Persistence frameworks: writing GUIs in tiOPF

Michaël Van Canneyt

May 27, 2007

Abstract

In this second article about the Object Persistence framework tiOPF, the GUI layer of the tiOPF framework is investigated. It will be shown how to use the persistence-aware controls of tiOPF.

1 Introduction

In a previous article, the architecture of the persistence framework tiOPF was investigated. In it, the 3 corner objects in the tiOPF framework were explained: the base object `TtiObject`, from which all objects in the model should descend; the basic list object `TtiObjectList`, and the `TtiVisitor`, which will be needed to save the data.

In this article, the use of these 3 objects to create a GUI application will be explained, and a rudimentary overview of the available persistence-aware controls will be given.

The GUI controls are located on 2 palettes in the component palette:

Techinsite Base These are the basic edit, combobox, checkbox, memo controls, in a persistence-aware version. They resemble the standard data-aware controls of Delphi, but there are some subtle differences. The persistence aware in this case means that the controls know how to retrieve, display and set a published property of an object. (much as the standard RTTI controls in Lazarus). The controls do not mark the object as 'dirty', this must be done manually.

Techinsite Extra Here more advanced controls are located. There are many utility controls (splitters, speedbuttons, rounded panels) but there are also 2 basic controls: `TtiVTListView` and `TtiVTTreeView`. These are quite advanced controls, and quite indispensable for creating list views of the objects in the application.

The reader is warned that the available documentation of the controls in the framework itself is rather out-of-date: some of the controls have radically changed. The old controls, as explained in the available documentation on the website are still available on the component palette under the 'techinsite old' page.

2 The model

Before a start can be made, the business model must be programmed: this means that the following classes must be programmed:

TCountry which represents a country. It has 2 published string properties: `ISO` and Name.

TCity which represents a city in a country. It has 3 published properties: `Name`, `Zip` (both strings) and `Country`, which is a reference to a `TCountry` instance.

TAddressType which has a single property 'Name', represents the type of an address.

TAddress represents an address of a contact person. The obvious string properties are `Street`, `Nr`, Telephone1, Telephone2, followed by the `AddressType` and `City` properties, both references to an instance of an object of type `TAddressType` and `TCity`, respectively.

TContact represents a contact person. It has 5 published string properties: FirstName, LastName, Email, mobile, Comments. It also has a property `Addresses`, which is a `TTiObjectList` descendent. More about this list below.

Next to the basic objects, a set of lists must be created:

TCountries A list of all countries.

TCities A list of all cities.

TAddresses a (general) list of addresses.

TContactAddresses is a descendent of `TAddresses`, which will be owned by a `TContact` instance. It will represent the addresses for that contact.

TContacts a list of all contacts in the application.

For all lists will be needed in the application, a list object should be created.

It's good practice to create, for each project, a descendent of the basic `TtiObject` and `TtiObjectList` classes: in these classes, some logic, common to all objects in the application. For the purpose of the contacts application, a single method will be implemented, `Mark`, and the basic objects will be called `TMarkObject` and `TMarkObjectList`:

```
Type
TMarkObject = Class(TTiObject)
Protected
  Procedure Mark; // Set to dirty;
end;

TMarkObjectList = Class(TTiObjectList)
Protected
  Procedure Mark; // Set to dirty;
end;
```

The `Mark` procedure simply sets the `Dirty` property:

```
procedure TMarkObject.Mark;
begin
  if (ObjectState<>posEmpty) then
    Dirty:=True;
end;
```

These 2 classes will be the base of all objects in the application:

```
TCountry = class(TMarkObject)
private
```

```
  FISO: string;
  FName: string;
  procedure SetISO(const Value: string);
  procedure SetName(const Value: string);
published
  property ISO: string read FISO write SetISO;
  property Name: string read FName write SetName;
end;
```

The `SetISO` and `SetName` properties are set using a Write procedure:

```
procedure TCountry.SetISO(const Value: string);
begin
  FISO:=Value;
  Mark;
end;
```

Which sets the 'Dirty' property of the object. This means that as soon as a property is set, the object is marked as dirty. As a result, as soon as an edit control sets a property, the object is marked as dirty, and will be saved when needed.

The `TCity` class contains a reference to a country:

```
TCity = Class(TMarkObject)
private
  FZIP: string;
  FName: string;
  FCountry: TCountry;
  procedure SetCountry(const Value: TCountry);
  procedure SetName(const Value: string);
  procedure SetZIP(const Value: string);
published
  property Country: TCountry read FCountry write SetCountry;
  property Name: string read FName write SetName;
  property ZIP: string read FZIP write SetZIP;
end;
```

The country property is a class property, it contains an instance. However, the instance is not owned by the `TCity` class, it exists as a separate entity. Therefore, the `SetCountry` simply stores the `TCountry` pointer:

```
procedure TCity.SetCountry(const Value: TCountry);
begin
  FCountry := Value;
  Mark;
end;
```

The `TAddress` type has a similar reference to a `TCity`. The `AddressType` property is an instance of `TCity`, but in contrast it is owned by the address. This means that an instance must be created when the `TAddress` record is created, and that when the property is set, special measures must be taken. The relevant parts of the `TAddress` record are declared as follows:

```
TAddress = Class(TMarkObject)
private
```

```
  Procedure SetAddressType(const Value: TAddressType);
Public
  constructor Create ; override;
  Destructor Destroy; override;
  procedure   AssignClassProps(pSource: TtiObject); override;
published
  property AddressType: TAddressType read FAddressType
                                     write SetAddressType;
end;
```

With the following implementation for the methods:

```
constructor TAddress.Create;
begin
  inherited;
  FAddressType:=TAddressType.Create;
  FAddressType.Owner:=Self;
end;

destructor TAddress.Destroy;
begin
  FreeAndNil(FAddressType);
  inherited;
end;
```

And specially

```
procedure TAddress.SetAddressType(const Value: TAddressType);
begin
  FaddressType.Assign(pSource);
  Mark;
end;

procedure TAddress.AssignClassProps(pSource: TtiObject);
begin
  inherited;
  FaddressType.Assign(pSource);
  FCity:=TAddress(pSource).City;
end;
```

The `AssignClassProps` is called when the tiOPF framework handles the `Assign` method, to assign class properties.

For each class, a list class is defined. For the `TCountry` class, this list is defined as:

```
TCountries = class(TMarkObjectList)
private
  function GetItems(i: integer): TCountry;
  procedure SetItems(i: integer; const Value: TCountry);
public
  procedure Add(AnAddress : TCountry) ; reintroduce;
  property Items[i:integer] : TCountry
           read GetItems write SetItems ; default;
end;
```

The implementation is quite simple and straightforward.

