez très bien reprendre les noms par la suite grâce à l'outil de refactoring ou l'utilisation de Rechercher/Remplacer proposé par l'éditeur de texte.

Pour récapituler :

- propriétés **ColorOn**, **ColorOff** de type **TAlphaColor**,
- propriétés **BrightColor** de type **TAlphaColor** et **BrightPosition** de type **TPosition**,
- une propriété **OnOff** de type **Boolean** pour indiquer si la diode sera allumée ou non.

```
type
  TDiode = class(TCircle)
  private
    { Déclarations privées }
  protected
    { Déclarations protégées }
  public
    { Déclarations publiques }
    property BrightPosition : TPosition;
  published
    { Déclarations publiées }
    property OnOff : Boolean ;
```

```
property ColorOn : TAlphaColor;
property ColorOff : TAlphaColor;
property BrightColor : TAlphaColor;
end;
```

Ensuite quelques autres prérequis vont être nécessaires, un constructeur et, comme dans celui-ci l'objet BrightPosition sera créé, un destructeur.

```
public
{ Déclarations publiques }
constructor Create(AOwner: TComponent); override;
destructor Destroy; override;
```

À cela s'ajoute la nécessité d'une procédure de dessin.

```
function Paint: Boolean; reintroduce;
```

Comme indiqué lors du **premier tutoriel de cette série** il faudra aussi ajouter trois éléments, deux procédures et une fonction pour la prise en charge des LiveBindings.

```
private
1. procedure ObserverToggle(const AObserver: IObservable; const Value: Boolean);
```

```
protected
1. function CanObserve(const ID: Integer): Boolean; override;
2. procedure ObserverAdded(const ID: Integer; const Observer: IObservable); override;
```

i Ces trois éléments nécessite l'ajout de l'unité **System.Classes**

Une fois tout ajouté, Ctrl+Shift+C permettra d'obtenir les déclarations complètes nécessaires.

```
TDiode = class(TCircle)
private
{ Déclarations privées }
FOnOff : Boolean;
FColorOn : TAlphaColor;
FColorOff : TAlphaColor;
FBrightPos: TPosition;
FBrightColor: TAlphaColor;
procedure ObserverToggle(const AObserver: IObservable; const Value: Boolean);
procedure SetColorOff(const Value: TAlphaColor);
procedure SetOnOff(const Value: Boolean);
procedure SetColorOn(const Value: TAlphaColor);
procedure SetBrightPosition(const Value: TPosition);
procedure SetBrightColor(const Value: TAlphaColor);
protected
{ Déclarations protégées }
function CanObserve(const ID: Integer): Boolean; override;
procedure ObserverAdded(const ID: Integer; const Observer: IObservable); override;
public
{ Déclarations publiques }
constructor Create(AOwner: TComponent); override;
destructor Destroy; override;
function Paint: Boolean; reintroduce;
property BrightPosition : TPosition read FBrightPos write SetBrightPosition;
published
{ Déclarations publiées }
property OnOff : Boolean read FOnOff write SetOnOff;
property ColorOn : TAlphaColor read FColorOn write SetColorOn;
property ColorOff : TAlphaColor read FColorOff write SetColorOff;
property BrightColor : TAlphaColor read FBrightColor write SetBrightColor;
end;
```

II-C - Codage

Tout d'abord proposer des propriétés par défaut si nous sommes dans l'IDE , et pour cela la procédure **Create** est tout indiquée.

Constructor

```
1. constructor TDiode.Create(AOwner: TComponent);
2. begin
3.   if not(csloading in ComponentState) then
4.     begin
5.       inherited;
6.       FOnOff:=True;
7.       FColorOff:=TAlphaColors.Lightgray;
8.       FColorOn:=TAlphaColors.Red;
9.       FBrightPos:=TPosition.Create(PointF(0.5,0.5));
10.
11.       FBrightColor:=TAlphaColors.Antiquewhite;
12.       Fill.DefaultColor:=FColorOn;
13.       Fill.Color:=FColorOn;
14.       Fill.Gradient.Color:=FColorOn;
15.       Fill.Gradient.Color1:=FBrightColor;
16.       Fill.Kind:=TBrushKind.Gradient;
17.       Fill.Gradient.Style:=TGradientStyle.Radial;
18.       Fill.Gradient.RadialTransform.RotationCenter.X :=FBrightPos.X;
19.       Fill.Gradient.RadialTransform.RotationCenter.Y :=FBrightPos.Y;
20.       Paint;
21.     end;
22. end;
```

N'oublions pas, à la ligne 9 je fais une création d'un objet de type **TPosition** il faudra donc le détruire.

Destructor

```
destructor TDiode.Destroy;
begin
  FBrightPos.Free; // libération
  inherited;
end;
```

Passons au dessin, la fonction **Paint** dont la subtilité la plus importante se situe dans la déclaration avec, à sa suite, le mot clé **reintroduire**.

Paint

```
function TDiode.Paint: Boolean;
begin
  if FOnOff then Fill.Gradient.Color:=FColorOn
  else Fill.Gradient.Color:=FColorOff;
  Fill.Gradient.Color1:=FBrightColor;
  Fill.Gradient.RadialTransform.RotationCenter.X:=FBrightPos.X;
  Fill.Gradient.RadialTransform.RotationCenter.Y:=FBrightPos.Y;
  Result:=True; // indiquera qu'il faut rafraichir l'affichage
end;
```

pour les autres propriétés, il faudra si nécessaire détecter le changement de valeur et forcer le dessin.

```
procedure TDiode.SetColorOff(const Value: TAlphaColor);
begin
  FColorOff := Value;
end;

procedure TDiode.SetColorOn(const Value: TAlphaColor);
begin
  if FColorOn<>Value then
  begin
    Fill.Color:=Value;
    FColorOn := Value;
  end;
```

```

    Paint;
end;
end;

procedure TDiode.SetBrightPosition(const Value: TPosition);
begin
    if (Value<>FBrightPos) then
        FBrightPos.Assign(Value); // attention, assign, pas :=
    end;

procedure TDiode.SetBrightColor(const Value: TAlphaColor);
begin
    if (Value<>FBrightColor) then
        begin
            FBrightColor:=Value;
            Fill.Gradient.Change; // force le calcul du gradient
            Paint;
        end;
    end;

procedure TDiode.SetOnOff(const Value: Boolean);
begin
    if FonOff<>Value then
        begin
            FonOff := Value;
            Paint; // force le dessin
        end ;
    end;
end;

```

II-D - Ajout de la partie LiveBindings

```

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

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

procedure TDiode.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
        end
        else
            Enabled := True;
    end;
end;

```

Reste alors à indiquer quelle propriété nous voulons voir apparaître lorsque nous utiliserons le concepteur de lien visuel.

