Formation C# – Delphi.NET – Delphi Win32
Développement & Sous-traitance

© Copyright 2005 Olivier DAHAN
Reproduction, utilisation et diffusion interdites sans l'autorisation de l'auteur. Pour plus d'information contacter odahan@e-naxos.com

Éditeur de propriété et éditeur de composant sous VCL.NET

Un composant n'est pas terminé tant qu'il n'est pas assorti de son fichier d'aide, d'exemples et de ses éditeurs de propriété ainsi que de son éditeur de composant. Il ne s'agit pas d'artifices réservés aux composants complexes. Le plus simple d'entre eux peut fort bien nécessité de saisir un nom de fichier par exemple et qu'en est-il alors si dans l'inspecteur d'objet il n'existe pas de moyen d'aller chercher ce nom dans une boîte de dialogue ? C'est simple, ce composant ne sera pas utilisé car il ne sera pas convivial… Si certains éditeurs de propriétés sont fournis par Delphi (pour les

types les plus courants comme les couleurs par exemple), il n'en va pas de même pour tous les types utilisables. Certaines propriétés de type objet peuvent réclamer un dialogue particulier. Que serait la propriété Font des composants visuels s'il n'y avait pas de boîte de dialogue pour faire son choix ? Plus avant, certains composants ne deviennent « pratiques » que s'ils sont dotés d'un éditeur de composant, dialogue complet permettant d'avoir une vision d'ensemble sur les propriétés et leurs effets. Pour tout cela il n'y a pas de magie mais du code. Un code particulier interagissant fortement avec l'IDE et devant suivre des conventions particulières. L'objet du présent article est de vous apprendre comment concevoir ces outils qui transforment un simple composant en un véritable accessoire fonctionnel de l'IDE.

Les Tools API

Les *Tools API* représentent un ensemble d'unités de services qui ne sont utilisables que dans le cadre de développements d'extension de l'IDE. Aucune des fonctions offertes ne peut apparaître dans une application « normale ». Cette séparation a été largement présentée dans notre article traitant de la conception des *packages* ; nous n'y reviendrons donc pas.

Aide au lecteur

Nous présentons les *Tools API* en introduction de cet article, car ils sont la matière première que nous utiliserons pour la conception des éditeurs de composant et de propriété. Toutefois, il vous est possible de passer directement à la partie concernant le codage des éditeurs en conservant un onglet sur la présente section, qui vous servira de documentation de référence.

Parmi les *Tools API*, on trouve de nombreuses unités qui ne nous intéressent pas ici directement, comme la création d'experts ou de plug'ins pour l'IDE que nous ne traiterons pas ici[1] ou la création d'un système de contrôle de version (VCS, *Version Control System*).

Les *Tools API* fonctionnent sur la base de classes et d'interfaces. L'IDE utilise les interfaces pour communiquer avec les éditeurs et les experts, ces derniers pouvant par le même biais interroger l'IDE.

Activation des Tools API

Les *Tools API* sont activés par l'ajout des unités correspondantes dans la clause `Uses` des unités qui définissent des extensions de l'IDE, mais aussi en ajoutant le *package* `Borland.Studio.Vcl.Design.dll` dans les

[1] La conception des experts a été traitée dans nos ouvrages précédents vers lesquels nous renvoyons le lecteur intéressé.

Requiert du projet. Ce *package* se trouve dans le répertoire Bin de Delphi et il faut l'ajouter et le localiser manuellement lorsqu'on appelle le dialogue « Ajouter une référence ». Lorsqu'on oublie ce détail, les unités des *Tools API* ne sont pas reconnues car ni leur code source ni le code compilé n'est fourni.

Sous BDS le développeur dispose de deux Tools API différentes, l'une est totalement managée et l'autre non. La raison est que l'IDE lui-même reste pour l'instant une application Win32. De fait, pour interagir en profondeur avec certaines fonctions de l'IDE il n'y a d'autres moyens que d'utiliser les Tools API non managées

Les Tools API managées permettent en revanche de concevoir des DLL .NET managées pouvant interagir avec l'IDE comme les DLL de type Wizard de Delphi 7. La partie managée des Tools API se trouvent dans la DLL Borland.Studio.ToolsApi. L'avantage des Tools API managés est qu'il est désormais possible de créer des extensions de l'IDE avec tout langage .NET, notamment C#.

Regardons de plus près ce que nous trouvons et qui concerne plus directement les éditeurs de composant et de propriété.

L'espace de nom Borland.Vcl.Design.DesignIntf

Cet espace de nom remplace l'unité DesgnIntf de Delphi 7 et 6 qui elle-même remplaçait l'unité DsgnIntf.pas, abandonnée depuis Delphi 5. Cet espace de nom contient des définitions de base qui restent toutefois compatibles avec Delphi Win32 et Kylix et qui aujourd'hui partagées par C#Builder, produit intégré à l'environnement BDS depuis Delphi 2005. Les définitions communes sont désormais plutôt stockées dans Borland.Studio.ToolsApi.dll riche en interfaces notamment toutes les interfaces IOTA. Cet assemblage regroupe les Tools API en code managé.

Remarque à propos de la compatibilité Delphi 5

Les références à DsgnIntf dans votre projet doivent être modifiées pour utiliser le nouvel espace de nom de BDS. Il peut aussi être nécessaire d'ajouter les autres espaces de nom (ayant même préfixe) DesignEditors, VCLEditors à la clause Uses. Vous devrez également ajouter la référence à la DLL indiquée plus haut dans la partie Requiert du paquet de *design* contenant les éditeurs, ainsi que les espaces de nom Borland.Delphi et tous les autres *packages* nécessaires à votre application IDE. De même, la classe TComponentEditor hérite de TBaseComponentEditor depuis Delphi 6, au lieu de TInterfacedObject dans Delphi 5 ; ceci peut avoir un impact sur votre code selon la façon dont il était conçu.

L'interface d'édition des propriétés

Les éditeurs de propriété sont créés à partir de `TBasePropertyEditor` ou, plus généralement, à partir de `TPropertyEditor`, qui en descend. En réalité, `TBasepropertyEditor` ne définit rien d'autre que le squelette ; tout le code utile se trouve dans la prise en charge d'une interface, `Iproperty`. Dans la classe de base, l'ensemble du code est abstrait (méthodes déclarées `abstract`) ; dans `TPropertyEditor`, une implémentation minimale est fournie pour toutes les méthodes abstraites. `TBasePropertyEditor` descend elle-même de `System.Object`, classe définie dans l'unité `System` du framework .NET et qui est utilisée comme ancêtre de tout objet.

L'interface `IProperty` est définie de la façon suivante :

```
IProperty = interface
procedure Activate;
   function AllEqual: boolean;
   function AutoFill: boolean;
   procedure Edit;
   function GetAttributes: TPropertyAttributes;
   function GetEditLimit: Integer;
   function GetEditValue(var Value: string): boolean;
   function GetName: string;
   procedure GetProperties(Proc: TGetPropProc);
   function GetPropInfo: MemberInfo;
   function GetPropType: Type;
   function GetValue: string;
   procedure GetValues(Proc: TGetStrProc);
   function HasInstance(Instance: MarshalByRefObject): boolean;
   procedure Revert;
   procedure SetValue(Value: string);
   function ValueAvailable: boolean;
end;
```

Il existe une variante pour Delphi 7, appelée `IProperty70`, et une autre depuis Delphi 8, appelée `IProperty80`.

`IProperty70` ajoute la propriété :

```
property IsDefault: boolean read GetIsDefault;
```

`IProperty80` ajoute la méthode :

```
procedure Edit(Host: IPropertyHost; DblClick: boolean);
```

Tout éditeur de propriété descendant de `TBasePropertyEditor` doit fournir une implémentation pour les méthodes de `IProperty` et `IProperty70` (sous Delphi 8, Delphi 2005 et BDS 2006 cette classe ne reconnaît pas `IProperty80`). En utilisant `TPropertyEditor` comme ancêtre, vous pouvez éviter cette écriture en ne surchargeant que les méthodes utiles à votre éditeur.

La signification des méthodes de `IProperty` est donnée dans le tableau 30.1.