For the `TAddress` type, a similar list, `TAddresses`, exists, plus a descendent: `TContactAddresses` class, which is declared as:

```
TContactAddresses = Class(TAddresses)
private
  function GetContact: TContact;
  procedure SetContact(const Value: TContact);
public
  property Owner : TContact read GetContact  write SetContact;
end;
```

The `Owner` property is used in the `TContact` class, when creating an instance of `TContactAddresses` to hold the list of addresses:

```
constructor TContact.Create;
begin
  inherited;
  FAddresses:=TContactAddresses.Create;
  FAddresses.Owner:=Self;
  FAddresses.ItemOwner:=Self;
end;
```

Setting the `ItemOwner` means that when the object hierarchy is traversed, the owner of an address is the contact instance to whom the address belongs.

All these classes are implemented in the `ContactModel` unit.

3 The manager class

To make saving and loading easier, a single class which owns all global lists is implemented

```
TContactManager = Class(TMarkObject)
Public
  Constructor Create; override;
  Procedure PopulateContacts;
Published
  Property Countries : TCountries Read FCountries;
  Property Cities : TCities Read FCities;
  Property Contacts : TContacts Read FContacts;
end;
```

A single instance of this class (`ContactManager`)is created and maintained in the `mgrContacts` unit: the instance is created in the initialization section, and destroyed in the finalization section. Note that the `Countries`, `Cities` and `Contacts` properties are lists classes as defined in the model.

When data needs to be saved, the `ContactManager` class can be saved: all lists will be saved in turn by the tiOPF framework, because they are published properties of the `TContactManager` class, which in itself is a `TMarkObject` descendent.

The `PopulateContacts` method creates some testdata for the application. It's instructive to have a look at this, because it shows how to create new data:

```
procedure TContactManager.PopulateContacts;

Var
  C : TContact;
  I,J : Integer;
  A : TAddress;

begin
  Randomize;
  PopulateCountries;
  PopulateCities;
  For I:=1 to 10 do
    begin
    C:=TContact.CreateNew;
    C.FirstName:=FirstNames[I];
    C.LastName:=LastNames[I];
    C.Mobile:=GenPhone;
    C.Email:=FirstNames[i]+'@freepascal.org';
    For J:=1 to 1+Random(2) do
      begin
      A:=TAddress.CreateNew;
      A.Street:=StreetNames[1+Random(10)];
      A.Nr:=IntToStr(Random(100)+1);
      A.City:=FCities[Random(10)];
      A.Fax:=GenPhone;
      A.Telephone1:=GenPhone;
      If Random(2)>0 then
        A.Telephone2:=GenPhone;
      C.Adresses.Add(A);
      end;
    FContacts.Add(C);
    end;
end;
```

First off, the countries and cities lists are populated, using methods similar to the one shown here. After that, 10 contact persons are generated: their first and last names are taken from 2 constants, `FirstNames` and `LastNames`, both of which are arrays of strings.

The `genphone` is an algorithm which generates a random telephone number. The `StreetNames` is again a constant array of string, containing 10 names of streets. This way, for each contact 1 or 2 addresses can be created. The generated addresses are simply added to the `Addresses` list of a contact, and the created contact is added to the global list of contacts (`FContacts`) which is owned by the `ContactsManager`.

Note that the `CreateNew` constructor is called, and not `Create`: Calling `CreateNew` tells the tiOPF framework that the instance is new, and does not yet exist in the database. The framework will then allocate a new unique identifier for the object instance. More about this below. If `Create` is called, then the framework will assume that it will be loaded with data from the database.

The `PopulateCountries` and `PopulateCities` work in a similar manner, and store their data in the global `Countries` and `Cities` lists.

4 The main form

The main form consists of a menu bar, an action list, and a `TtiVTListView` instance (`LVContacts`). The `TtiVTListView` component is one of the most important GUI components of the `tiOPF` framework. It is the equivalent of the `TDBGrid` in delpi's list of data-aware controls. It allows to display the contents of any `TtiObjectList` list, and can do more:

1. There is a full-text search function (activated with Ctrl-F), which searches through all properties in the list. The `Searching` boolean property controls the availability of this feature.
2. The rows in the list can be shown in alternating colors, which is controlled by the `ShowAlternateRowColor`, `AlternateRowColors` and `AlternateRowCount` properties.
3. The list can be ordered on any property that is shown in the columns of the listview, by simply clicking on the header of a column. This behaviour is controlled by the `HeaderClickSorting` property.
4. It has a context menu and a series of buttons to add/delete/edit/view items in the list. The list of buttons that is shown is controlled by the `VisibleButtons` property. The look of the buttons is controlled by the `ButtonStyle` property.
5. It can export it's data in a `CSV` file by default. The availability of this feature is controlled by the `Exporting` boolean propery.
6. Derived properties can be shown - this is the equivalent of a calculated field in a TDataset descendent.

All this is controlled by a large number of properties. The `Header` property is the biggest of them: It has by itself a number of properties that control the list's view. The most important one is the `Columns` property. This is a collection which – similar to the TColumns in a TDBGrid – controls which properties of the objects in the lists are shown in a column, and how they are shown:

FieldName is the name of the property

Text the caption shown above the column.

Derived If set to true, the column will contain a custom (calculated) text. The value is not directly available, but must be calculated in the `OnDeriveColumn` event.

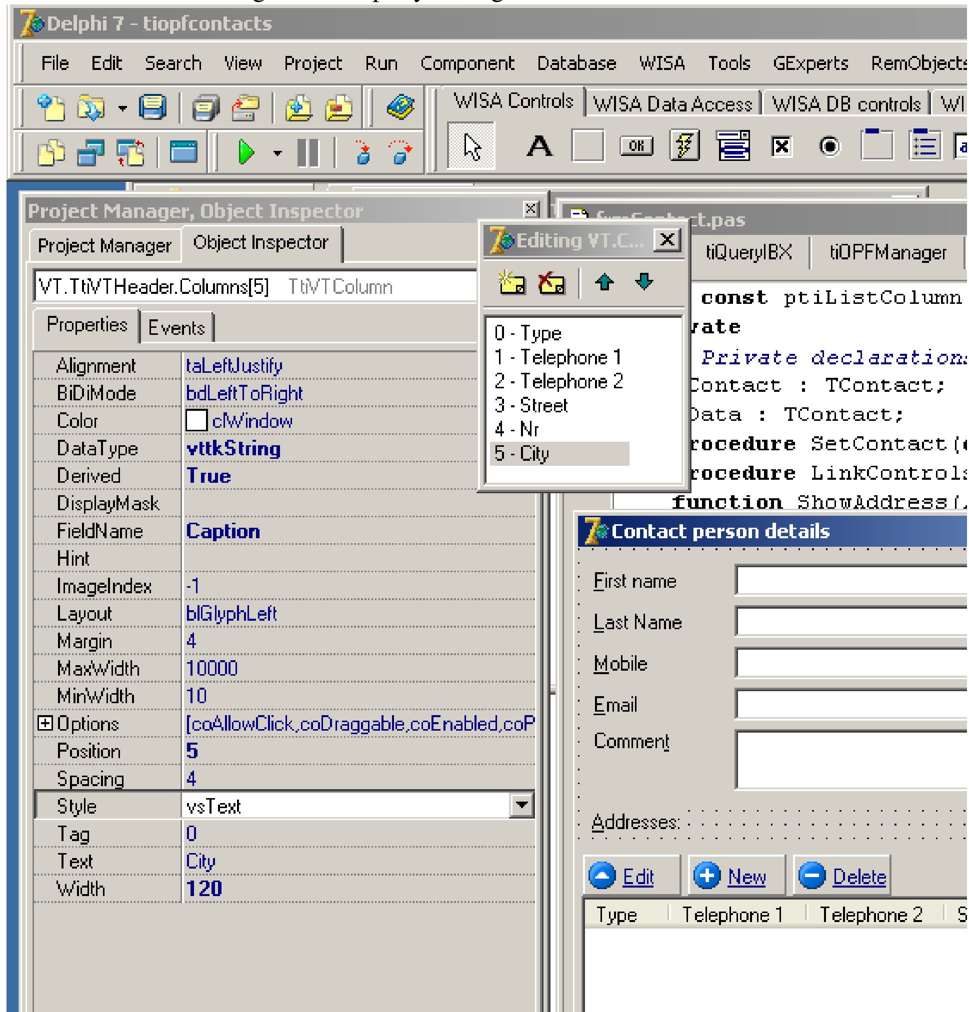
OnDeriveColumn If `Derived` is set to `True`, then the value displayed in this column must be calculated in this event.