1. [ObservableMembers('ColorOn',false)]
2. TDiode = class(TCircle)

À indiquer juste au dessus de la déclaration de classe.

Mais il faut surtout faire en sorte que les propriétés soient prises en compte avec le lieu et pour cela enregistrer celles-ci au niveau du lieu de l'unité **Data.Bind.Components**.

```
1. initialization
2.
3. Data.Bind.Components.RegisterObservableMember
4.   (TArray<TClass>.create(TDiode), 'ColorOn', 'FMX');
5.
6. Data.Bind.Components.RegisterObservableMember
7.   (TArray<TClass>.create(TDiode), 'OnOff', 'FMX');
```

Qu'il faudra bien sûr désinscrire par la suite, à la libération du composant.

```
1. finalization
2.
3. Data.Bind.Components.UnregisterObservableMember
4.   (TArray<TClass>.create(TDiode));
```

III - Installation et Tests

Ceux qui ont opté pour l'utilisation du wizard peuvent d'ores et déjà envisager une installation du composant, pour les autres une étapes supplémentaires est nécessaire : la rédaction de l'unité de design. Elle permet d'enregistrer le composant grâce à sa procédure **register**.

```
unit Lug.FMX.DiodeD;

interface
uses   System.Classes, Lug.FMX.Diode;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Govel', [TDiode]);
end;

end.
```

Pourquoi ai-je séparé en deux unités, contrairement à ce que le wizard génère ?

En priorité, pour séparer la partie design de la partie runtime, ainsi pourrai-je ne distribuer que la bibliothèque du runtime (bpl) si je le souhaite.

Enfin, il n'est pas exclu que je nécessite de créer des boîtes de dialogues pour saisir des valeurs de propriétés, ce qui nécessite l'inclusion d'unités uniquement accessible en design (**DesignIDE**) .



Lug.FMX.Diode sera renommée **Lug.FMX.DiodeR**. Le suffixe R, pour runtime, rendra le nom plus explicite.

III-A - Installer le composant

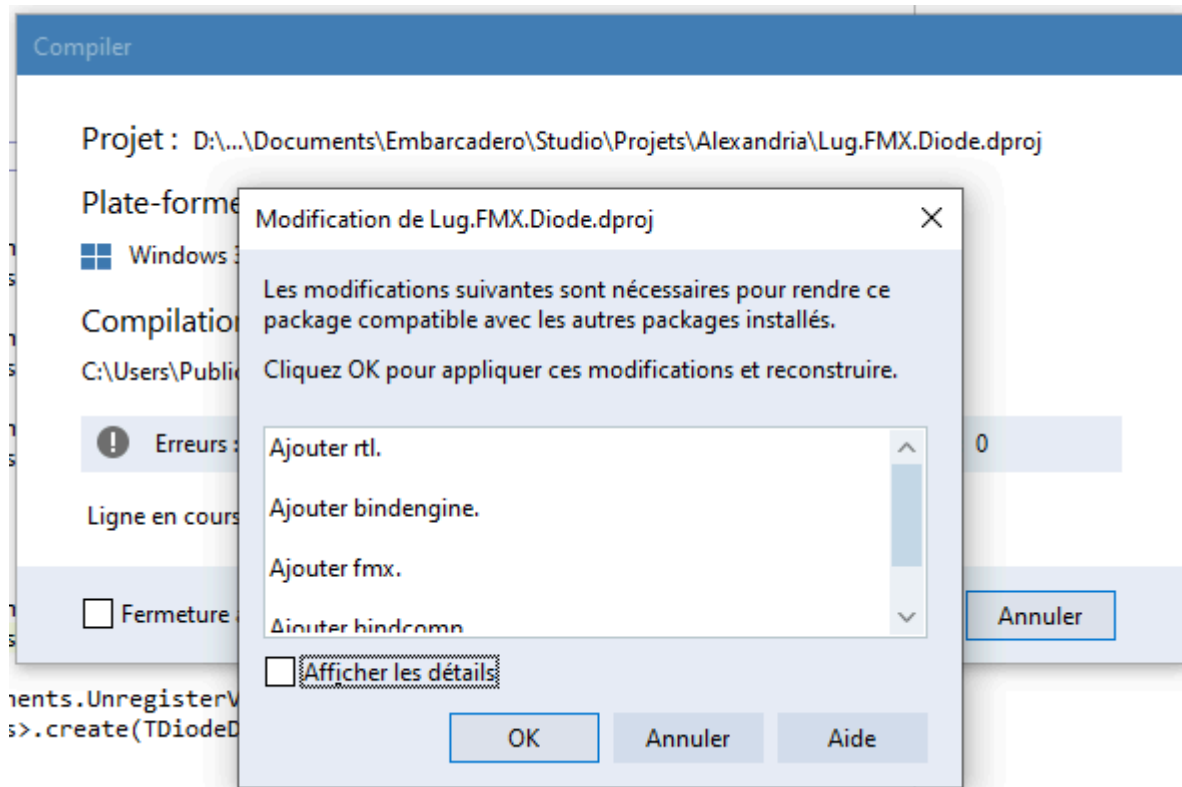
Compiler ou construire un composant se fait de la même manière que pour un programme, soit en utilisant les options du menu Projet soit en utilisant les raccourcis Ctrl+F9 et Maj+F9.

Par contre, ne vous avisez pas, par habitude, d'utiliser le menu Exécuter car un paquet n'est pas un programme. C'est un peu pour cette raison que je préfère utiliser le menu contextuel de la fenêtre projet qui a l'avantage de proposer ces options plus celle d'installation.

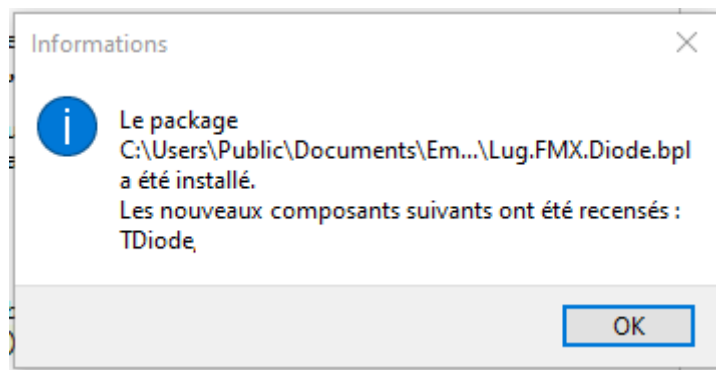


Notez qu'il y a aussi une possibilité de désinstaller le composant.

Une première compilation vous proposera certainement d'ajouter certaines unités à votre package, acceptez.



Une installation réussie sera validée par un écran d'information recensant les composants installés.