Tableau 30.1

Les méthodes de l'interface IProperty

METHODE	DESCRIPTION
`Activate`	Cette méthode est appelée quand la propriété est sélectionnée dans l'inspecteur d'objet. La surcharge de cette méthode peut être intéressante pour fixer certains attributs de l'éditeur au dernier moment. Seuls les attributs `paSubproperties` et `paMultiSelect` ont besoin d'être définis (si besoin) avant l'appel à `Activate`.
`AllEqual`	Cette méthode est appelée lorsque plusieurs composants sont sélectionnés et qu'une propriété est elle aussi sélectionnée dans l'inspecteur d'objet. Si cette méthode retourne `True`, l'IDE appelle `GetValue` pour afficher la valeur dans l'inspecteur. Si la méthode retourne `False`, la valeur est affichée à vide dans l'inspecteur. `AllEqual` n'est appelée que lorsque l'éditeur possède l'attribut `paMultiSelect`.
`AutoFill`	Si l'éditeur définit l'attribut `paValueList`, cette méthode est appelée pour déterminer si les valeurs retournées par `GetValues` peuvent être sélectionnées de façon incrémentale.
`Edit`	Cette méthode est appelée lorsque l'utilisateur clique sur le bouton « … ». Elle a la charge d'afficher un dialogue (généralement) permettant d'éditer la propriété. L'éditeur doit posséder l'attribut `paDialog` pour activer ce fonctionnement.
`GetAttributes`	Cette méthode est appelée par l'inspecteur d'objet avant d'afficher la valeur de la propriété, pour savoir sous quelle forme il le fera (présence d'une `combo box`, d'un bouton d'appel de dialogue…). La méthode retourne un ensemble constitué d'attributs, dont la définition est donnée plus loin dans cette section.
`GetEditLimit`	Cette méthode est appelée par l'inspecteur d'objet pour connaître la taille limite (en caractères) de la zone d'édition de la valeur. Par défaut, cette limite est de 255 caractères.
`GetName`	Cette méthode est appelée par l'inspecteur d'objet pour connaître le nom à afficher de la propriété. On ne la surcharge qui si on désire changer ce nom visible (ce qui est rare).
`GetProperties`	Cette méthode doit être surchargée et doit appeler `Proc` passée en paramètre pour chaque sous-propriété définie dans l'éditeur. Un nouveau `TPropertyEditor` doit être fourni pour chaque sous-propriété. Ce mode n'est actif que si l'éditeur définit l'attribut `paSubProperties`.
`GetPropInfo`	Cette méthode retourne des informations sur la classe du composant sous la forme d'un `MemberInfo` du système de réflexion du *framework* .NET.
`GetpropType`	Doit retourner le pointeur vers les informations de type de la propriété éditée.
`GetValue`	Doit retourner la valeur de la propriété sous la forme d'une chaîne.
`GetValues`	Si `paValueList` est défini dans les attributs, c'est cette méthode qui

	sera appelée pour énumérer les valeurs possibles de la propriété et remplir la `combo box` dans l'inspecteur d'objet.
`SetValue`	Cette méthode est appelée lorsque la propriété doit être modifiée. L'éditeur de propriété doit être capable d'interpréter la chaîne passée en paramètre pour appeler la bonne méthode `SetXxxValue` selon le type réel. Si la valeur passée n'est pas acceptable (contrôles de bornes ou autre problème), l'éditeur doit lever une exception décrivant le problème rencontré.
`Revert`	Cette méthode remet la propriété dans son état initial.
`ValueAvailable`	Retourne `True` quand la valeur peut être atteinte sans causer d'exception. Cela a été ajouté pour gérer les valeurs de composants ActiveX qui sont mal écrits.

Les attributs des éditeurs de propriété

Voici la définition du type des attributs :

```
type TPropertyAttribute =
(paValueList, paSubProperties, paDialog, paMultiSelect,
 paAutoUpdate, paSortList, paReadOnly, paRevertable,
 paFullWidthName, paVolatileSubProperties, paVCL, paNotNestable,
 paDisplayReadOnly, paCustomDropDown, paValueEditable);
  TPropertyAttributes = set of TPropertyAttribute;
```

Lorsqu'on crée un éditeur de propriété, on doit indiquer de quel type sera cet éditeur. La signification de ce type est donnée dans le tableau 30.2

Tableau 30.2

Les types d'éditeurs de propriété.

ATTRIBUT	DESCRIPTION
`paValueList`	L'éditeur de propriété peut retourner une liste de valeurs pour la propriété. La procédure `GetValues` de l'éditeur sera appelée par l'IDE pour remplir la liste des valeurs acceptées et l'inspecteur d'objet proposera un affichage de type `combo box`.
`paSubProperties`	L'éditeur de propriété possède des sous-valeurs qui seront affichées sous la propriété avec une indentation positive à droite. Si cette option est définie, la méthode `GetProperties` de l'éditeur doit l'être également.
`paDialog`	Indique que la méthode `Edit` de l'éditeur déclenchera l'ouverture d'un dialogue. Un bouton avec trois points « … » sera affiché à droite de la valeur de la propriété dans l'inspecteur d'objet.
`paSortList`	L'inspecteur d'objet triera la liste retournée par `GetValues`.
`paAutoUpdate`	Si cette option est présente, la mise à jour par `SetValue` sera appelée à chaque changement dans la propriété au lieu d'un appel global une fois la saisie terminée. L'effet est

	celui de la propriété `Caption` d'un `TForm` par exemple : lorsque vous saisissez le texte dans l'inspecteur d'objet, le titre de la fenêtre se met à jour caractère par caractère durant la saisie.
`paMultiSelect`	Permet à la propriété d'être affichée lorsque plusieurs composants sont sélectionnés dans l'IDE en conception de fiche. La plupart du temps, un éditeur accepte ce mode, mais il arrive que cela ne soit pas approprié. La propriété `Name` d'un composant est de ce type, une modification globale n'aurait pas de sens et provoquerait des exceptions en cascade (les doublons étant interdits).
`paReadOnly`	Rarement utilisée, cette option rend la propriété non modifiable.
`paRevertable`	L'éditeur de propriété peut restaurer la valeur d'origine de la propriété.
`paFullWidthName`	La valeur de la propriété n'apparaît pas dans l'inspecteur d'objet. L'inspecteur d'objet utilise toute sa largeur (colonnes de noms et de valeurs) pour afficher le nom de la propriété.
`paVolatileSubProperties`	Toute modification de valeur de propriété provoque un nouveau calcul des sous-propriétés affichées (celles-ci sont réduites puis de nouveau développées).
`paVCL`	L'éditeur de propriété est un composant VCL (et non un composant CLX, ce qui n'est valable que sous Win32).
`paNotNestable`	L'inspecteur d'objet n'affiche pas la propriété lors de l'affichage des propriétés d'une référence développée.
`paDisplayOnly`	La propriété est seulement affichée et ne peut être modifiée.
`paCustomDropDown`	Permet d'implémenter une liste `drop down` personnalisée.
`paValueEditable`	La valeur est modifiable.

L'enregistrement des éditeurs

La méthode `RegisterPropertyEditor` permet d'enregistrer un nouvel éditeur de propriété.

Quand un composant est sélectionné dans l'inspecteur d'objet, l'IDE crée automatiquement une instance de l'éditeur de chaque propriété. Cet éditeur est créé en fonction du type de cette dernière. Par exemple, si la propriété est de type `Integer`, un éditeur de propriété `TIntegerProperty` sera créé. Les propriétés ne nécessitent généralement pas d'éditeur spécial, car leur type est reconnu par l'IDE qui fournit lui-même le bon éditeur. Par exemple, si la propriété est d'un type ordinal, l'éditeur par défaut saura comment limiter la saisie à l'étendue correspondant à ce type (comme `Byte` ou `Char`). Cela fonctionne aussi pour les types définis dans votre code. Si vous créez un type `TMonOrdinal = 1..10`, l'éditeur par défaut saura limiter la

saisie aux valeurs comprises entre 1 et 10 sans que vous ayez à écrire un éditeur spécifique. Il en va de même pour les types énumérés qui sont des ordinaux. L'inspecteur d'objet affichera une liste déroulante des différentes valeurs possibles automatiquement (grâce à la réflexion). Cela ne signifie pas qu'un éditeur de propriété est inutile pour les propriétés de ce type ! En effet il peut s'avérer beaucoup plus parlant pour le développeur utilisateur du composant édité si un éditeur adapté vient simplifier ou expliciter la signification des choix disponibles. Une propriété Char peut être par exemple accompagnée d'une éditeur affichant une table ASCII pour simplifier la sélection.

Ainsi, puisque techniquement l'EDI prend en charge les types simples, la création d'un éditeur de propriété ne se justifie que pour ajouter un dialogue (pour les couleurs par exemple) ou rendre plus conviviale la sélection d'une propriété (la liste des valeurs d'une propriété de type `TCursor` affiche le dessin de chaque curseur). En revanche un éditeur de propriété devient indispensable lorsque la propriété n'est pas reconnue par les éditeurs de base de l'IDE et de la VCL, ce qui est le cas de propriétés complexes comme une matrice ou tout type composite.

La procédure d'enregistrement des éditeurs permet l'entassement d'éditeurs concernant les mêmes types, l'IDE utilisera celui qui a été défini en dernier. Cette possibilité vous permet de surcharger les éditeurs déjà fournis en proposant les vôtres. Cela peut se faire pour les propriétés `TStrings` pour lesquelles vous pourriez souhaiter installer un éditeur qui ouvre un dialogue ressemblant à un mini-traitement de texte par exemple. L'ordre d'activation des packages dans lesquels les éditeurs sont définis est donc important. Si, dans l'IDE, vous activez un package en dernier et qu'il définit des éditeurs, ces derniers prendront le pas sur ceux déjà déclarés pour les mêmes types.

La déclaration de la procédure d'enregistrement est la suivante :

```
procedure RegisterPropertyEditor(PropertyType: PTypeInfo;
 ComponentClass: TClass; const PropertyName: string;
 EditorClass: TPropertyEditorClass);
```

Pour recenser un éditeur de propriété, il faut donner à `PropertyType` le pointeur d'informations de type renvoyé par la fonction `TypeInfo`. Le paramètre `ComponentClass` est conçu pour limiter l'éditeur de propriété à une classe de composant et à ses descendants. Si vous passez ce paramètre à `nil`, l'éditeur de propriété est associé au type de la propriété pour tous les composants.

`PropertyName` peut être défini pour limiter l'éditeur de propriété aux propriétés ayant un nom précis et le type spécifié. Si

`PropertyName` est une chaîne vide, l'éditeur de propriété est associé à toutes les propriétés du type spécifié.