In figure 1 on page 8 the properties for a column in an address list is shown: the column displays the name of the city, which must be calculated from the `City` instance from the address that must be displayed.

The list can also show a set of 4 buttons on top of the list: View, Edit, New and Delete. Which buttons are shown is determined by the `VisibleButtons` property of the list. In addition to this property, the event handlers for each of these actions (as explained below) must be set, or the button will not be shown.

Unlike it's `TDBGrid` cousin, the `TtiVTListview` does not know how to edit, add or delete the items in the list. This must be done using events. The relevant events are:

Figure 1: Property settings for a derived column



OnItemEdit When an item must be edited. The chosen item (a TtiObject) is passed to the event handler.

OnItemDelete When an item must be deleted.

OnItemInsert When a new item must be inserted.

OnItemView When an item must be viewed (i.e. more details should be shown than are visible in the list).

All events have the same prototype:

```
procedure TCitiesForm.LVCitiesItemInsert
          (pVT: TtiCustomVirtualTree;
           AData: TtiObject;
           AItem: PVirtualNode);
```

The first item is the instance of the list (The `OnInsert` event is introduced in `TtiCustomVirtualTree`, and `TTiVTListView` descends from that, which explains the type of the first parameter in the event); the second parameter is the data object associated with it, and the third parameter is the actual node in the list that represents the data object.

There is no design-time property that can be set to tell the listview which list of objects must be shown. This must be done run-time. In the case of the contacts application, this is done in the `ShowContacts` method of the form:

```
procedure TMainForm.ShowContacts;
begin
  With LVContacts do
    begin
    AddColumn('LastName',vttkString,'Last name',120);
    AddColumn('Firstname',vttkString,'First name',120);
    AddColumn('Email',vttkString,'E-mail',150);
    AddColumn('Mobile',vttkString,'Mobile',100);
    AddColumn('Comment',vttkString,'Comment',200);
    Header.MainColumn:=0;
    Header.SortColumn:=0;
    Data:=ContactManager.Contacts;
    end;
end;
```

Below we'll show when this method is called.

As can be seen from the code above, the `TtiVTListview` introduces a method `AddColumn` which can be used to quickly define the columns run-time. The call has the following parameters:

AFieldName Name of the property that must be shown.

pDataType the kind of data to display (in the above case, strings)

pDisplayLabel the text for the column header (optional).

pColWidth the width of the column (optional).

The main column is the index of the main column for the list, and the sortcolumn is the initial column on which must be sorted.

Figure 2: Listview showing contact data

Contacts application in tiOPF

File Edit System

View Edit New Delete

Last name	First name	E-mail	Mobile	Comment
Michael	Michael	Michael@freepascal.org	131 957 785 7	
Florian	Florian	Florian@freepascal.org	705 342 038 94	
Daniel	Daniel	Daniel@freepascal.org	873 416 230	
Jonas	Jonas	Jonas@freepascal.org	573 513 416	
Pierre	Pierre	Pierre@freepascal.org	828 603 341	
Peter	Peter	Peter@freepascal.org	872 638 857 7	
Tomas	Tomas	Tomas@freepascal.org	182 824 070 8	
Marco	Marco	Marco@freepascal.org	050 469 227 9	
Micha	Micha	Micha@freepascal.org	302 777 043 04	
Mazen	Mazen	Mazen@freepascal.org	573 311 700 31	

Finally, the `Data` property is set to the global `ContactManager.Contacts` list. After this, the listview is ready to show data. It will look something like figure 2 on page 10

The menu contains 2 entries under the 'System' menu: 'Countries', and 'Cities'. Each shows a modal form to display the global lists of countries and cities (for maintenance). The list of cities is a good example to show how to display Derived columns.

In the `OnClick` event handme of the 'Cities' menu, the cities form (`TCitiesForm` is shown modally). The cities form contains - just as the main form - simply a `TtiVTListView` (`LVCities`), which is filled in the `OnCreate` event of the form:

```
procedure TCitiesForm.FormCreate(Sender: TObject);
begin
  LVCities.Data:=ContactManager.Cities;
end;
```

All other properties are set in the designer. The last column, `Country`, is a derived column. It's value is computed in the `OnDeriveColumn` event of the column:

```
procedure TCitiesForm.VTTtiVTHeaderColumns2DeriveColumn(
  const pVT: TtiCustomVirtualTree; const AData: TtiObject;
  const ptiListColumn: TtiVTColumn; var pResult: String);

Var C : TCity;

begin
  If (Adata is TCity) then
    begin
    C:=TCity(AData);
    If Assigned(C.Country) then
      pResult:=C.Country.ISO+' '+C.Country.Name;
    end;
end;
```

The meaning of the parameters should be obvious from the names. The important ones are:

AData is the object instance that is shown in the current row. In the example above, this will be a `TCity` instance.

pResult is the string that will be shown in the column: this is the value that must be calculated by the event handler. In the above case, `presult` will be filled with the `ISO` code and `Name` of the country, if a country is assigned to the city.