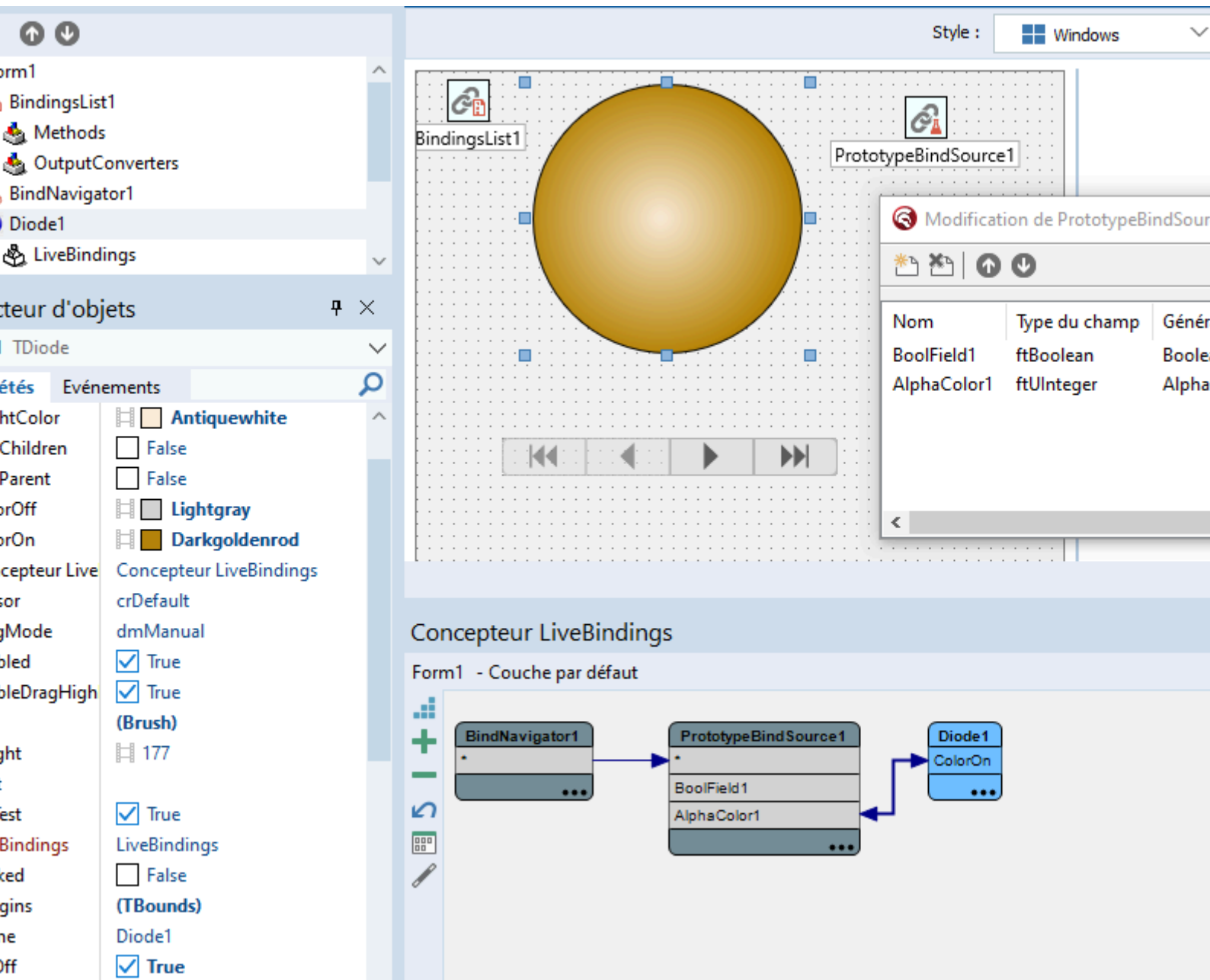
III-B - Programme de test

Créez un nouveau programme FMX et posez sur votre fiche vierge le composant Diode de la palette. Pour « simuler » des données, ajoutez un **TPrototypeBindSource** contenant deux champs : un champ booléen pour tester la propriété **OnOff**, un champ de type **TAlphaColor** pour la propriété **ColorOn**. Un **BindNavigator**, dont on enlèvera les boutons non nécessaires, permettra de naviguer au sein de l'ensemble de données créé.



Tant qu'à faire, limitez le nombre d'enregistrements créés.

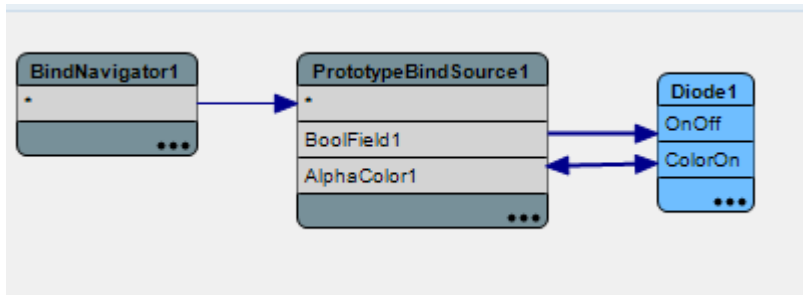
Pour cela modifiez la propriété **RecordCount** du **TPrototypeBindSource** selon votre convenance (une vingtaine devrait suffire).



premier test

Constatations :

- Les propriétés publiées sont bien visibles dans l'inspecteur d'objets.
- Une fois liée, notre diode réagit tout de suite et ce sans aucune ligne de code, dois-je, encore, le souligner ?
- Les propriétés **OnOff** et **ColorOff** n'apparaissent pas dans le lieu. Il faudra les sélectionner en utilisant le bouton Comment faire en sorte que les propriétés autres que celle indiquée au dessus de la déclaration de type du composant, reste encore, pour moi, un point à élucider.



- Le lien entre **ColorOn** et le champ **AlphaColors1** est de type bidirectionnel, ça, c'est absurde. Bien sûr il est toujours possible de modifier la propriété **Direction** du lien en **LinkDataToControl**, mais il y a certainement mieux à faire. En effet si l'on regarde le lien de la propriété **OnOff**, la flèche est bien uniquement du champ vers la propriété.

III-C - Corrections nécessaires sur les liaisons

Pourquoi donc, la liaison « principale » est-elle bidirectionnelle alors que les liaisons à d'autres propriétés ne le sont pas ? C'est ce qui m'a fait mettre au feu plusieurs fois mon ébauche.

Une étude des liaisons fourni une première piste.