Enfin, le paramètre `EditorClass` doit être le type de l'éditeur de propriété qui sera affiché lorsque la propriété est sélectionnée dans l'inspecteur d'objets. Cette classe doit être un descendant de `TBasePropertyEditor`, qui implémente l'interface `IProperty`.

L'enregistrement des catégories de propriétés

L'IDE de Delphi définit depuis Delphi 7 des catégories de propriétés pour l'inspecteur d'objet, ainsi que des filtrages possibles des propriétés affichées. Pour que les propriétés de vos composants soient reconnues dans une catégorie donnée, il est nécessaire de le déclarer. `DesignIntf` offre ainsi une série de procédures permettant d'effectuer cet enregistrement :

```
procedure RegisterPropertiesInCategory(const CategoryName: string;
  const Filters: array of const); overload;
procedure RegisterPropertiesInCategory(const CategoryName: string;
  ComponentClass: TClass; const Filters: array of string);
  overload;
procedure RegisterPropertiesInCategory(const CategoryName: string;
  PropertyType: PTypeInfo; const Filters: array of string);
  overload;
```

Le tableau 30.3 donne la signification des paramètres de ces procédures.

Tableau 30.3

Les paramètres d'enregistrement dans une catégorie

PARAMETRE	DESCRIPTION
`CategoryName`	C'est le nom de la catégorie. Il apparaît dans le menu « Voir » de l'inspecteur d'objet.
`Filters`	Ensemble de noms ou de types de propriétés à associer à la catégorie. Si `Filters` est un tableau de chaînes, il énumère les noms de propriétés ; si c'est un tableau de `Const`, il peut aussi indiquer des types.
`ComponentClass`	Classe du composant dont les propriétés doivent être associées à la catégorie.
`PropertyType`	Enregistrement contenant les informations du type de la propriété. On utilise `TypeInfo` pour passer cette information.

Vous noterez qu'il existe une seconde série de procédures portant le même nom, mais avec `Property` au singulier au lieu de `Properties` au pluriel, et qui permet de faire la même chose pour une propriété seule au lieu d'un groupe de propriétés.

Les éditeurs de composant

L'unité définit aussi le type `TBaseComponentEditor`, qui est celui dont doit descendre tout nouvel éditeur de composant (ou bien de `TComponentEditor`, qui implémente les méthodes abstraites en fournissant un comportement par défaut). Ces éditeurs doivent de plus implémenter l'interface `IComponentEditor`.

L'enregistrement s'effectue par la procédure suivante :

```
procedure RegisterComponentEditor(ComponentClass: TComponentClass;
  ComponentEditor: TComponentEditorClass);
```

Le paramètre `ComponentClass` est le type du composant auquel on associe l'éditeur et `ComponentEditor` est le type de l'éditeur de composant qui sera utilisé par l'IDE lors d'un double-clic sur un composant du type indiqué en premier paramètre.

Un éditeur de composant est automatiquement créé à chaque fois qu'un composant est sélectionné dans la fiche en cours. La création s'effectue en fonction de la classe du composant et selon les enregistrements d'éditeurs effectués avec la procédure décrite ci avant.

Quand le composant est double-cliqué, l'IDE appelle la méthode `Edit` de l'éditeur de composant. Quand le menu contextuel est appelé sur un composant, l'IDE appelle la méthode `GetVerCount` pour connaître le nombre d'items à ajouter dans le

menu puis, autant de fois que nécessaire, la méthode `GetVerb` pour obtenir le texte de chaque entrée du menu. Si l'une des entrées ainsi créées est activée par l'utilisateur, l'EDI appelle alors `ExecuteVerb` en passant en paramètre le numéro de l'item. Lorsqu'un éditeur modifie un composant, il doit absolument appeler `Designer.Modified` pour informer le *designer* qu'il doit mettre à jour la fiche en cours d'édition (notamment les valeurs affichées par l'inspecteur d'objet).

L'interface `IcomponentEditor` définit les méthodes données au tableau 30.4.

Tableau 30.4

Les méthodes de IComponentEditor

METHODE	DESCRIPTION
`Edit`	Est appelée sur le double-clic du composant dans une fiche en cours de modification.
`ExecuteVerb(index)`	Est appelée par l'IDE pour indiquer que l'utilisateur a choisi l'option numéro `index` dans le menu contextuel.
`GetVerb`	Est appelée par l'IDE pour connaître le texte à afficher dans chaque entrée du menu contextuel.
`GetVerbCount`	Est appelée par l'IDE pour savoir combien d'items doivent être créés dans le menu contextuel.
`PrepareItem`	Est appelée pour chaque nouvel item ajouté au menu contextuel en passant en paramètre l'item lui-même. Cela permet de personnaliser l'item en question, de lui créer des sous-éléments, voire de le désactiver en mettant sa propriété `Visible` à `False`.
`Copy`	Est appelée par l'IDE quand le composant est copié dans le presse-papiers. L'image du composant se trouve déjà dans ce dernier et l'intérêt de cette méthode est de pouvoir ajouter un nouveau format dans le presse-papiers. Il sera ignoré par Delphi, mais peut-être utilisé par une autre application (imaginons une fonction `Copy` qui place aussi une image *bitmap* de l'icône du composant dans le presse-papiers).
`IsInInlined`	Indique si le composant en cours d'édition se trouve à l'intérieur d'un composant de niveau supérieur qui peut être modifié à la conception et aussi incorporé dans une fiche. Par exemple, quand un composant est utilisé dans une `Frame` et que celui-ci est déplacé sur une fiche, `IsInInlined` renvoie `True` pour le composant s'il est incorporé dans la `Frame` sur la fiche, mais renvoie `False` lorsque le composant se trouve uniquement dans la `Frame`.
`GetComponent`	Retourne l'instance sélectionnée dans l'IDE.
`GetDesigner`	Retourne le concepteur en cours dans l'IDE.

L'espace de nom Borland.Vcl.Design.VCLEditors

L'unité `VCLEditors` définit les éditeurs de propriété et de composant de la VCL.

La liste des éditeurs définis dans la VCL, incluant ceux de `VCLEditors`, est donnée à la figure 30.1 suivante qui offre une vue arborescente sur les classes.

Lorsque vous créez un nouvel éditeur de propriété, vous utilisez systématiquement `TBasePropertyEditor` comme ancêtre ou bien son descendant `TPropertyEditor` qui implémente toutes les méthodes abstraites en fournissant un comportement par défaut. Il est toutefois utile de connaître les éditeurs existants, ne serait-ce que pour ne pas réinventer ce qui existe déjà ou réutiliser l'un des éditeurs de Delphi dans vos composants.

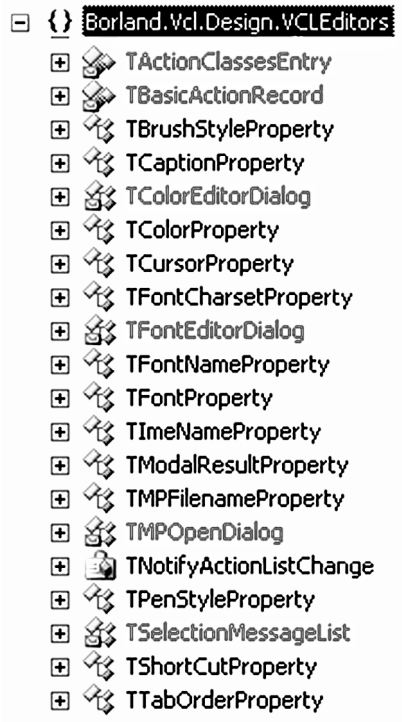

Figure 30.1

Les éditeurs de propriété de VCLEditors

Les éditeurs définis dans `VCLEditors` sont les suivants :

- `TBrushStyleProperty`
- `TCaptionProperty`
- `TColorProperty`

- TCursorProperty
- TFontCharSetProperty
- TFontNameProperty
- TFontProperty
- TimeNameProperty
- TModalResultproperty
- TMPFileNameProperty
- TPenStyleProperty
- TShortCutProperty
- TTabOrderProperty

L'unité VCLEditors définit aussi les procédures suivantes permettant de placer des flux objets dans le presse-papier et de récupérer de tels flux :

```
procedure CopyStreamToClipboard(S: TMemoryStream);
function GetClipboardStream: TMemoryStream;
```

Cette unité contient enfin un ensemble de procédures permettant de gérer les actions, notamment d'en enregistrer de nouvelles qui apparaîtront dans la liste de base de l'éditeur d'action.

L'espace de nom Borland.Vcl.Design.DesignEditors

Cet espace de nom contient les déclarations de nombreux éditeurs de propriété. Parmi les déclarations contenues dans cette unité, notons les suivantes qui présentent un intérêt particulier :

- TPropertyEditor – Dérive de TBaseProperty et implémente l'interface IProperty présentée plus haut. Cette classe est généralement utilisée comme ancêtre pour écrire un nouvel éditeur de propriété.
- TOrdinalProperty – C'est l'éditeur de propriété par défaut des ordinaux.
- TIntegerProperty – Dérive du précédent et gère les bornes de saisies selon le type (Byte, Word, …).

- `TCharProperty` – Pour la saisie des propriétés de type `Char`.