The editing events in the city listview serve as a nice example of how to edit items in a listview. In the `OnItemDelete` event, the following code will delete the item:

```
procedure TCitiesForm.LVCitiesItemDelete(
                       pVT: TtiCustomVirtualTree;
                       AData: TtiObject;
                       AItem: PVirtualNode);
begin
  if MessageDlg(SConfirmDelete,
                mtConfirmation,
                [mbYes, mbNo],0)=mrYes then
    begin
    Adata.Deleted:=True;
    ContactManager.Cities.Extract(AData);
    LVCities.Refresh;
    AData.Free;
    end;
end;
```

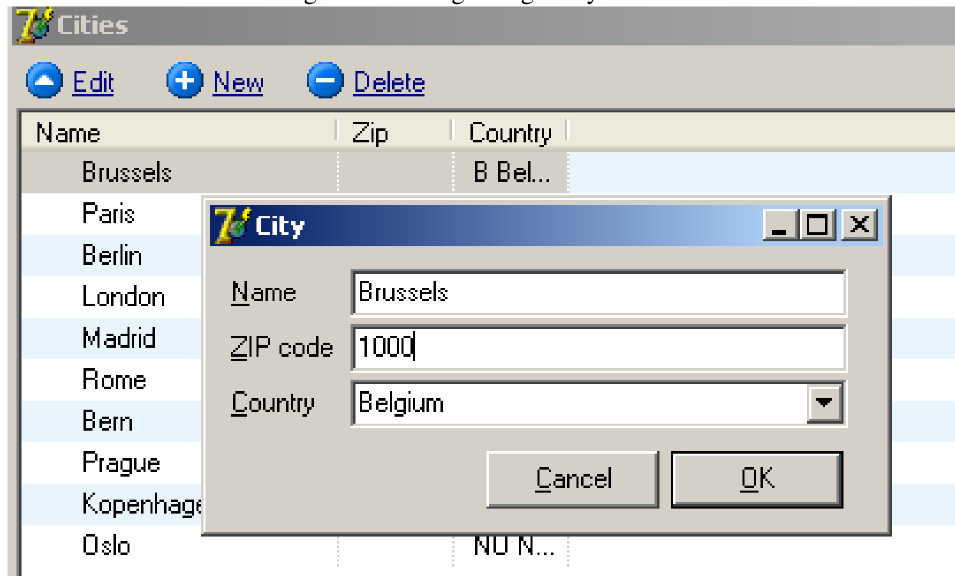
After a confirmation of the user, the `AData` instance is marked as deleted, and then extracted from the list of cities. (The `Extract` method will not free the instance, as opposed to `Remove`). After the list was updated, the listview is refreshed (this must be done manually. Only after this, the item is freed. The order of events is important, otherwise, access violations may occur.

The code for editing and adding an element resembles each other:

```
procedure TCitiesForm.LVCitiesItemEdit(pVT: TtiCustomVirtualTree;
  AData: TtiObject; AItem: PVirtualNode);
begin
  ShowCity(TCity(AData));
end;

procedure TCitiesForm.LVCitiesItemInsert(pVT: TtiCustomVirtualTree;
  AData: TtiObject; AItem: PVirtualNode);

Var
  C : TCity;

begin
  C:=TCity.Create;
  If ShowCity(c) then
    begin
    ContactManager.Cities.Add(C);
    LVCities.Refresh(C);
    end
  else
    C.Free;
end;
```

The edit event handler simply calls the `ShowCity` method. The insert method handler first creates a new `TCity` instance, passes it to `ShowCity`, and if that function returns

Figure 3: Editing a single city instance



True, the instance is added to the global list of cities, and the listview is refreshed, and positioned on the newly created city (this is the meaning of the Refresh method's optional parameter). If False is returned, the newly created instance is freed again.

The ShowCity method is quite simple, and looks as follows:

```
Function TCitiesForm.ShowCity(C : TCity) : Boolean;

begin
  With TCityForm.Create(Self) do
    try
      City:=C;
      Result:=ShowModal=mrOK;
    Finally
      Free;
    end;
end;
```

The real work is done in the TCityForm, which is a form that allows to edit a single city instance, as shown in figure 3 on page 12.

The TCityForm form demonstrates some of the edit controls that exist in the tiOPF framework. It contains 2 edit boxes (EName, EZip) of type TtoPerAwareEdit and a special combobox (CBCountry) of type TtiPerAwareComboBoxDynamic. All of them have an associated label which (much like the TLabeledEdit control in Delphi) which shows the caption for the edit control.

The CBCountry combobox is special in the sense that it will be used to display a list of country instances, and the selected country will be determined by the Country property of the TCity - also an object instance. This is handled transparently by the TtiPerAwareComboBoxDynamic control.

The FieldName property of the controls should contain the name of a published property of an object: the value of this property will then be edited by the control. It can be set at design-time in the object inspector, or it can be set runtime. The Data property must be set

at runtime to the `TtiObject` instance whose properties must be edited. Both properties can be set with a single call using the `LinkData` method. The `LinkCity` method uses this method to link a city instance to the controls in the form:

```
procedure TCityForm.LinkCity;

begin
  EName.LinkToData(FCity,'Name');
  EZip.LinkToData(FCity,'Zip');
  CBCountry.LinkToData(FCity,'Country');
end;
```

As can be seen, the first parameter to `LinkToData` is the object instance whose properties must be edited, the second parameter is the name of the property.

The class name `TtiPerAwareComboBoxDynamic` indicates that the `CBCountry` combobox must be filled dynamically with a list of objects. The list property (of type TList) must be set run-time to the list of of objects to display in the combobox. The actual texts that will be shown must be set in the 'FieldNameDisplay' property. For the `CBCountry` combobox, The list is filled in the `OnCreate` event handler of the form:

```
procedure TCityForm.FormCreate(Sender: TObject);

Var
  i : Integer;

begin
  FCity:=TCity.Create;
  FCountries:=TList.Create;
  For I:=0 to COntactManager.Countries.Count-1 do
    FCountries.Add(ContactManager.Countries[i]);
  CBCountry.List:=FCountries;
end;
```

As can be seen, the list is created, and filled with the elements in the global `Countries` list. Then it's assigned to the `List` property of the `CBCountry` combobox. The `FieldNameDisplay` is set to 'Name' in the object inspector, so the name of the country will be shown.

The list is again destroyed in the `OnDestroy` handler of the form:

```
procedure TCityForm.FormDestroy(Sender: TObject);
begin
  FreeAndNil(FCity);
  FreeAndNil(FCountries)
end;
```

In the 3 above methods, a `FCity` variable is initialized with a `TCity` instance, is linked to the controls, and is freed again when the form is destroyed. Obviously, the correct city must be shown in the form, and this will not be the `FCity` instance. Which city should be edited is determined by the `City` property, defined as follows:

```
  Property City : TCity Read FData Write SetCity;
```

The `SetCity` makes clear what the intention of the definition is:

```
procedure TCityForm.SetCity(const Value: TCity);
```

```
begin
  FData:=Value;
  FCity.Assign(Value);
  LinkCity;
end;
```

The actual `TCity` instance that must be edited is saved in `FData`. The data of this instance is then copied to the instance in `FCity` by means of the `Assign` call. This data is edited in the form, since all controls are linked to the `FCity` instance.