Modification de Form1.BindingsList1

Catégories : (tous les LiveBindings) Liaisons rapides

Composants de liaison :

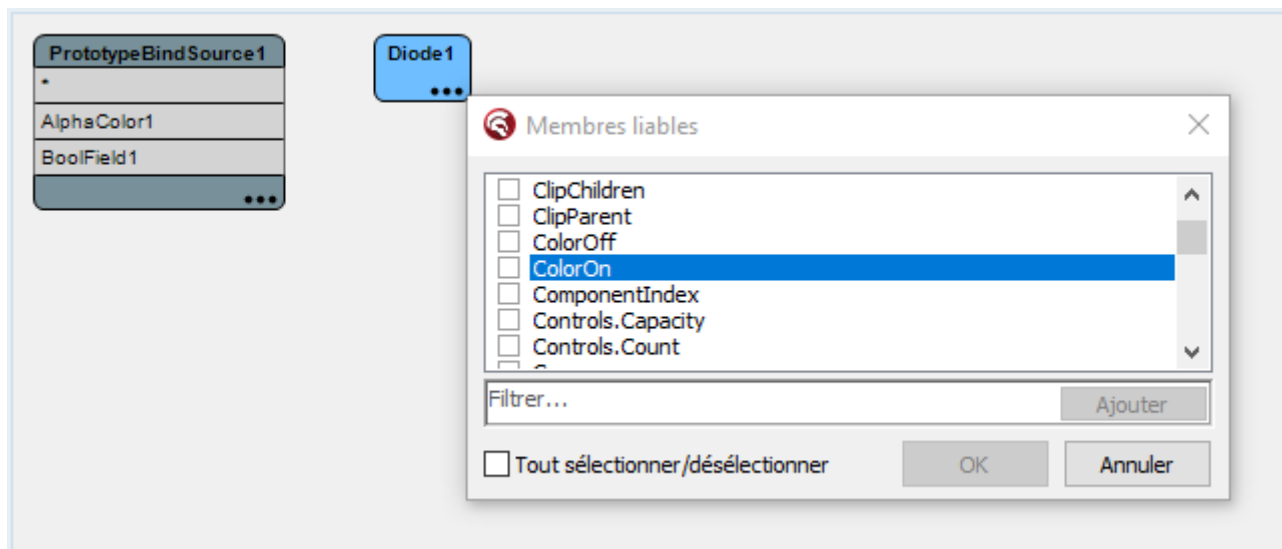
Nom	Description
LinkControlToField1	Lier le composant d'expression "Diode1" au composant "PrototypeBin
LinkPropertyToFieldOnOff	Lier le composant d'expression "Diode1" au composant "PrototypeBin

Form1 - Couche par défaut

Il est facile de remarquer que la première liaison, celle sur la propriété « principale », définie lors du recensement crée un lien de type **ControlToField** alors que la seconde est de type **PropertyToField**.

III-C-1 - Première solution

Une première solution consisterai, bien que cela aille à l'encontre du tutoriel Embarcadero, à ne pas recenser la propriété.



Effectivement, après essai, si l'on doit créer des liens il faut choisir la propriété et le lien établi sera bien alors de type **PropertyToField**. Seulement cette solution laisse quand même un goût amer, il faut cette opération de choix de propriété c'est pour le moins gênant ! Un futur utilisateur de ce composant ne le comprendrait pas.

III-C-2 - Solution

Après avoir vainement chercher dans la documentation, il m'a fallu fouiller dans les sources.

Embarcadero propose ce type de liaison pour un TLabel il me fallait retrouver l'endroit où ce composant, ou plutôt la propriété text du composant était recensée. Pas facile comme recherche ! Avec persévérance j'ai fini par trouver l'unité contenant la plupart des recensements, à savoir **Fmx.Bind.Editors**, suite à quoi il fut alors facile de retrouver le TLabel et la propriété Text et découvrir que la procédure utilisée se nommait en fait : **RegisterValuePropertyName**.

Les parties initialization et finalization sont donc à remplacer comme suit

```
initialization

//Data.Bind.Components.RegisterObservableMember
// (TArray<TClass>.create(TDiode), 'ColorOn', 'FMX');
Data.Bind.Components.RegisterValuePropertyName
  (TArray<TClass>.create(TDiode), 'ColorOn', 'FMX');

finalization

// Data.Bind.Components.UnRegisterObservableMember
// (TArray<TClass>.create(TDiode));
Data.Bind.Components.UnregisterValuePropertyName
  (TArray<TClass>.create(TDiode));
```

IV - Extension, rendre la propriété BrightPos accessible

La position de l'éclat de lumière, propriété **BrightPosition**, n'est pas publiée. Bien qu'il soit possible d'y accéder par code

```
Diode1.BrightPosition.x:=0.7 ;
Diode1.BrightPosition.y:=0.3 ;
```

Cette méthode n'est pas forcément agréable, il serait plus aisé de publier cette propriété pour y accéder au moment du design.

J'ai plutôt décidé de faire un nouveau composant dérivé de **TDiode**.

```
[ObservableMembers('ColorOn',false)]
TDiodeEx = class(TDiode)
published
{ Déclarations publiées }
property BrightPosition : TPosition read FBrightPos write SetBrightPosition;
end;
```

Cependant, les valeurs X et Y sont à limiter entre 0 et 1. Fallait-il donc créer pour cela un nouvel éditeur de propriété ? Nous allons voir que, heureusement, ce ne sera pas nécessaire.

IV-A - Les valeurs X et Y de la propriété BrightPos

Un objet **TPosition** fourni la possibilité d'ajouter un évènement **OnChange** (voir [documentation](#)). Je vais donc lui assigner une procédure qui prendra en charge les déplacements du centre du gradient pendant le design. Cette procédure sera indiquée dans la partie **protected**.

```
procedure TDiode.ChangeBrightPos(sender: TObject);
begin
if (not FInPaintTo) and (not IsUpdating) then
begin
if FBrightPos.X>1 then FBrightPos.X:=1;
if FBrightPos.X<0 then FBrightPos.X:=0;
if FBrightPos.Y>1 then FBrightPos.Y:=1;
if FBrightPos.Y<0 then FBrightPos.Y:=0;
Paint;
end;
end;
```



J'ai préféré mettre cette procédure dans la classe **TDiode** plutôt que d'avoir à surcharger certaines procédures existantes.