- `TEnumProperty` – Autorise la saisie des types énumérés. Le type `boolean` étant un énuméré, il est édité par défaut par cette classe et non plus par `TBooleanProperty`, qui n'existe plus depuis Delphi 6 (la classe existe encore mais est marquée `Deprecated`).

- `TInt64Property` – Sert à éditer tous les types `Int64` et dérivés.

- `TFloatProperty` – Sert à éditer les propriétés de type `Single`, `Double`…

- `TStringProperty` – Éditeur par défaut des types chaînes : `String`, `String[30]`…

- `TNestedProperty` – Utilisée comme classe de base pour créer des éditeurs de propriété imbriqués comme pour les types ensembles (utilisé par exemple dans `TSetElementProperty`).

- `TSetElementProperty` – Utilise les possibilités de création de sous-propriétés pour « éclater » les éléments d'un ensemble (`Set`) sous la forme de lignes indentées considérées chacune comme une valeur booléenne. Cet éditeur est créé par `TSetProperty`, l'éditeur des types ensemble.

- `TSetProperty` – Éditeur par défaut des ensembles (`Set of` …).

- `TClassProperty` – Éditeur par défaut des propriétés de type `Class`, par exemple, la propriété `DataSource` d'un `TDBEdit`.

- `TMethodProperty` – Éditeur par défaut des propriétés de type méthode. C'est le cas des événements notamment.

- `TComponentProperty` – Éditeur par défaut des propriétés de type `TComponent`. L'éditeur lui-même ne donne pas accès aux propriétés du composant sélectionné, mais permet sa sélection depuis tous les composants visibles dans la portée et dont le type est compatible avec la propriété en cours d'édition.

- `TInterfaceProperty` – Éditeur par défaut pour les propriétés de type `Interface`. Il permet de sélectionner une

interface compatible avec celle éditée parmi les interfaces implémentées par les composants dans la portée.

- `TComponentNameProperty` – Éditeur spécialisé pour les noms de composants. Il s'occupe notamment de vérifier que le nom choisi est unique et valide. Cet éditeur refuse la multisélection, ce qui est logique.

- `TDateProperty` – Éditeur pour les types dates (portion d'un `TDateTime`).

- `TTimeProperty` – Éditeur pour les types heures (portion d'un `TDateTime`).

- `TDateTimeProperty` – Éditeur pour les types dates et heures (`TDateTime` complet).

- `TVariantProperty` – Éditeur pour le type `Variant`.

Remarque
Nous le verrons dans les exemples à venir, il est souvent bien plus simple de partir d'un éditeur de propriété déjà existant, comme ceux indiqués ici, que de partir de zéro. Bien connaître les éditeurs fournis et les espaces de nom où ils sont définis permet de gagner beaucoup de temps.

L'unité déclare aussi la classe `TComponentEditor`, qui sert de base aux éditeurs de composant. Un éditeur par défaut est aussi défini pour gérer le double-clic sur un composant (`TDefaultEditor`). D'autres déclarations utiles se trouvent dans cette unité, comme la fonction `ClassInheritesFrom` qui permet de savoir si une classe descend d'une autre mais à partir du nom de cette dernière plutôt qu'à partir d'un type (comme le font les instructions `IS` ou `AS`)

Les éditeurs de propriété

On peut voir au départ un composant comme un objet qui descend de la classe `TComponent`. Cela n'est pas faux, mais comme nous l'avons vu dans d'autres articles, de nombreuses autres choses sont nécessaires pour en faire un acteur de l'IDE (séquence d'enregistrement, icône pour la palette…). Nous vous avons aussi présenté la stratégie de création des *packages* dans lesquels vos composants prennent place. Tout cela s'inscrit dans un processus global qui intègre l'écriture d'éditeurs de propriété. Nous allons voir ici comment ces derniers s'insèrent dans la conception d'un composant.

Définition

Les composants installés dans l'IDE de Delphi bénéficient des services de l'inspecteur d'objet lorsque vous les posez sur une fiche. Ce dernier affiche la liste des propriétés publiées d'un composant pour en autoriser la modification.

Comme vous l'avez constaté, l'inspecteur d'objet n'offre pas les mêmes possibilités selon les propriétés. Une propriété de type `String` sera modifiée simplement en tapant le texte dans la cellule de l'inspecteur d'objet. Une propriété de type `TAlignment` affichera une liste déroulante proposant les diverses options. Certaines propriétés possèdent aussi un bouton qui appelle un dialogue. C'est le cas des propriétés de type `TColor`. Certaines autres ont des listes qui comportent en plus du nom des options une représentation graphique de celles-ci (les propriétés `TColor` ou `TCursor` par exemple).

Cette souplesse et cette grande diversité des comportements de l'inspecteur d'objet n'ont rien de magique. Tout repose sur une bonne conception de l'IDE, laquelle permet à tout créateur de composant d'enregistrer des éditeurs de propriété personnalisés qui seront reconnus et utilisés par l'inspecteur d'objet au bon endroit et au bon moment.

Un éditeur de propriété est ainsi un descendant d'une classe particulière (nous verrons cela plus bas), qui implémente un ensemble de comportements et qui sera explicitement enregistré dans l'IDE d'une façon assez similaire à celle d'un composant. La différence majeure sera qu'un composant aura une existence visible dans une palette, alors qu'un éditeur de propriété ne se verra qu'à l'effet qu'il produit dans l'inspecteur d'objet.

Un premier exemple

Avant de nous lancer dans des choses plus sophistiquées, nous allons, par un premier exemple rudimentaire, vous prouver qu'écrire un éditeur de propriété peut être quelque chose de très simple.

Delphi intègre de très nombreux éditeurs qui couvrent les besoins de tous les composants de la VCL. C'est le cas notamment pour les propriétés de type `TFileName` des dialogues et de quelques autres composants. Cela n'était pas le cas dans toutes les versions précédentes. Il est difficile donc de trouver un exemple très simple sans redéfinir un éditeur existant. C'est pourtant ce que nous allons faire ici, pour vous démontrer à quel point il est facile de modifier

le comportement de Delphi, même pour des composants déjà installés que vous n'avez pas écrits.

Nous allons ainsi définir un éditeur pour les propriétés `FileName`, peu importe le composant dans lequel cette propriété apparaît. Comme les noms de fichiers sont normalement définis avec le type `TFileName` (basé sur `String`), nous allons partir d'un éditeur existant pour les chaînes, `TStringProperty`. Vous voyez ici l'intérêt de connaître les éditeurs existants tels que la figure 30.1 les montre.

Code source des Tools API

Une partie du code source des unités des Tools API est fourni dans le répertoire `/source/ToolsAPI` de BDS. Nous vous encourageons à lire ces fichiers qui regorgent de commentaires.

Comme `TStringProperty` fait déjà tout le travail, nous ne modifierons que la méthode `GetAttributes` : ajoutons l'attribut `paDialog`, qui indique que notre éditeur gère un dialogue, ce qui se matérialisera par un bouton « … » à droite de la valeur de la propriété dans l'inspecteur d'objet. Delphi propose déjà cette option ; nous ferons la différence grâce au titre du dialogue qui nous prouvera que c'est bien le nôtre qui a été appelé. Ensuite, nous définirons la méthode `Edit` puisque c'est elle qui est appelée lorsque l'utilisateur clique sur le bouton.

Le code de cette unité est réellement trivial, le voici :

```
{-----------------------------------------------------------------
 Unit Name: FName
 Author:    od
 Purpose:   Livre BDS - Editeur de propriété rudimentaire
 History:   Juil-05 V 2.1 .NET
-----------------------------------------------------------------}

unit FName;

interface
uses Sysutils,
 Borland.Vcl.Design.DesignIntf,
 Borland.Vcl.Design.DesignEditors;

Type
// l'éditeur de propriété descend de l'éditeur de chaîne
 TFileNameProperty = Class(TStringProperty)
  function GetAttributes : TPropertyAttributes; override;
  procedure Edit; override;
 end;

procedure Register;

implementation
uses dialogs;  // pour utiliser le dialogue d'ouverture de fichier
```

```
function TFileNameProperty.GetAttributes: TPropertyAttributes;
begin
 Result := [paDialog]; // on indique que la propriété supporte un dialogue
end ;

// appelé par l'inspecteur d'objet
// lorsque l'utilisateur clique sur le bouton "..."
procedure TFileNameProperty.Edit;
begin
 with TOpenDialog.Create(nil) do // on crée le dialogue
  try
   Title := 'Exemple Livre BDS : ' +GetName; // le nom de la propriété
   if GetValue<>'' then
    begin // on extrait le chemin et le nom de fichier
     InitialDir := ExtractFilePath(GetValue);
     FileName   := ExtractFileName(GetValue);
    end;
   Filter := 'Tous (*.*)|*.*';
   Options := Options + [ofPathMustExist, ofFileMustExist];
   // on exécute le dialogue et on modifie la propriété si ok
   if Execute then SetValue(Filename);
  finally
   Free;
  end;
end;

// enregistrement de l'éditeur dans l'IDE
procedure Register;
begin
 RegisterPropertyEditor(TypeInfo(TFileName),nil, '',
                        TFileNameProperty);
end;

end.
```

Comme vous pouvez le voir, il n'y a là rien de bien compliqué. Certes, cela ne fait pas grand-chose non plus, mais c'est une chose essentielle ! Si vous développez un composant dont l'une des propriétés attend un nom de fichier, et si cette propriété n'est pas de type `TFileName`, vos utilisateurs seront bien contents que vous leur fournissiez un petit éditeur comme celui-ci (que vous adapterez pour qu'il reconnaisse votre propriété d'après son nom par exemple).