When the user has finished editing, (s)he has now 2 options: hit the `Cancel` button, or the `OK` button. When the `OK` button is hit, the data from the `FCity` instance must be copied back to the instance that should be edited, in `FData`. This is done in the following way:

```
procedure TCityForm.BOKClick(Sender: TObject);
begin
  FData.Assign(FCity);
end;
```

When the user hits the `Cancel` button, the form is simply closed (with modalresult `mrCancel`). Since the data is not copied back, the original city instance remains unmodified. And this is the reason for the copying back and forth of data: since the tiOPF controls edit the object instance directly, there is no way to cancel an edit, such as it is possible in DB-Aware controls: for database-aware controls, all edits are done in a temporary buffer, and the changes are only saved to the dataset when the `Post` method of the attached dataset is called. Until `Post` is called, all edits can be undone with the `Cancel` method of `TDataset`: the temporary buffer is then simply discarded. By contrast, persistent-aware edits apply the changes to the object immediatly. So, to be able to do a cancel operation, the controls must edit a copy of the objects. Cancelling then means simply not copying back the data to the original object. If this mechanism was not used, a cancel would mean reloading the data from the database - which would not be possible if the object had not yet been saved to the database in the first place.

The other forms in the contact application work in similar manners, or even simpler. The resulting application can be seen in figure 4 on page 15

5 Interfacing the database: Loading and saving

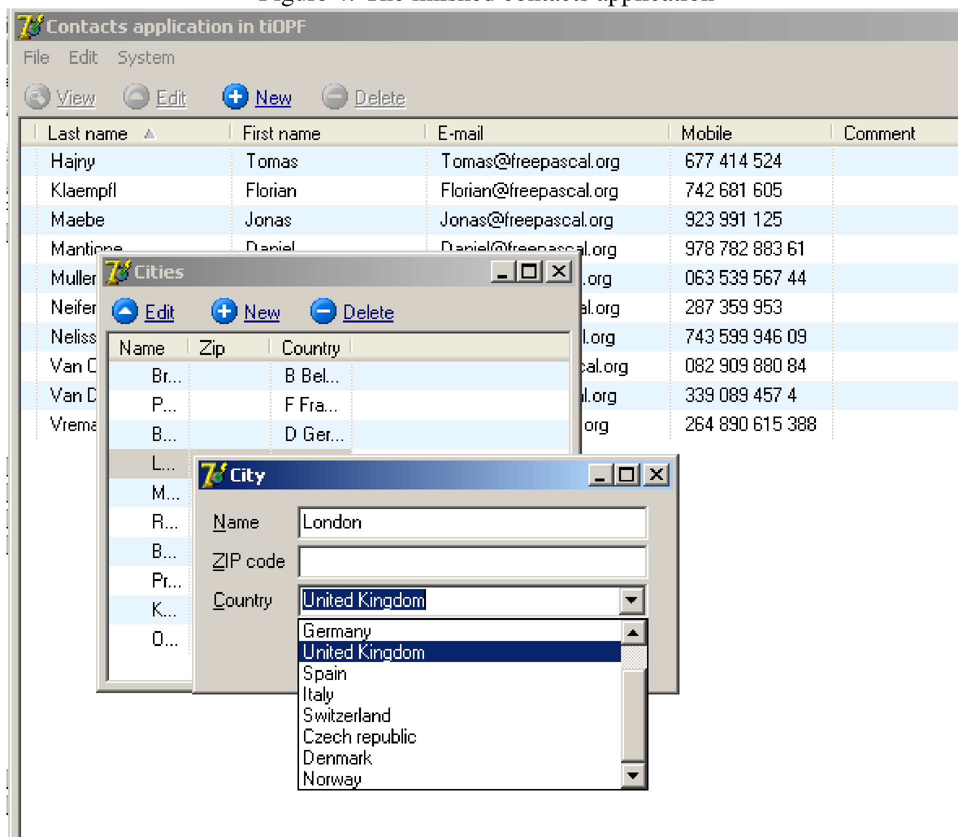
The application as it is now is ready to edit data. The `PopulateContacts` call can be used to create sample data. But the data cannot yet be saved to a database, nor can it be loaded from database. For this, the visitors need to be written, as explained in the previous article. Per class in the business model, visitors must be registered for read and save operations. For save operations, 3 visitors must be registered: one for inserting, one for updating and one for deleting. For reading, a visitor must also be created.

When tiOPF needs to save an object, it lets all registered save visitors visit the object. As soon as the correct visitor has saved the data, the loop stops. The same operation is done when an object must be read from database.

tiOPF has several pre-defined abstract classes for saving and loading objects from database. The central class is `TtiPerObjVisitor`, defined in tiVisitorDB: it is a base class for all vistors that must read or write an object to a database. It introduces the following properties:

Database the database to which this visitor is connected. The database connection will be set up by the tiOPF framework. This property is of type `TtiDatabase`.

Figure 4: The finished contacts application



Query a query object which will be usedto execute the queries. It is of type `TtiQuery`.

Visited the object which must be saved, of type `TtiVisited`

It also has a `SetupParams` method, which must be overridden by the descendents, this should be overridden to set up all parameters that are needed to execute the query correctly.

2 descendents of this class exist. The first descendent is `TtiVisitorSelect`, which can be used to create read visitors. In addition to the `SetupParams` method, the read visitors must override 2 extra methods:

Init in which the SQL statement of the query must be set up.

MapRowToObject in which the query result must be copied to the object. This method will be called for each record returned by the query: this means that it is also suitable to load a list: each element in the list can be mapped from a row.

The second descendent of `TtiPerObjVisitor` is `TtiVisitorUpdate`, which can serve as parent for the update, delete and insert visitors.

The read visitor works in the following way:

- The visitor is created by the framework, a query object is created, and the database property is set to the active connection.
- If the visited is accepted, the query is initialized by calling `Init`.
- The `SetupParams` method is called.
- The query is opened.
- For each record returned by the query, `MapRowToObject` is called.
- The query is closed again.

For the contacts application, the list objects will be responsible for loading all objects in memory. This means that read visitors will be created for the list obects only. No read visitors will be made for the individual objects.

For the countries visitor, the following class is defined:

```
TReadCountriesVisitor = Class(TtiVisitorSelect)
Protected
  Function AcceptVisitor : Boolean; override;
  Procedure Init; override;
  Procedure SetupParams; override;
  Procedure MapRowToObject; override;
end;
```

The methods are as follows. First, the `AcceptVisitor` must be coded:

```
function TReadCountriesVisitor.AcceptVisitor: Boolean;
begin
  Result:=Visited is TCountries;
end;
```

This is necessary, because all known read visitors are tried on an object that must be read. Only after this method has returned `True`, the `Init` method is called:

```
procedure TReadCountriesVisitor.Init;
begin
  Query.SQL.Text:='SELECT ID, ISO, NAME FROM COUNTRY';
end;
```

It simply sets the SQL statement of the query. The following method to be called is the `SetupParams` method, which in this case needs to do nothing (note that it must be overridden anyway, or an exception will be raised at runtime. The method can simply be left empty). After the query is opened, the `MapRowToObject` method is called for each record returned by the query:

```
procedure TReadCountriesVisitor.MapRowToObject;

Var
  C : TCountry;

begin
  C:=TCountry.Create;
  C.OID.AssignFromTIQuery('ID',Query);
  With Query do
    begin
    C.ISO:=FieldAsString['ISO'];
    C.Name:=FieldAsString['NAME'];
    end;
  C.ObjectState:=posClean;
  TCountries(Visited).Add(C);
end;
```

This method creates a new `TCountry` instance (using `Create`, so no new ID is assigned). The unique object ID is assigned from the ID field from the `COUNTRY` table. Then the fields of the country instance are filled from the fields in the `Query` object. After all fields are loaded, the instance is marked as clean, and finally it is added to the countries list.