Nouvelle Classe TDiode

```
1. [ObservableMembers('ColorOn',false)]
2. TDiode = class(TCircle)
3. private
4. { Déclarations privées }
5. FOnOff : Boolean;
6. FColorOn : TAlphaColor;
7. FColorOff : TAlphaColor;
8. FBrightPos: TPosition;
9. FBrightColor: TAlphaColor;
10. procedure ObserverToggle(const AObserver: IObservable; const Value: Boolean);
11. procedure SetColorOff(const Value: TAlphaColor);
12. procedure SetOnOff(const Value: Boolean);
13. procedure SetColorOn(const Value: TAlphaColor);
14. procedure SetBrightPosition(const Value: TPosition);
15. procedure SetBrightColor(const Value: TAlphaColor);
16. protected
17. { Déclarations protégées }
18. function CanObserve(const ID: Integer): Boolean; override; { declaration is in
System.Classes }
19. procedure ObserverAdded(const ID: Integer; const Observer:
IObservable); override; { declaration is in System.Classes }
20. procedure ChangeBrightPos(sender : TObject);
21. public
22. { Déclarations publiques }
23. constructor Create(AOwner: TComponent); override;
24. destructor Destroy; override;
25. function Paint: Boolean; reintroduce;
26. property BrightPosition : TPosition read FBrightPos write SetBrightPosition;
27. published
28. { Déclarations publiées }
29. property OnOff : Boolean read FOnOff write SetOnOff;
```

Nouvelle Classe TDiode

```
30.     property ColorOn : TAlphaColor read FColorOn write SetColorOn;
31.     property ColorOff : TAlphaColor read FColorOff write SetColorOff;
32.     property BrightColor : TAlphaColor read FBrightColor write SetBrightColor;
33. end;
```

L'assignation de l'évènement se fera dans le constructeur de la diode.

Constructor

```
1.     constructor TDiode.Create(AOwner: TComponent);
2.     begin
3.         if not(csloading in ComponentState) then
4.             begin
5.                 inherited;
6.                 FOnOff:=True;
7.                 FColorOff:=TAlphaColors.Lightgray;
8.                 FColorOn:=TAlphaColors.Red;
9.                 FBrightPos:=TPosition.Create(PointF(0.5,0.5));
10.                FBrightPos.OnChange:=ChangeBrightPos;
11.                FBrightColor:=TAlphaColors.Antiquewhite;
12.                Fill.DefaultColor:=FColorOn;
13.                Fill.Color:=FColorOn;
14.                Fill.Gradient.Color:=FColorOn;
15.                Fill.Gradient.Color1:=FBrightColor;
16.                Fill.Kind:=TBrushKind.Gradient;
17.                Fill.Gradient.Style:=TGradientStyle.Radial;
18.                Fill.Gradient.RadialTransform.RotationCenter.X :=FBrightPos.X;
19.                Fill.Gradient.RadialTransform.RotationCenter.Y :=FBrightPos.Y;
20.                Paint;
21.            end;
22.        end;
```

Avant d'installer, n'oublions pas de modifier l'unité « design » **Lug.FMX.DiodeD** pour déclarer ce nouveau composant et l'inclure dans la palette.

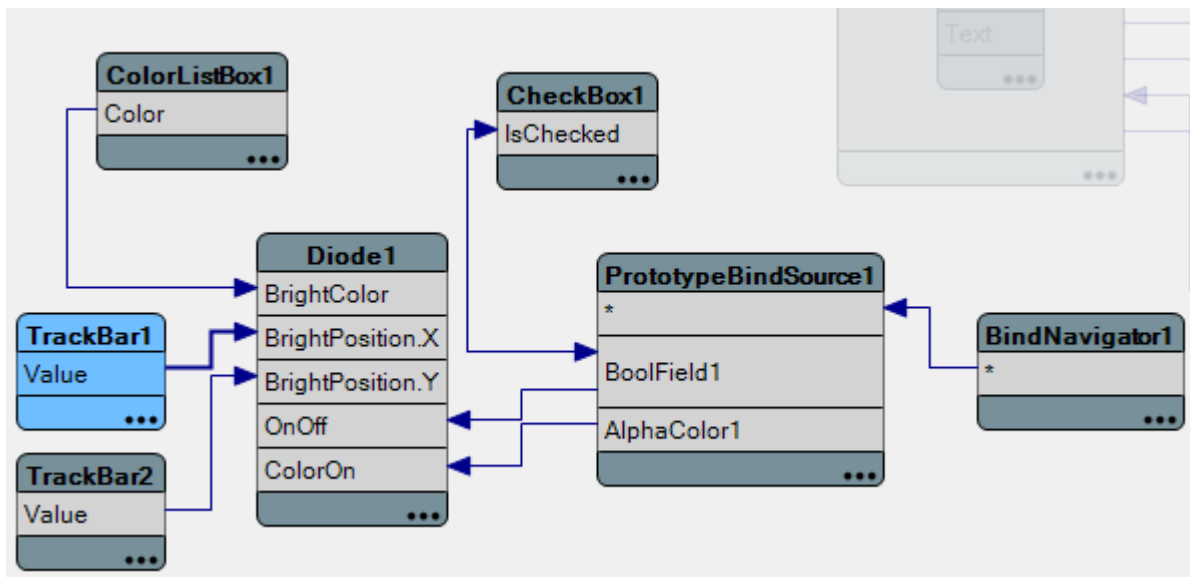
```
RegisterComponents('Govel', [Tdiode,TdiodeEx]) ;
```

IV-B - Test final

Après l'installation, nous avons maintenant deux composants **Diode** et **DiodeEx** dans la palette **Govel**. Positionnons quelques éléments supplémentaires sur la forme de départ :

- Remplaçons la diode par le nouveau composant **DiodeEx**.
- Un **ColorListbox** pour donner une couleur à l'éclat.
- Ajoutons deux **TrackBars** qui déplaceront la lumière intérieure. C'est plus un gadget qu'autre chose mais permettra de montrer l'utilisation des **CustomFormats**.
- Un **CheckBox** pour activer ou désactiver la diode, qui écrasera la valeur pouvant être contenue dans l'ensemble de données prototype.
- Puis établissons quelques liaisons.

- 21 -



Une fois ces éléments indiqués, le programme est fonctionnel et toujours sans aucune ligne de code.

Je reviens sur les liaisons avec les **TrackBars**. La valeur d'un trackbar sera comprise entre 0 et 100. Pour transformer la valeur il faut donc diviser par 100, donc la propriété **CustomFormat** contiendra la formule %s/100 .

A contrario, si la position pouvait être déplacée, l'interprétation serait dans la propriété **CustomParse** et exprimée ainsi : %s*100.

V - Un composant plus orienté données de la vie entreprise

Côté informatique de gestion, une diode pourrait servir à fournir une indication visuelle à l'utilisateur par exemple distinguer sur une fiche client les bons en vert, les litigieux en orange et les mauvais en rouge. Par contre il faut quelque chose de plus facile que le stockage de valeurs de couleurs (**TAlphaColors**) dans la base de données. L'idéal serait donc d'avoir un tableau d'équivalence entre le contenu d'un champs (toujours pour le même exemple V,O, ou R) et une couleur. Ce qui, codifié, pourrait être un objet de ce type.