Écrire le code de l'éditeur est une chose, il faut maintenant l'intégrer dans un *package* de *design*. Voici le code de ce dernier :

```
package FileNameEditor;

{$ALIGN 0}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
```

```
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO ON}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST OFF}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$IMPLICITBUILD OFF}
{$DESIGNONLY}

[assembly: AssemblyDescription('')]
[assembly: AssemblyConfiguration('')]
[assembly: AssemblyCompany('')]
[assembly: AssemblyProduct('')]
[assembly: AssemblyCopyright('')]
[assembly: AssemblyTrademark('')]
[assembly: AssemblyCulture('')]

[assembly: AssemblyVersion('1.0.*')]

[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile('')]
[assembly: AssemblyKeyName('')]

requires
  System.Drawing,
  Borland.Delphi,
  Borland.Studio.Vcl.Design,
  Borland.VclRtl;

contains
  FName in 'FName.pas';

end.
```

Il n'y a pas grand-chose à dire à propos de ce code, sauf qu'il s'agit bien d'un *package* de *design*, voir la ligne `{$DESIGNONLY}`, et qu'il intègre la gestion de *design* de l'IDE par la présence dans sa clause `Requires` de la référence à `Borland.Studio.Vcl.Design`.

La partie `Contains` ne contient qu'une chose, le nom de l'unité présentée plus haut qui contient le code de l'éditeur.

Une fois le *package* compilé et installé dans l'IDE, nous pouvons ouvrir un nouveau projet, poser un composant qui possède une propriété `FileName`, un `TOpenDialog` par exemple, et cliquer sur le bouton « … » de la propriété en question dans l'inspecteur d'objet. Le résultat se passe d'autres commentaires (voir figure 30.2).

Rappel

Pour installer une DLL package .NET dans l'IDE de Delphi il faut utiliser le menu Composant / Composants .NET installés qui ouvre un dialogue. Ce positionner alors sur l'onglet Composants VCL .NET et ajouter la DLL dans cette page.

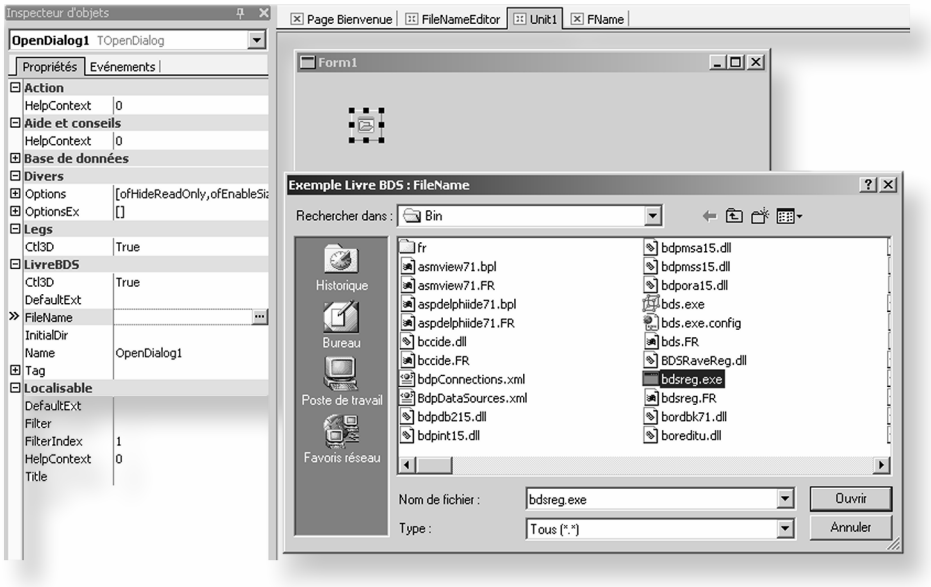

Figure 30.2

L'éditeur de propriété TFileName en action

La boîte de dialogue de sélection de fichier est ouverte et son titre indique « Exemple Livre BDS : FileName », ce qui prouve bien que c'est notre éditeur qui a été appelé et non l'éditeur par défaut de l'IDE.

Les interfaces et les comportements sophistiqués

Les éditeurs de propriété dérivent de la classe `TBasePropertyEditor` et de ses descendants, mais ce n'est pas tout. Certains éditeurs peuvent fournir des services supplémentaires qu'il aurait été trop lourd d'imposer à toute la hiérarchie.

C'est là qu'interviennent les `Interfaces`, qui, contrairement à une idée trop répandue sous Win32, ne sont pas limitées à COM.

Une interface est un contrat. Les classes qui désirent remplir ce contrat déclarent qu'elles prennent en charge telle ou telle interface et en implémentent les méthodes. À l'exécution, il est possible de demander à un objet s'il répond ou non à un contrat particulier et, dans l'affirmative, d'avoir accès aux méthodes qui constituent ce contrat en dehors de toute référence à l'instance réelle.

Ainsi, Delphi propose aux éditeurs de composant de prendre en charge des interfaces qui en étendront les fonctionnalités. D'ailleurs, en réalité, l'IDE ne communique avec les éditeurs qu'au travers des interfaces, puisque 90 % de la partie utile d'un éditeur de propriété consiste à répondre à l'interface `IProperty` que nous avons présentée plus haut.

Les interfaces disponibles sont les suivantes :

- `ICustomPropertyDrawing` – Permet à un éditeur de prendre le contrôle du dessin du nom de la propriété et de sa valeur dans l'inspecteur d'objet.

- `ICustomPropertyListDrawing` – Permet à un éditeur de prendre le contrôle du dessin de la liste déroulante pour les propriétés qui présentent cette option.

Nous allons utiliser cette seconde interface pour le prochain exemple. Celui-ci va consister à modifier l'affichage de la liste déroulante pour les propriétés de type `Boolean`, afin d'afficher une petite croix rouge devant `False` et une petite coche verte devant `True`. La technique démontrée est utilisable pour personnaliser toute liste déroulante dans l'inspecteur d'objet.

Avant de passer au code, regardons l'effet visuel qui sera obtenu, cela vous aidera à mieux comprendre le but recherché (figure 30.3)

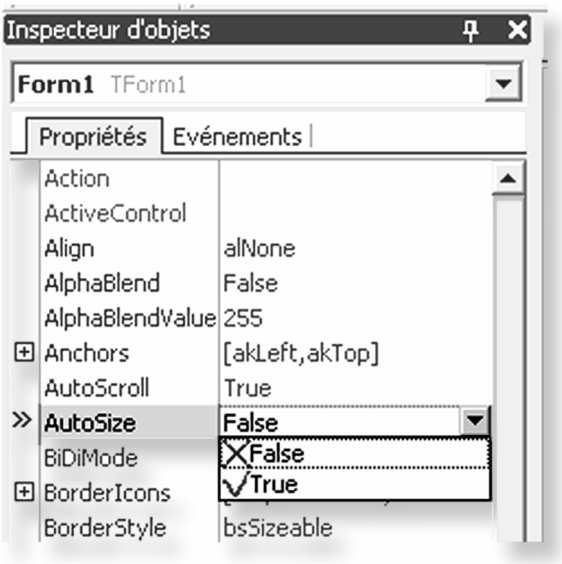

Figure 30.3

Modification de l'éditeur des propriétés booléennes

Comme le montre la capture écran, la propriété `AutoSize`, qui est de type booléen, possède maintenant des petites icônes devant chaque valeur quand on ouvre la liste déroulante. Bien entendu, toutes les propriétés de même type possèdent désormais cet artifice. Cela est lié à notre éditeur, mais aussi à la séquence d'enregistrement de celui-ci.