The `SetupParams` method in this case needs to do nothing. But for the list of addresses of a contact (`TContactAddresses`), the following query is set up:

```
procedure TReadContactAddressesVisitor.Init;
begin
  Query.SQL.Text:= 'SELECT ID, STREET, NR, ' +
                   ' TELEPHONE1, TELEPHONE2, '+
                   ' FAX, CITYID, ADDRESSTYPE '+
                   'FROM '+
                   '  ADDRESS '+
                   'WHERE '+
                   ' CONTACTID = :ContactID';
end;
```

And the `ContactID` parameter must be set up with the unique ID from the `TContact` class that owns the addresses:

```
procedure TReadContactAddressesVisitor.SetupParams;

Var
  C : TContact;
```

```
begin
  C:=(Visited.Owner as TContact);
  C.OID.AssignToTIQuery('CONTACTID', Query);
end;
```

The `Owner` of the list is the `TContact` instance, and it's `OID` must be assigned to the `ContactID` parameter of the `Query` class. This is what the standard `AssignToTIQuery` method does.

There are 3 save visitors for the `TCountry` class:

```
TCreateCountryVisitor = Class(TtiVisitorUpdate)
Protected
  Function AcceptVisitor : Boolean; override;
  Procedure Init; override;
  Procedure SetupParams; override;
end;
```

The `TUpdateCountryVisitor` and `TDeleteCountryVisitor` classes have the same declaration. The implementation of the `TCreateCountryVisitor` is similar to the one of the read visitor:

```
function TCreateCountryVisitor.AcceptVisitor: Boolean;
begin
  Result:=(Visited is TCountry) and
          (Visited.ObjectState=posCreate);
end;
```

Note that the `ObjectState` is explicitly checked. This is necessary, because the 3 save visitors will be tried on the `TCountryInstance` class, and only the correct one should actually execute. The query is again set up in the `Init` method:

```
procedure TCreateCountryVisitor.Init;
begin
  Query.SQL.Text:='INSERT INTO COUNTRY (ID, ISO, NAME) '+
                 ' VALUES (:ID, :ISO, :NAME)';
end;
```

This time, the `SetupParams` method has a little more work:

```
procedure TCreateCountryVisitor.SetupParams;

Var
  C : TCountry;

begin
  C:=Visited as TCountry;
  C.OID.AssignToTIQuery('ID',Query);
  With Query do
    begin
    ParamAsString['ISO']:=C.ISO;
    ParamAsString['NAME']:=C.Name;
    end;
end;
```

Each parameter is filled with the corresponding fields of the `TCountry` instance.

The update and delete classes are completely similar. With these 4 classes, the `TCountry` instances can be loaded from database and saved to database. The same exercise must be made for all other classes in the business model.

Once the classes are implemented (in a unit called viscontactmodel, they must be registered with `tiOPF`. This is done in the `RegisterHardCodedVisitors` procedure:

```
Procedure RegisterHardCodedVisitors;

begin
  With gTIOPFManager do
    begin
    // Countries
    regReadVisitor(TReadCountriesVisitor);
    regSaveVisitor(TDeleteCountryVisitor);
    regSaveVisitor(TUpdateCountryVisitor);
    regSaveVisitor(TCreateCountryVisitor);
    end;
```

The `regReadVisitor` and `regSaveVisitor` calls register the visitor instances with `tiOPF` mechanism. Note that the order in which they are registered is significant, because all save visitors will be tried.

At this point, some things should be noted about the used technique:

- For each class, at least 4 visitors must be created. This is a lot. It pays off to use the code templates features of Delphi's IDE: some sample code templates are present in the tiOPF sources.
- The read visitors are database agnostic, i.e. they should run on all databases: it is not possible to register visitors which work only on 1 database layer.
- tiOPF does not guarantee that the queries will run on all database systems: the queries are passed as-is to the database layer. It is therefore possible that queries that run on e.g. MySQL may not run on Firebird.
- The hardcoded queries offer an incredible amount of flexibility: they are very suitable for reading legacy data.

With all visitor classes in place, how is the data saved or loaded ? This is handled through the `TContactManager` class, which has 2 methods:

```
procedure TContactManager.LoadData;
begin
  gTIOPFManager.Read(Countries);
  gTIOPFManager.Read(Cities);
  gTIOPFManager.Read(Contacts);
end;

procedure TContactManager.SaveData;
begin
  If Connected then
    begin
    gTIOPFManager.Save(Countries);
    gTIOPFManager.Save(Cities);
```

```
    gTIOPFManager.Save(Contacts);
    end;
end;
```

It is simply a matter of calling the `Read` or `Save` methods of the global `gTIOPFManager` instance, and passing it the class one wishes to load or save. In the above case, the global list instances are passed to the `tiOPF` framework: the framework will recursively load or save the objects. Obviously, the `Read` or `Save` methods can be called at any point in the application lifecycle. A single object can be passed to it, this is all up to the application programmer, `tiOPF` does not enforce any restrictions on this.

Note the order in which the lists are loaded. This is important, because in order to load the cities, the countries list must be present, as can be seen in the `MapRowToObject` method in the `TReadCitiesVisitor`:

```
procedure TReadCitiesVisitor.MapRowToObject;
Var
  C : TCity;
  CC : TCountries;

begin
  C:=TCity.Create;
  C.OID.AssignFromTIQuery('ID',Query);
  With Query do
    begin
    C.Zip:=FieldAsString['ZIP'];
    C.Name:=FieldAsString['NAME'];
    CC:=ContactManager.Countries;
    C.Country:=CC.Find(FieldAsString['COUNTRYID']) as TCountry;
    end;
  C.ObjectState:=posClean;
  TCities(Visited).Add(C);
end;
```

The `Find` method (a standard method of any list class such as `TCountries`) will search for an object with a matching `OID`. Obviously, if the `Countries` global list would not yet be loaded when the cities are loaded, the city would have `Nil` as a `Country` pointer. Therefore the countries list must be loaded first (and saved first, for database integrity).

In case no global `TCountries` list is available, then the following could be done instead:

```
C.Country:=TCountry.Create;
C.Country.OID.AssignFromTIQuery('COUNTRYID',Query);
```

This would create a `TCountry` instance, and set its OID property with the ID of the country as stored in the database. At a later point the `TCountry` instance could be read fully with a read visitor. A drawback of this technique is that if 2 cities refer to the same country, 2 country instances with the same OID would be present in the application, and changes to one instance would not be reflected in the other. This can be remedied by using a hybrid technique: use the `Find` method to search for the instance (`Find` works recusively), and if not already present, instantiate one.