Élément

```
/// Définition des équivalences
/// une valeur = une couleur
TDiodeDataColor = Class
private
    FDataValue: String;
    FDiodeColor: TAlphaColor;
    procedure SetDataValue(const Value: String);
    procedure SetDiodeColor(const Value: TAlphaColor);
published
    constructor Create(const DataValue : String; const DiodeColor : TAlphaColor);
    property DataValue: String read FDataValue write SetDataValue;
    property DiodeColor: TAlphaColor read FDiodeColor write SetDiodeColor;
end;
...

implementation

{ TDiodeDataColor }
constructor TDiodeDataColor.Create(const DataValue : String; const DiodeColor: TAlphaColor);
begin
    FDataValue := DataValue;
    FDiodeColor := DiodeColor;
end;
```

Élément

```
procedure TDiodeDataColor.SetDataValue(const Value: String);
begin
    FDataValue:=Value;
end;

procedure TDiodeDataColor.SetDiodeColor(const Value: TAlphaColor);
begin
    FDiodeColor:=Value;
end;
```



Pour faire bonne mesure, j'ai ajouté un constructor afin de rendre une codification plus simple lors d'ajout d'un couple valeur/couleur.

Toutefois, comment inclure cela au niveau d'un composant et de ses propriétés, pour une saisie facilitée au cours du design ? L'astuce, utiliser un éditeur de propriété. Heureusement il en existe un qui convient parfaitement, moyennant quelques ajustements : l'éditeur de collections.

Première étape, transformer la classe en classe de type élément de collection (***TCollectionItem***).

CollectionItem

```
TDiodeDataColor = Class(TCollectionItem)
```

Ensuite ajoutons une nouvelle classe de type ***TCollection***.

Collection

```
TItemCollection<T: TDiodeDataColor> = class(TCollection)
private
    FOwner: TComponent;
    FCollString: string;
public
    constructor create(CollOwner: TComponent);
    function GetOwner: TPersistent; override;
    procedure Update(Item: TCollectionItem); override;
end;
...

implementation

{ TItemCollection<T> }
constructor TItemCollection<T>.create(CollOwner: TComponent);
begin
    inherited create(T);
    FOwner := CollOwner;
end;

function TItemCollection<T>.GetOwner: TPersistent;
begin
    Result := FOwner;
end;

procedure TItemCollection<T>.Update(Item: TCollectionItem);
var
    str: string;
    I: Integer;
begin
    inherited;
    str := '';
    for I := 0 to Count - 1 do
    begin
        str := str + Format('%#x', [(Items[I] as TDiodeDataColor).FDiodeColor]);
        if I < Count - 1 then
            str := str + '-';
        end;
    end;
    FCollString := str;
end;
```

Cette partie est plus corsée et j'avoue avoir été chercher le modèle du code dans les sources d'Embarcadero. À remarquer, le constructeur un peu particulier et surtout la nécessité de la procédure **Update**.

V-A - Le composant DiodeDB

Il est désormais possible de créer le nouveau composant, à dériver de **TDiode** ou **TDiodeEx**.

Au préalable je défini un nouveau type (**TDefinedColors**) , afin de l'utiliser.

DiodeDB

```
TDefinedColors = Class(TItemCollection<TDiodeDataColor>);

[ObservableMembers('DataValue',false)]
TDiodeData = class(TDiodEx)
private
    FDataValue: String;
    FColors : TDefinedColors;
    function ColorsStored: Boolean;
    procedure SetColors(const Value: TDefinedColors);
    function FindColor(const Value: String) : TAlphaColor;
    procedure SetDataValue(const Value: String);
published
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
public
    property DataValue : String read FDataValue write SetDataValue;
    function Paint: Boolean; reintroduce;
published
    { Déclarations publiées }
    property Colors : TDefinedColors read FColors write SetColors
        stored ColorsStored;
end;
...
implementation
...
{ TDiodeData }

function TDiodeData.ColorsStored: Boolean;
begin
    result:=True;
end;

constructor TDiodeData.Create(AOwner: TComponent);
begin
    inherited;
    if (not(csloading in ComponentState)) OR (NOT Assigned(FColors)) then
        FColors:=TDefinedColors.create(Self);
end;

destructor TDiodeData.Destroy;
begin
    FreeAndNil(FColors);
    inherited;
end;

function TDiodeData.FindColor(const Value: String) : TAlphaColor;
var I: Word;
begin
    for I := 0 to FColors.Count - 1 do
        if TDiodeDataColor(FColors.Items[I]).DataValue= Value then
            begin
                Exit(TDiodDataColor(FColors.Items[I]).DiodeColor);
            end;
    Result := FColorOff;
end;

function TDiodeData.Paint: Boolean;
var ApplyColor : TAlphaColor;
```


DiodeDB

```
begin
  ApplyColor:=FindColor(FDataValue);
  FonOff:=ApplyColor<>FColorOff;
  if FonOff then Fill.Gradient.Color:=ApplyColor
    else Fill.Gradient.Color:=FColorOff;
  Fill.Gradient.Color1:=FBrightColor;
  Fill.Gradient.RadialTransform.RotationCenter.X:=FBrightPos.X;
  Fill.Gradient.RadialTransform.RotationCenter.Y:=FBrightPos.Y;
  Result:=True;
end;

procedure TDiodeData.SetColors(const Value: TDefinedColors);
begin
  FColors.Assign(Value);
end;

procedure TDiodeData.SetDataValue(const Value: String);
begin
  FDataValue := Value;
  Paint;
end;
```

Par rapport aux composants déjà écrit, vous remarquerez certaines nouveautés, à commencer par la déclaration de la propriété publiée **Colors** qui est suivie du mot clé **stored** ainsi que du nom d'une fonction (**ColorsStored**).



Important car va indiquer, via la fonction retournant **true**, que la collection va être stockée dans le dfm.