Voyons le code de l'éditeur :

```
// Livre BDS  - Editeur de propriété boolean
// OD – Jui 05
// V 2.1 .NET

unit BoolEdit;

interface

uses
  windows,
  classes,
  Types,              // définition de TRec notamment
  ExtCtrls,           // La classe TShape se trouve là
  // Déclaration des espaces de nom de design
  Borland.Vcl.Design.DesignIntf,
  Borland.Vcl.Design.DesignEditors,
  Borland.Vcl.Design.VCLEditors,
  Borland.Vcl.WinUtils,  // contient une function remplaçant HInstance
  Graphics;           // pour manipuler les graphiques

type
  // L'éditeur de propriété
  TGraphBooleanProperty =
              class(TEnumProperty,ICustomPropertyListDrawing)
```

```
    // support de l'interface ICustomPropertyListDrawing
    procedure ListMeasureWidth(const Value: string;
                               ACanvas: TCanvas;
                               var AWidth: Integer);
    procedure ListMeasureHeight(const Value: string;
                                ACanvas: TCanvas;
                                var AHeight: Integer);
    procedure ListDrawValue(const Value: string;
                            ACanvas: TCanvas;
                            const ARect: TRect; ASelected: Boolean);
  end;

procedure Register;

implementation
uses typinfo;

var
 TrueBmp : TBitmap;
 FalseBmp: TBitmap;


procedure TGraphBooleanProperty.ListDrawValue(const Value: string;
        ACanvas: TCanvas; const ARect: TRect; ASelected: Boolean);
var
  X,Y,H,W: Integer;
  BordDroit: integer;
  ValeurBooleenne: boolean;
begin
  // calculer le bord pour laisser la place de l'image
  BordDroit := (ARect.Bottom - ARect.Top) + ARect.Left;
  with ACanvas do
  try
    ValeurBooleenne := Boolean(GetEnumValue(GetPropType, Value));
    // limites de l'image à dessiner
    X := ARect.Left + 1;
    Y := ARect.Top + 1;
    W := BordDroit - ARect.Left - 1;
    H := ARect.Bottom - ARect.Top - 1;
            if ValeurBooleenne then
               StretchDraw(Rect(X, Y, X+W, Y+H), Truebmp) else
               StretchDraw(Rect(X, Y, X+W, Y+H), FalseBmp);
  finally
           // dessiner le texte
           acanvas.TextRect(Rect(BordDroit, ARect.Top,
                                 ARect.Right, ARect.Bottom),
                                 BordDroit,ARect.Top,Value);
      end;
end;

procedure TGraphBooleanProperty.ListMeasureWidth(
                    const Value: string;
                    ACanvas: TCanvas; var AWidth: Integer);
```

```
begin
  AWidth := Awidth + Acanvas.TextWidth('W')+1;  // largeur pour l'image
end;

procedure TGraphBooleanProperty.ListMeasureHeight(
                 const Value: string; ACanvas: TCanvas;
                 var AHeight: Integer);
begin
 AHeight := Acanvas.TextHeight('A')+ 1;  // hauteur d'une ligne
end;


// enregistrement de l'éditeur
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(Boolean), nil, '',
                                  TGraphBooleanProperty);
end;  { Register }

initialization
 // chargement des bitmaps
  Truebmp := TBitmap.Create;
  FalseBmp := TBitmap.Create;
  TrueBmp.Handle := LoadBitmap(Hinstance,'TRUE');
  FalseBmp.Handle := LoadBitmap(HInstance, 'FALSE');
finalization
 // libération des images
  Truebmp.Free;
  FalseBmp.Free;
end.
```

L'éditeur qui est défini est représenté par la classe `TGraphBooleanProperty`. Elle hérite de `TEnumProperty` qui est l'éditeur des propriétés de type booléen dans Delphi. Comme on peut le voir, la classe implémente aussi l'interface `ICustomPropertyListDrawing` définie dans `VCLEditors`. La classe de l'éditeur ne fait rien d'autre qu'implémenter les procédures de cette interface puisqu'en héritant de `TEnumProperty` l'essentiel du travail est déjà effectué.

L'interface définit trois procédures qu'il faut surcharger :

```
procedure ListMeasureWidth(const Value: string; ACanvas: TCanvas;
      var AWidth: Integer);
procedure ListMeasureHeight(const Value: string; ACanvas: TCanvas;
      var AHeight: Integer);
procedure ListDrawValue(const Value: string; ACanvas: TCanvas;
      const ARect: TRect; ASelected: Boolean);
```

Les deux premières sont appelées par l'inspecteur d'objet pour connaître la hauteur et la largeur de chaque élément de la liste déroulante. On peut fort bien ne rien faire de spécial dans le code, puisque l'inspecteur d'objet s'occupe de passer une valeur qui

convient *a priori*. Dans le cas où les items auraient des tailles réellement imprévisibles (des images de tailles différentes, des polices différentes…), il conviendrait de fournir une valeur exacte. Si la taille dépend du texte, on utilise la valeur `Value` qui doit être affichée et qui est passée en paramètre.

La troisième procédure est celle qui est appelée pour chaque item à dessiner dans la liste. L'inspecteur d'objet passe la valeur à afficher, une référence du `canvas` à utiliser, le rectangle dans lequel l'affichage doit être circonscrit et un drapeau `ASelected` qui indique si l'item est sélectionné ou non.

La dernière chose que fait le code de l'éditeur, c'est de définir une procédure `Register` qui appelle `RegisterPropertyEditor`. Les paramètres ont été présentés dans la partie sur les *Tools API* en introduction de cet article. Ceux qui sont utilisés ici enregistrent l'éditeur pour les propriétés de type `boolean`, quels que soient la classe et le nom de la propriété.

Le *package* qui contient l'éditeur est aussi simple que celui de notre premier exemple :

```
package BooleanEditor;

{$ALIGN 0}
{$ASSERTIONS ON}
{$BOOLEVAL OFF}
{$DEBUGINFO ON}
{$EXTENDEDSYNTAX ON}
{$IMPORTEDDATA ON}
{$IOCHECKS ON}
{$LOCALSYMBOLS ON}
{$LONGSTRINGS ON}
{$OPENSTRINGS ON}
{$OPTIMIZATION ON}
{$OVERFLOWCHECKS OFF}
{$RANGECHECKS OFF}
{$REFERENCEINFO ON}
{$SAFEDIVIDE OFF}
{$STACKFRAMES OFF}
{$TYPEDADDRESS OFF}
{$VARSTRINGCHECKS ON}
{$WRITEABLECONST OFF}
{$MINENUMSIZE 1}
{$IMAGEBASE $400000}
{$IMPLICITBUILD OFF}
{$DESCRIPTION 'Livre BDS - Editeur de propriété pour type Boolean'}
{$DESIGNONLY}


requires
  System.Drawing,
```

```
  Borland.Studio.Vcl.Design,
  System,
  System.Data,
  System.XML,
  Borland.Delphi,
  Borland.Vcl,
  Borland.VclRtl;

contains
  booledit in 'booledit.pas';

end.
```

On retrouve le drapeau de compilation `DESIGNONLY`, qui indique qu'il s'agit d'un *package* de *design*, ainsi que la référence à l'espace de nom des outils de *design* dans la clause `Requires`.

Notez que notre éditeur fait usage de deux petites images `BMP` 16x16 stockées dans un fichier de ressource de même nom que celui de l'unité contenant l'éditeur (avec l'extension `RES`).

Pour conclure sur les éditeurs de propriété

Les éditeurs de propriété sont indispensables au bon fonctionnement d'un composant. Un composant qui définit une propriété pour saisir un nom de fichier et qui, dans l'inspecteur, ne propose pas de moyen d'ouvrir une boîte de dialogue pour aller chercher ce nom sur les disques n'est pas utilisable. Il est essentiel de bien comprendre ce point, qui dépasse la simple raison et le sens de l'ergonomie : *les éditeurs de propriété font partie intégrante du développement d'un composant*.

Il y a même des cas où les éditeurs de propriété deviennent des outils indispensables. Cela est vrai dès que les propriétés sont complexes, c'est-à-dire non fondées sur des types de base (ordinaux en général). La VCL nous fournit de nombreux exemples : les colonnes d'un `TDBGrid`, les champs d'un `TDataset` (et dérivés), les sous-panneaux d'un `TStatusBar`, autant de composants qui ne seraient pas utilisables en pratique si certaines de leurs propriétés ne disposaient pas d'éditeurs spécialisés.

En dehors de rendre un composant agréable et pratique, les éditeurs de propriété remplissent aussi un rôle fonctionnel lorsque les propriétés ne sont pas des ordinaux, des objets ou des chaînes. Sachez les utiliser pour développer des composants performants.

Les catégories de propriété

L'inspecteur d'objet permet d'afficher les propriétés selon leur catégorie au lieu de l'ordre alphabétique traditionnel. Cela est géré par un système d'enregistrement se trouvant dans `Borland.Vcl.Design.DesignIntf`, qui se matérialise par une procédure existant en différentes versions (`overload`) :

```
procedure RegisterPropertyInCategory(const CategoryName, PropertyName: string);
   overload;
procedure RegisterPropertyInCategory(const CategoryName: string;
  ComponentClass: TClass; const PropertyName: string); overload;
procedure RegisterPropertyInCategory(const CategoryName: string;
  PropertyType: PTypeInfo; const PropertyName: string); overload;
procedure RegisterPropertyInCategory(const CategoryName: string;
  PropertyType: PTypeInfo); overload;
```

Selon la version choisie, il est possible d'enregistrer soit un nom de propriété dans une catégorie, soit un nom de propriété dans un composant précis, soit un type de propriété avec un nom précis, soit un type seul.

Il existe un second jeu de procédures de même nom, à la différence de l'utilisation du pluriel « `properties` » au lieu de « `property` », qui permet les mêmes enregistrements mais pour un ensemble de propriétés au lieu d'une seule.

Pour créer une nouvelle catégorie, les choses sont très simples puisqu'il suffit de donner un nom qui n'existe pas encore. Une propriété peut être définie dans plusieurs catégories.

L'utilisation de la méthode d'enregistrement est illustrée par le code suivant :

```
unit PropCat;

interface

uses
  Borland.Vcl.Design.DesignIntf,
  Sysutils;


procedure Register;

implementation


procedure Register;
begin
RegisterPropertyInCategory('LivreBDS','Caption');
```

```
RegisterPropertyInCategory('LivreBDS','Ctl3D');
RegisterPropertyInCategory('LivreBDS','Hint');
RegisterPropertyInCategory('LivreBDS','A');
end;

end.
```

Ce code doit ensuite être intégré dans un *package* de *design* identique à ceux donnés en exemple plus haut pour les éditeurs de propriété.

Son effet est visible dès lors qu'on sélectionne le mode d'affichage par catégorie dans l'inspecteur d'objet : toutes les propriétés ayant pour nom `Caption`, `Ctl3D`, `Hint`, et toutes celles contenant un `A` se retrouvent dans la catégorie `LivreBDS`. Une petite image valant mieux qu'un long discours, regardons l'effet avant et après l'installation du `package` dans l'IDE (voir figure 30.4).