6 Interfacing the database: Connecting and disconnecting

All methods to load and save objects from/to database are present. However, no code has been created which actually connects to a database. Fortunately, this is very easy. `tiOPF` comes standard with a lot of database connectivity components (persistence layers), which are completely self-contained. This means that it is not necessary to drop any components on a form, or instantiate database classes: `tiOPF` takes care of all that. The only thing that needs to be done is to include the unit with the desired persistence layer in the project. The unit will register itself in the `tiOPF` persistence framework, and is then ready for use.

It is even possible to use multiple persistence layers at once in the same application, or to connect to multiple databases with a single persistence layer: when loading and saving objects (using the `Read` and `Save` methods) the persistence layer to be used can be specified, or the specific database to be used can be specified. Both parameters are optional, and if they are not specified, defaults are used.

What persistence mechanisms are available ? A lot:

tiQueryADO Any database access using ADO components

tiQueryBDE Any database access using BDE components

tiQueryXML Database in XML file

tiQueryCSV Database in CSV text file (comma separated)

tiQueryTAB Database in tab-delimited text file

tiQueryIBX Interbase/Firebird database access using IBX

tiQueryIBO Interbase/Firebird database access using IBO

tiQueryFBL Interbase/Firebird database access using FBlib

tiQuerySQLDBIb Interbase/Firebird database access using Lazarus' SQLDB.

tiQueryZEOSXXXX with XXXX one of IBFB, FB10 or FB15: Interbase/Firebird database access using ZeosLib.

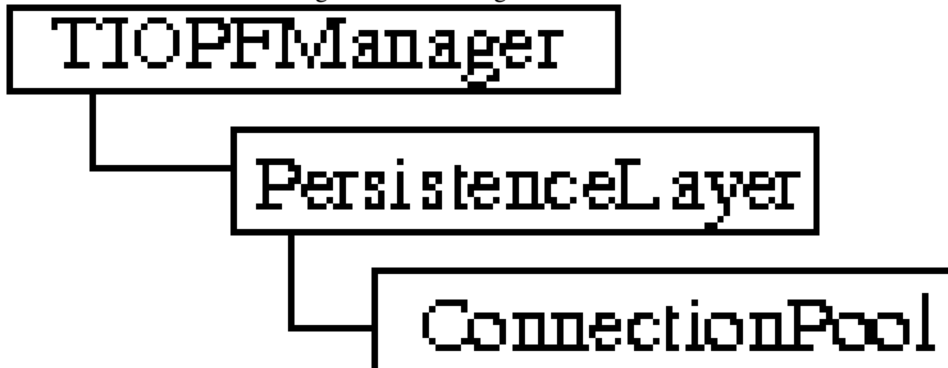
There are some more, but these are the main ones (the available units show that Firebird/Interbase is still a natural compagnon for Delphi/Lazarus).

The contacts application is coded with IBX components, since IBX comes standard with Delphi. Therefore `tiQueryIBX` is included in the uses clause of the application.

It is possible to check at run-time which persistence layers are compiled-in: The global `gTIOPFManager` class has a property `PersistenceLayers` which holds the registered persistence layers. To show the contents of this list, a menu item is added to the main menu, and in the `OnClick` handler, the `ShowDatabaseLayers` procedure is called:

```
procedure TMainForm.ShowDatabaseLayers;

Var
  I : Integer;
  S,L : String;
  PL : TtiPersistenceLayer;

begin
```

Figure 5: Accessing the connections



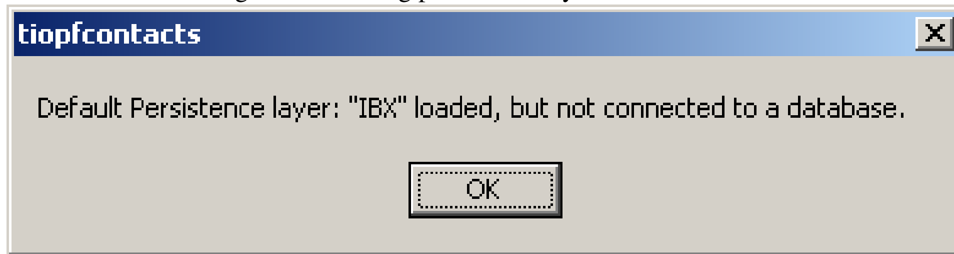

```
  L:='';
  for i := 0 to gTIOPFManager.PersistenceLayers.Count - 1 do
    begin
    if (L<>'') then
      L:=L+sLineBreak;
    PL:=gTIOPFManager.PersistenceLayers.Items[i];
    If (PL.PerLayerName=gTIOPFManager.DefaultPerLayerName) then
      L:=L+'Default ';
    L:=L+Format('Persistence layer: "%s" ',[PL.PerLayerName]);
    S:=Trim(PL.DBConnectionPools.DetailsAsString);
    if (S='') then
      L:=L+'loaded, but not connected to a database.'
    else
      L:=L+'Loaded and connected with :'+S;
    end;
  If (L='') then
    ShowMessage('No persistence layers compiled-in')
  else
    ShowMessage(L);
end;
```

This procedure loops over the `PersistenceLayers` list (it is a descendent of `TtiObjectList`), and examines every `TtiPersistenceLayer` item in the list. If the persistence layer is the default persistence layer, it is marked as such. The `DBConnectionPools` property contains a list of database connections for that persistence layer. The `DetailsAsString` returns a string representation of all connected databases for that persistence layer. The connection between the various classes is shown in figure 5 on page 22, and the output for the dialog when no connection is yet made, is shown in figure 6 on page 23. By adding some of the units described above to the uses clause of the program, more persistence layers will be shown.

Now that a persistence layer is built in our application, it's possible to connect to a database. The global `gTIOPFmanager` object does this using the `ConnectDatabase` call, which is defined as follows:

```
procedure  ConnectDatabase(const ADatabaseName : string;
                           const AUserName    : string;
                           const APassword    : string;
                           const AParams      : string;
                           const APackageID   : string);
```

Figure 6: Showing persistence layers and databases



The first three parameters are obvious: they are the database filename, username and password for the database one wishes to connect to. The `AParams` argument are extra parameters that can be passed to the database: for IBX, this can be anything that can be specified in the `Params` property of the `TIBDatabase` component. The `APackageID` parameter is the name of the persistence layer, in the case of the contacts application this should be `'IBX'`. Overloaded versions of the `ConnectDatabase` call exist without `AParams` and `APackageID` parameters: if omitted, they will be replaced with default values (for the persistence layer name, the default is the first registered persistence layer).

For the contacts application, which has just 1 persistence layer, this means that in the main form of the application, a `Connect` method can be coded as follows:

```
procedure TMainForm.Connect;

begin
  gTIOPFManager.ConnectDatabase(
    'host:/home/firebird/contacts.fb',
    'SYSDBA',
    'masterkey');
  ContactManager.LoadData;
end;
```

As can be seen, all the data is loaded right after the connection was made. Until the user actually connects to the database using the File-Connect menu item, no data will be visible.