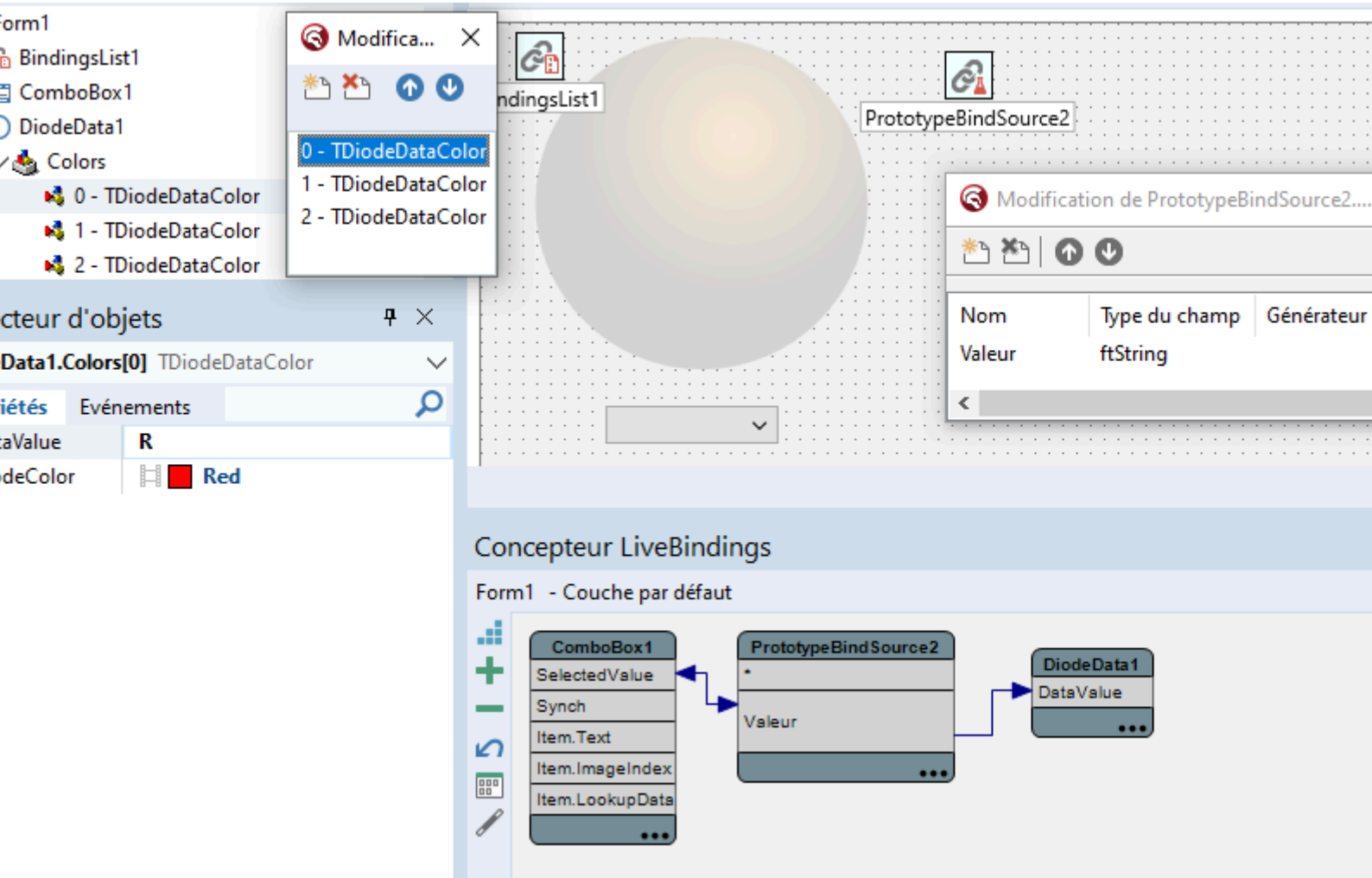
Cette nouvelle propriété nécessite que le constructeur et le destructeur soit surchargé pour réserver et libérer la mémoire.

Une réintroduction de la méthode **Paint** sera également nécessaire.

Enfin il faut retrouver la couleur à utiliser en fonction de la valeur du champ, ce sera chose faite avec une fonction de classe : **FindColor**.

V-B - Test rapide

Après ajout à la palette et installation de ce nouveau composant, un test rapide peut se réaliser en positionnant sur une forme, le nouveau composant et un combobox puis en créant deux liens.



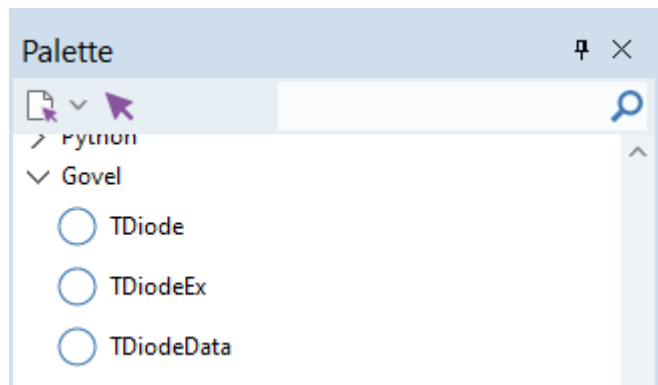
Le combobox contiendra les éléments (R,O,V).
L'éditeur de collection fera la jonction entre ces éléments et la couleur souhaitée.

i Ici, toute l'astuce est dans l'utilisation d'un TPrototypeBindSource (d'une seule ligne) pour faire la liaison entre la boîte de choix et le composant DiodeDB. C'est d'ailleurs la seule solution pour lier les propriétés de deux composants.

Seul bémol, il faudra exécuter le programme pour vérifier que les liaisons et le composant fonctionnent.

VI - Touche finale

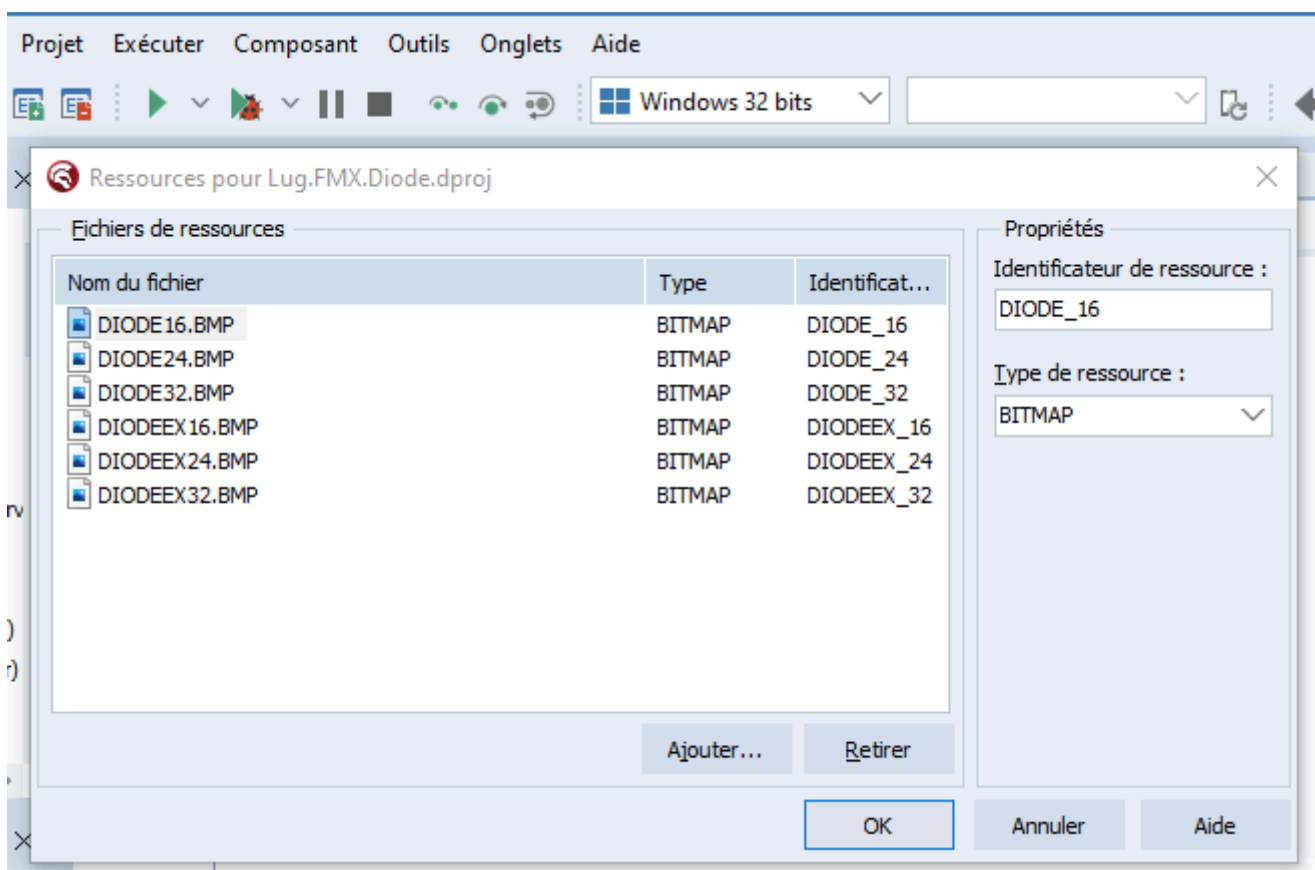
Il est désormais temps de peaufiner nos composants en ajoutant des icones à notre palette car, pour l'instant, les glyphes qui les représentent sont loin d'être significatifs.