Astuce

Les noms de propriétés ne sont pas comparés par une égalité, mais par inclusion. Cela signifie que lorsqu'on enregistre une catégorie pour la propriété `A` (qui n'existe pas), l'inspecteur d'objet inclut toutes les propriétés contenant un `A`. Il en est de même pour `Hint` (`ShowHint` sera aussi sélectionnée). Ainsi, quand vous définissez une catégorie de cette façon, faites attention à ce que le mot utilisé ne se retrouve pas dans d'autres propriétés…

Figure 30.4

L'enregistrement de la catégorie LivreBDS

Les éditeurs de composant

Les éditeurs de composant sont à la fois plus simples et plus complexes que les éditeurs de propriété. Ils sont plus simples car ils entrent moins en interaction avec l'IDE que ne le font les éditeurs de propriété, qui peuvent aller jusqu'à modifier le comportement de la fenêtre de l'inspecteur d'objet. Ils sont en même temps plus complexes car, comme nous l'avons vu, les éditeurs de composant proposent le plus généralement des dialogues complets qui réclament plus de développement. Il reste toutefois possible de créer un éditeur de composant en quelques lignes.

Définition

Techniquement, un éditeur de composant est un morceau de code qui répond à deux événements précis intervenant dans l'IDE lorsqu'une fiche est en mode conception : le double-clic sur l'un des composants posés sur la fiche et l'affichage du menu contextuel de ce même composant. L'une ou l'autre des deux fonctions peut être implémentée ou non, les deux le sont souvent.

Un éditeur de composant est créé en partant de la classe `TComponentEditor`, qui implémente déjà toutes les méthodes de l'interface `IComponentEditor` en partant de la base `TBaseComponentEditor`.

Au travers d'un exemple simple mais fonctionnel, nous allons étudier la stratégie de développement d'un tel éditeur.

Un éditeur de composant complet

Pour créer un éditeur de composant, il faut… un composant à éditer ! En général, il s'agira d'un composant que vous avez créé vous-mêmes et que vous compléterez d'éditeurs de propriétés et d'un éditeur de composant. Cependant, comme nous allons le voir, il est aussi possible de créer un éditeur pour un composant déjà existant, même si on ne dispose pas de son code source et même s'il existait déjà un éditeur défini.

Afin de ne pas encombrer l'exemple d'un code inutile à la compréhension de la conception de l'éditeur lui-même, nous avons

choisi le composant `TShape`. Il possède peu de propriétés-clés et son apparence graphique se prête bien à une démonstration visuelle mettant en évidence l'intérêt d'un éditeur de composant. Avec un composant aussi simple et connu de tous, nous éviterons aussi d'avoir à d'abord expliquer ce que fait ce dernier.

Ce que nous allons faire

Nous allons développer un éditeur de composant pour la classe `TShape`. Cet éditeur proposera un dialogue complet d'édition des principales propriétés du composant en fournissant une simulation visuelle des modifications apportées. De plus, nous ajouterons trois entrées dans le menu contextuel du composant. Le dialogue sera automatiquement activé par un double-clic sur le composant posé sur un `TForm` en édition, les entrées du menu contextuel apparaîtront après un clic droit sur le composant en édition.

Comment nous allons procéder : stratégie de développement

Développer un éditeur de composant qui possède un dialogue d'édition réclame avant tout de développer ce dialogue et de le tester au sein d'une application, ce qui évite d'avoir à installer le *package* pour le désinstaller ensuite et le corriger dans une série de cycles itératifs devenant rapidement lourds.

De plus, un dialogue d'édition de composant peut fort bien devenir intéressant à l'exécution. Hélas, la séparation du code imposée par Delphi (à juste titre par ailleurs) entre le code de conception et celui d'exécution fait que, si nous développons le dialogue comme partie intégrante de l'éditeur de composant, ce dialogue ne sera pas réutilisable en dehors de l'IDE de Delphi.

Pour satisfaire ces deux objectifs (pouvoir tester simplement le dialogue et pouvoir le réutiliser dans une application), nous allons développer une fiche indépendante qui exportera une procédure permettant d'invoquer le dialogue. Cette procédure prendra en paramètre l'instance du composant `TShape` à modifier.

De ce fait, nous construirons le dialogue d'édition au sein d'une petite application de test, ce qui satisfera le premier objectif : la simplicité de développement.

Une fois le dialogue bien testé, il nous suffira alors de créer un véritable éditeur de composant qui ne fera qu'appeler notre dialogue. Ce dernier reste parfaitement utilisable dans une application classique, puisque non dépendant de l'éditeur de

composant (nous créons la situation exactement inverse en fait : l'éditeur dépend du dialogue). Nous satisferons ainsi le second objectif : réutilisation possible de l'éditeur dans une application.

Le dialogue d'édition

Pour ne par alourdir l'exemple, nous avons choisi de sélectionner les propriétés les plus importantes du `TShape`, à savoir la brosse, le crayon, le type de forme, le `hint` et la taille de la forme. Concernant le style de brosse et de crayon, dans le même souci de simplification, nous n'avons retenu que deux possibilités : couleur pleine ou transparence. Les autres options, comme les hachures pour la brosse ou les pointillés pour le crayon, sont volontairement omises. Libre à vous de compléter l'éditeur en ajoutant la prise en charge de ces options, ainsi que des autres propriétés délaissées par notre exemple.

Le code complet de l'exemple étant disponible sur le CD-Rom du livre, nous vous éviterons les parties les moins significatives de ce dernier, notamment tout ce qui concerne les déclarations du `TForm` ou la totalité des événements gérés par la fiche. Pour vous aider à suivre ce texte, vous trouverez ci-dessous un extrait du code de la fiche du dialogue. Les parties sautées sont symbolisées par trois étoiles.

```
unit uCompEdit;

* * *

type
  TFEditTShape = class(TForm)      // La classe de notre fiche dialogue
* * *
  private
    Updating : boolean;        // drapeau de mise à jour
    fSCName:String;            // le nom du composant en cours d'édition
    procedure CenterShape;     // recentre la simulation dans son espace visuel
    Procedure UpdateDisplay;   // met à jour l'affichage des propriétés
  public
    Procedure InitShape(UneForme:TShape); // initialisation du dialogue
    procedure CopyProps(Source,Dest:Tshape);   // copie d'un tshape sur un autre
    property ShapeComponentName : string
                                  read fSCName write fSCName;
  end;

// La procédure à appeler pour modifier un TShape
Procedure EditTShape(LaForme:TShape);

implementation
* * *
// exemple de méthode définie pour gérer le dialogue
// Couleur de la brosse
```

```
procedure TFEditTShape.ColorBox1Change(Sender: TObject);
begin
 If updating then exit; // si mode de mise à jour on ne fait rien
 Shape1.Brush.Color := Colorbox1.Selected; // changement de couleur brosse
end;
* * *

// La procédure d'édition à appeler de l'extérieur
Procedure EditTShape(LaForme:TShape);
begin
 with TFEditTShape.Create(Application) do
  try
    ShapeComponentName := LaForme.Name;
    InitShape(LaForme);
    If ShowModal=mrOk then CopyProps(Shape1,LaForme);
  finally
   Release;
  end;
end;

end.
```

Dans ce code, nous définissons une fiche qui comporte des boîtes de sélection de couleurs, une `listbox` pour sélectionner le type de forme, des `trackbars` pour changer la hauteur et la largeur de la forme, ainsi que des boutons radio et d'autres composants. Tout cela est de la programmation classique sans grand intérêt dans le cadre présent.

Ce qu'il faut remarquer dans ce code, c'est la procédure `EditTShape`, qui est définie dans l'interface de l'unité et dont le code se trouve en fin de celle-ci. Son principe est simple : elle crée une instance du dialogue, elle initialise ce dernier en passant le nom du composant à éditer et copie les informations de celui-ci dans la simulation (appel à `InitShape`). Ensuite, le dialogue est appelé en *modal* et, s'il est validé (`modalresult = mrOk`), alors le composant à éditer voit ses propriétés recopiées depuis la simulation se trouvant dans le dialogue (appel de `CopyProps`, procédure définie dans l'unité et recopiant les propriétés d'un `TShape` sur un autre, la méthode `Assign` ne fonctionnant pas dans le `TShape`).

Vous pouvez voir l'aspect visuel du dialogue lui-même dans les captures d'écran qui vont suivre.

L'application de test

Il s'agit d'une application minimaliste : sur un `TForm`, nous avons posé un composant `TShape` et un bouton permettant d'appeler notre dialogue. Cette application utilise (`Uses`) la fiche du

dialogue présentée ci-dessus. Le code du bouton se borne à un appel à `EditTShape(Shape1)`, `Shape1` étant le `TShape` posé sur la fiche de l'application de test.

Le code étant d'une simplicité extrême et se trouvant lui aussi dans le CD-Rom du livre, nous passerons immédiatement à la suite.

L'éditeur de composant proprement dit

Nous voici arrivés au cœur de notre étude, l'écriture de l'éditeur de composant lui-même. Ayant adopté un découplage fort entre la partie dialogue et la partie éditeur, le code de ce dernier va être grandement simplifié et sa mise au point ne posera aucun problème, le plus délicat étant déjà testé.