Disconnecting happens in a similar way:

```
procedure TMainForm.Disconnect;
begin
  If ContactManager.Connected then
    begin
    ContactManager.SaveData;
    gTIOPFManager.DisconnectDatabase;
    end;
end;
```

If the application is still connected, the data is saved prior to actually disconnecting using the `disconnectdatabase` call of the `TTIOPFManager` class. This means that all changes made to the contact data are not saved to database until the user quits the application. While this is not the best design decision, for the sample application this is enough. In a real-world application, data would be saved as soon as one of the dialog windows is closed: in case the application crashes, the changes will not be lost.

Checking whether the application is still connected to the database is delegated to the `ContactManager` class. The `Connected` function takes care of this. Since `tiOPF`

is capable of working with multiple persistence layers, each of which can be connected to various databases, checking whether the application is still connected is not as straightforward as in a traditional 1-database application. The `Connected` function looks as follows:

```
function TContactManager.Connected: Boolean;

Var
  N : String;

begin
  With gTIOPFManager do
    begin
    N:=DefaultDBConnectionName;
    Result:=DefaultPerLayer.DBConnectionPools.IsConnected(N)
    end;
end;
```

The `DefaultDBConnectionName` property of `gTOIPFManager` contains the name of the connection that was first set up - in the case of the contacts application, there is only 1 connection, so this will automatically be the correct one. Since there is only 1 persistence layer, it will be the default one, and so the `DefaultPerLayer` property will point to the IBX persistence layer. The `DBConnectionPools` property of the persistence layer has a function to decide whether a given database name is connected or not: `IsConnected`. This function returns the result that is actually needed.

In case of multiple persistence layers and multiple databases, checking wether the application is still connected to a database would mean iterating over all persistence layers and databases to see if they are still connected, much like in the `ShowDatabaseLayers` procedure presented earlier.

The scenario of multiple persistence layers is not as unlikely as it may seem: in the interest of performance, it could be logical to have one persistence layer which connects to a remote database, and a second one which stores data locally in CSV or XML files. Whenever lookup lists (such as cities or countries) are needed, one would look for them in the local persistence layer: if not not found there, they can be loaded from the remote database, and immediatly saved to disk for future use. This would considerably speed up the startup time of the application.

7 Interfacing the database: Unique objects

Till now, the actual database schema for the tiOPF contacts application was not yet discussed. This is because an important design decision is expected by tiOPF. All business objects have a unique ID assigned to them, exposed in the `OID` property of the `TtiObject` class. This property is a class in itself (`TOID`). The reason is that tiOPF does not enforce a certain type of ID: the ID can be numerical, a GUID, a string - whatever fits the need. The only requirement is that the OID is unique among all objects.

tiOPF assigns an ID automatically if the `CreateNew` method of `TtiObject` is called. How does it know which kind of ID should be assigned ? For this, the programmer must include one of the pre-defined units that provide OID support in his application. tiOPF comes with several units:

tiOIDGUID each OID is a GUID. The `OID` has the advantage that no extra database storage is needed; the next value is simply fetched from the OS and should normally

be unique.

tOIDHex each OID is a 32-char hexadecimal value.

tOIDInt64 each OID is a 64-bit integer.

tOIDInteger each OID is a 32-bit integer.

tiOIDString each OID is a string, much like tOIDHex but with all possible characters.

If one of these units does not fit the need of the application, a new descendent of `TOID` can be created which suits the need of the database. 2 classes must be created: one descendent of `TOID`, and a descendent of `TNextOIDGenerator`, which is responsible for generating the new value.

In case the last generated value must be stored in the database, a read visitor can be registered for the `TNextOIDGenerator` which stores the value in the database. The units above give sample implementations of such a visitor.

For the contacts application, a 64-bit integer will be used as the primary key for each table, and as value for the `OID` property. This means that the tiOIDInt64 unit must be included in the uses clause of the application. This unit expects to be able to save the value for the next OID in a table called `NEXT_OID` with a single field `OID`: the table and field must be present in the database, and will automatically be filled with the next value. (actually, the next value divided by 100 is stored: this means that the next ID must only be stored for every 100 generated IDs. This avoids unnecessary database access.

For the contacts application, the SQL statement for the `NEXT_OID` table would look like this:

```
CREATE TABLE NEXT_OID (
       OID BIGINT NOT NULL
);
```

While for the countries table it would look like this:

```
CREATE TABLE COUNTRY (
  ISO VARCHAR(2),
  NAME VARCHAR(50) NOT NULL,
  ID BIGINT NOT NULL,
  PRIMARY KEY (ID)
);
CREATE INDEX COUNTRYISO ON COUNTRY(ISO);
```

The rest of the statements can be found in the contacts.sql file that comes with the source code.

There are are 2 things to note:

1. The ID is generated by the application: Database mechanisms such as generators or sequences are not used at all: the database is used solely to store the next value, it is not used to actually generate the value, this happens in code.
2. The generated ID is unique for all object instances in the business model. In traditional relational databases, a single ID is used per table; tiOPF uses a single ID for all objects (and hence all tables). This allows the `Find` method to locate the instance of any object, using solely it's OID.

Both items can be dealt with: in the case of a legacy database, which is also used by other applications, custom `TOID` and `TNextOIDGenerator` descendents can be created which use for instance a database-generated ID. The unique TOID value for use by tiOPF could for instance be the classname, followed by the ID generated by the database, which should together form a unique OID value.

8 Conclusion

In this article, the basics of creating an tiOPF application have been shown. It should be obvious that creating a `tiOPF` application requires a lot of coding: the RAD aspect of Delphi programming is a bit lost: many properties should be set run-time, events must be created for a lot of things that can be done without coding in traditional database applications, or even in other object persistence frameworks: InstantObjects does not need events for insert/delete/add operations.

When coding the GUI, one must know the various tiOPF units: many of the parameters to events of persistence-aware controls are 'unknown': the unit in which the parameter type is defined is not in the same unit as the control, and hence was not added automatically to the 'uses' clause of the form unit. The programmer must be prepared to add a lot of tiOPF units manually to the uses clause of the forms he is programming - which presupposes that he knows the unit in which the parameter types reside.

The database persistence mechanism is very powerful, and can be taylored so tiOPF can deal with almost any legacy database. In difference with InstantObjects, no database model is enforced: the programmer is free to save/load the objects as he wishes. Once more, this power comes with a price: for each object in the business model, no less than 4 visitors must be created in order to be able to load or save the objects to the database. Delphi's code templates are a must for the tiOPF programmer.

As it was said in the previous article: programming tiOPF is not for the faint of heart. For all the power it offers, still many things are left to the programmer. This may be good for the experienced OOP programmer, but is likely to put off the beginner. However, once the basics have been mastered, `tiOPF` more than makes up for the time spent on learning it: it's a very powerful framework, which definitely should be in the toolbox of any Object Pascal programmer.