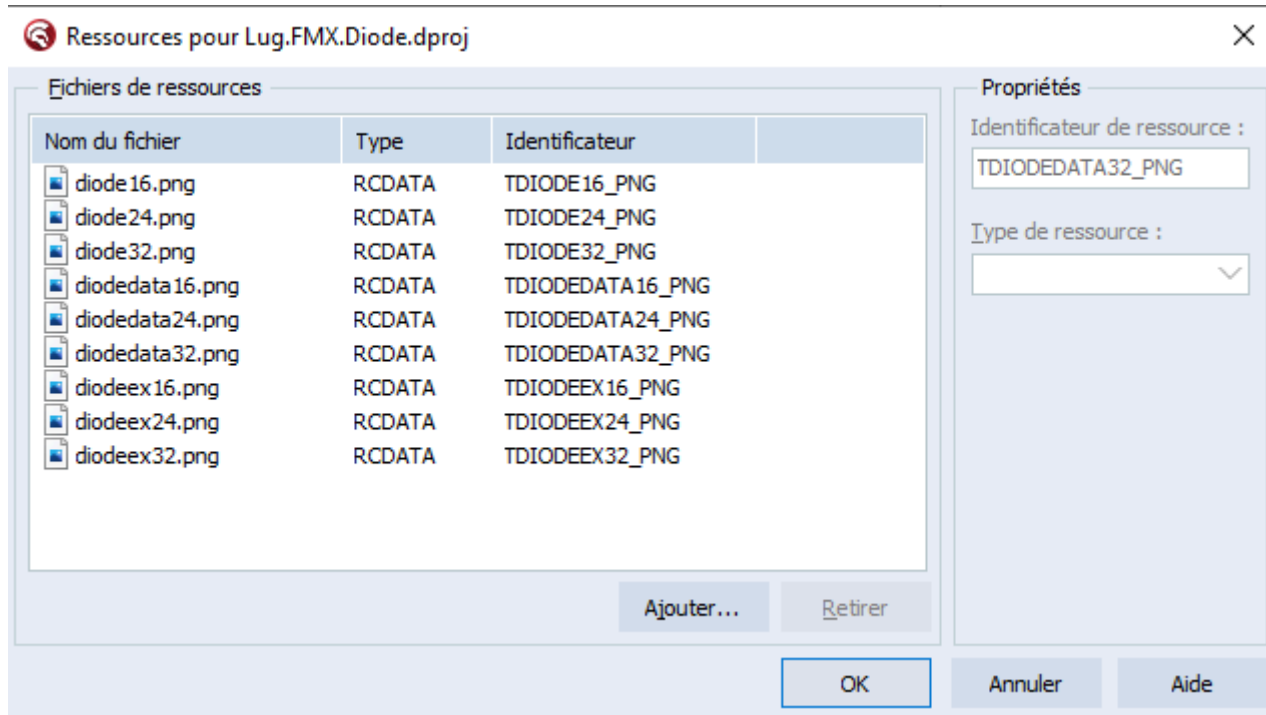
Bien évidemment la première tâche va être de composer des images. A l'ancienne, il nous faut , en théorie, des images de type bitmap de taille 16x16, 24x24 et 32x32. Une fois les diverses images créées il faut les incorporer au sein d'un fichier ressource.

Le plus simple, à partir du menu principal, sélectionnez **Projet/Ressources et images ...**

et ajoutez les images créées pour chaque composant en respectant un principe : le nom de la ressource doit être composé ainsi, <Nom du composant>_<taille>



J'ai donc préféré créer des images png.



Mais, selon le billet, un mix de bitmaps (32ppm) et de png serait l'idéal si vous vous souciez de rétrocompatibilité ou de vitesse de chargement de l'IDE.



Pour créer ces images, j'ai utilisé un programme concocté lors de mes études sur le composant FMX.Graphics.Tpath que vous pourrez retrouver soit dans [la section source du forum](#) soit dans un de mes dépôts GitHub <https://github.com/Serge-Girard/TPath>

Pour appliquer ces images vous devrez :



- Reconstruire (pas simplement compiler) le paquet
- Redémarrer l'IDE

▼ Govel



TDiode



TDiodeEx



TDiodeData

palette finale

VII - Conclusion

Avec ce parcours j'espère vous avoir démontré qu'il n'était pas si compliqué que cela de créer des composants réactifs aux liaisons. S'il y a quelque chose à retenir c'est que, pour les rendre réactifs :

- Il faut ajouter trois méthodes à la classe : la fonction **CanObserve** et les procédures **ObserverToggle** et **ObserverAdded**.
- Il faut recenser les liaisons dans la partie **initialization**, sans oublier de les libérer par la suite dans la section **finalization**.

- Il y a deux méthodes de recensement, **Data.Bind.Components.RegisterObservableMember** qui créera un lien bidirectionnel et **Data.Bind.Components.RegisterValuePropertyName**, plus pratique si le lien n'est qu'en sens unique données vers propriété.

1 : Cette recommandation a souvent fait, lors de conférence, grincer les dents de concepteurs de composants renommés dont Ray Konopka.