Voici le code de l'éditeur :

```
{-----------------------------------------------------------------
 Unit Name: uTshapeEditor
 Author:    od
 Purpose:   Editeur de composant Tshape - Livre BDS
 History:   juil 2005 - V 2.1 - .NET
-----------------------------------------------------------------}



unit uTshapeEditor;

interface
uses Borland.Vcl.Design.DesignIntf,      // les unités de design
     Borland.Vcl.Design.DesignEditors,
     Borland.Vcl.Design.DesignMenus,
     ExtCtrls,        // définition du TShape
     dialogs,         // pour le showmessage
     uCompEdit;       // le dialogue d'édition du composant

// L'éditeur de composant
Type TShapeComponentEditor = Class (TComponentEditor)
    private
     fdejavu : boolean;
    public
    procedure ExecuteVerb(Index: Integer); override;
    function GetVerb(Index: Integer): string; override;
    function GetVerbCount: Integer; override;
    procedure PrepareItem(Index: Integer; const AItem: IMenuItem);
              override;
    procedure Edit; override;
    end;

Procedure Register;

implementation
```

```
// enregistrement de l'éditeur
Procedure Register;
begin
 RegisterComponentEditor(TShape,TShapeComponentEditor);
end;

// exécution d'une des entrées du menu contextuel
procedure TShapeComponentEditor.ExecuteVerb(Index: Integer);
begin
 case Index of
 0: begin
      Showmessage('Vous avez sélectionné le composant '+
                          Tshape(Component).Name);
      fdejavu := true;
    end;
 1: Showmessage('© 2004 Olivier Dahan'#13#10'odahan@e-naxos.com'+
                          #13#10'Livre BDS Eyrolles');
 2: Edit;
 end;
end;

// Retourne les libellés du menu contextuel
function TShapeComponentEditor.GetVerb(Index: Integer): string;
begin
 case Index of
 0 : Result := 'Nom du composant [exemple]';
 1 : Result := 'A propos de l''éditeur [exemple]';
 2 : Result := 'Editer le composant [exemple]';
 end;
end;

// retourne le nombre d'éléments du menu contextuel
function TShapeComponentEditor.GetVerbCount: Integer;
begin
 Result := 3;
end;

// Est appelé avant l'affichage de chaque item du menu
procedure TShapeComponentEditor.PrepareItem(Index: Integer;
                                    const AItem: IMenuItem);
begin
// on cochera l'item 0 s'il a déjà été cliqué dans cette
// session d'édition
AItem.Checked := (Index=0) and fdejavu;
end;

// double clic sur le composant
procedure TShapeComponentEditor.Edit;
begin
 // appel de notre dialogue
 EditTShape(Tshape(Component));
 // on avertit le designer de la modification
 Designer.Modified;
```

```
end;

end.
```

Comme vous le constatez, ce code est assez compact et ne présente pas de difficulté majeure. Voyons dès lors les points clés qui doivent retenir votre attention.

La première chose est la définition de la classe de l'éditeur :

```
TShapeComponentEditor = Class (TComponentEditor)
```

La classe hérite de `TComponentEditor`, qui hérite elle-même de `TBaseComponentEditor` et implémente un service minimum pour l'interface `IComponentEditor`.

Ensuite, cinq méthodes sont redéfinies. Elles concernent pour quatre d'entre elles la gestion du menu contextuel. `GetVerbCount` renvoie le nombre d'éléments que nous ajoutons au menu (ici trois), `GetVerb` retourne, en fonction de l'index, le texte de chacun des items ajouté au menu et `ExecuteVerb` doit exécuter le code qui correspond à l'item dont le numéro est passé en paramètre (c'est en quelque sorte un `OnClick` centralisé pour les entrées ajoutées au menu). `PrepareItem` n'est généralement pas nécessaire ; nous l'avons utilisé afin de vous faire voir cette possibilité. Cette méthode est appelée avant l'affichage de chaque item du menu et l'interface de ce dernier est passée en paramètre. Cela permet par exemple d'ajouter des sous-menus ou de désactiver une entrée. Ici, nous l'utilisons pour marquer d'une croix la première entrée, dès lors qu'elle a été déjà visitée.

La cinquième méthode est `Edit`, qui est appelée par l'EDI lors du double-clic sur le composant. Vous remarquerez que la troisième entrée du menu contextuel que nous avons ajoutée permet aussi d'appeler cette méthode. Un tel rappel est ergonomique, il évite à l'utilisateur de se demander s'il faut essayer le double-clic pour voir si quelque chose de particulier se passe.

Enfin, l'unité possède une procédure `Register` qui appelle `RegisterComponentEditor`, dont les paramètres sont clairs : en premier la classe pour laquelle l'éditeur est installé et en second la classe de l'éditeur installé.

Le code de l'éditeur est lui-même inclus dans un *package* de *design* qui a la même forme que tous ceux présentés dans cet article. Nous n'en donnerons donc pas le code ici.

L'éditeur en action

Une fois l'éditeur installé, reste la récompense : le voir fonctionner correctement.

C'est ce que nous allons faire en ouvrant un nouveau projet et en posant un composant `TShape` sur la fiche. Roulements de tambour… (voir figure 30.5, désolé pour le son, vous n'aurez que l'image !).

Figure 30.5

Le menu contextuel du composant affiche nos options

La figure 30.5 montre l'effet du clic droit sur le composant `Shape1` posé sur la fiche `Form1`. Les trois premières options sont bien celles de l'éditeur que nous venons d'installer dans l'EDI. La première option devrait afficher le nom du composant, la seconde une petite boîte « à propos » et la troisième devrait invoquer le dialogue, ce qui sera équivalent à un double-clic sur le composant.

Figure 30.6

L'effet de la première option

La figure 30.6 montre l'effet de la première option. En effet, c'est bien le composant nommé `Shape1` qui est sélectionné dans l'EDI.

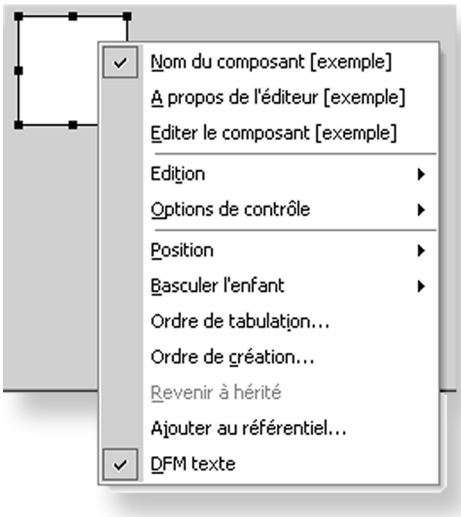

Figure 30.7

L'effet de PrepareItem

Comme le montre la figure 30.7, puisque nous avons déjà appelé l'option 1 du menu contextuel, celle-ci s'affiche maintenant précédée d'une coche verte. C'est ici l'effet de la méthode `PrepareItem` qui devient visible.

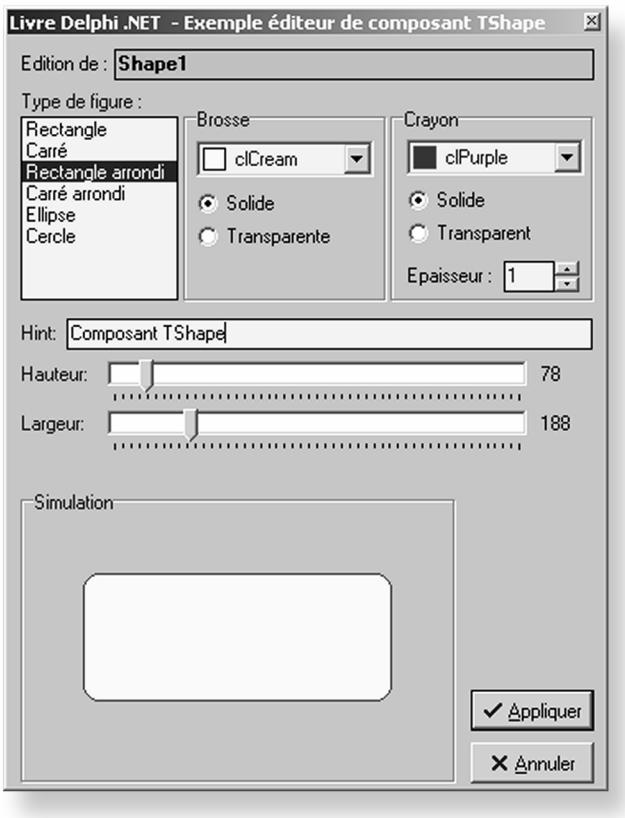

Figure 30.8

Le dialogue d'édition du TShape

Il ne reste plus qu'à sélectionner la troisième option du menu contextuel, puis à réessayer en double-cliquant sur le composant pour nous assurer que le dialogue est bien affiché dans les deux cas. La figure 30.8 nous montre le dialogue en pleine action, mission accomplie.

Conclusion

Nous avons vu ici comment créer des éditeurs de propriétés, comment créer de nouvelles catégories de propriétés et comment enregistrer certaines propriétés dans ces catégories. Enfin, nous avons étudié la conception d'un éditeur de composant en présentant la stratégie de développement la plus efficace pour ce faire.

Il ne vous reste plus qu'à développer de nombreux composants, ne serait-ce que pour le plaisir de créer des éditeurs de propriétés et de composants !