Object Persistence: Using InstantObjects

Michaël Van Canneyt

December 6, 2006

Abstract

In the series about Object Persistence Frameworks for Object Pascal, InstantObjects is the first framework to be investigated. The architecture of this free package is discussed and investigated using a small contacts application.

1 Introduction

InstantObjects is a free framework for Object Persistence for the Object Pascal language. It goes back a long way: in a previous life, it was the property of a commercial company, but was released to the public, and ever since it is managed by a group of enthousiasts. It has been around for quite a while now, and can therefore be considered quite stable and usable.

The current version is 2.0, and it can be downloaded from the InstantObjects website:

`http://www.instantobjects.org/`

Installation is rather straightforward (more about this below) and it comes with several demonstration projects, and a help file.

InstantObjects can store objects in many databases: it comes with several so-called connection brokers, which allow to store data in a multitude of databases: NexusDB, BDE, IBX, DBX, UIB, ADS, ISAM, ZeosDBO and ADO. It also features an XML broker, which stores the data in XML files.

To create the classes which should be stored in the database, InstantObjects comes with a class editor dialog. It allows to define the properties that should be made persistent, and specifies some options for storage in the database. It does not allow editing of methods, only properties. The class editor dialog interacts with the Delphi IDE: it creates and modifies the code for all the properties.

The presentation layer works through standard Data-Aware controls. This has the advantage that all of the data-aware controls out there can be used with InstantObjects, but also has all drawbacks related to DB-Aware controls. Of course, the presentation layer does not need to be used.

InstantObjects creates the database for the model, in any of the supported databases. Since applications and models evolve through time, InstantObjects also offers a mechanism to update the database from one version of the model to the next: the 'evolve' feature.

All these features make InstantObjects an easy-to-use object persistence framework, and any Delphi programmer should be able to start to work with it in no time.

2 Installation in Delphi

InstantObjects comes as a series of packages which must be installed in the IDE. The IDE needs to have TDataset support, so not all versions of Delphi are possible (more specifically, the free versions will not work). The packages are located in the Source subdirectory of the distribution. There are 3 sets of packages:

1. The InstantObjects core packages: Located in the Core and `Design` subdirectories of the Source directory. The design package installs InstantObjects support in the IDE. These packages must always be installed.
2. The database brokers: this is a series of packages, one for each database access engine in Delphi. Each of them handles the connection to a database. Obviously, only the ones for which the appropriate database components are installed should be installed, i.e. it makes no sense to install the ZeosDBO connector if ZeosDBO is not installed. The XML engine can always be installed. Each package will install a `TInstantConnector` descendent in the component palette.
3. Catalogs read metadata from the database engine. Currently catalogs for Interbase and MS-SQL are shipped with InstantObjects.

That's it. After this is done, InstantObjects is ready to be used.

The example projects can be compiled after this, and should work out of the box.

3 InstantObjects Architecture

InstantObjects is built around several basic classes:

TInstantObject This is the base class for all persistent objects. Any object that needs to be persistent in an InstantObjects model, descends fro TInstantObjects.

TInstantConnector This class is never used directly, but a lot of descendents of this class exist: each of these descendents encapsulates the access to a database. For almost each database technology present in Delphi, a corresponding `TInstantConnector` descendent exists.

TInstantSelector This class allows to retrieve a set of classes from the database, and represents ('exposes') them as records in a `TDataset`. This is what allows to use DB-aware controls to edit the objects.

TInstantExposer This class allows to edit a single object (or an attribute of an object that is a list of objects) as a record in a `TDataset`.

Many more classes exist in InstantObjects, some auxiliary components are even installed on the component palette, such as the following.

TInstantConnectionManager in case the end-user application must be able to connect to multiple databases, then this component can be used to let the end-user manage the database connections. It comes with database connection configuration dialogs for all databases.

TinstantPump This component allows to move model data from one database to another.

TInstantDBBuilder This component allows to build a database which can be used to contain the model data.

TInstantDBEvolver This component allows to adapt a database from one version of a model to the next version.

These components will not be discussed here.

It is instructive to examine some properties and methods of the `TInstantObject` class. This class has 3 constructors:

```
Constructor Create(AConnector: TInstantConnector = nil);
Constructor Retrieve(const AObjectId: string;
                     CreateIfMissing: Boolean = False;
Constructor Clone(Source: TInstantObject;
                  AConnector: TInstantConnector = nil); overload; virtual;
```

The meaning of these constructors should be obvious:

Create creates a new instance. The database connector can be specified in `AConnector`. If none is specified, the default connector is used. (This is the `TInstantConnector` in the application whose property `IsDefault` is set to `True`).

Retrieve With this constructor, a new instance is created, and the attributes are fetched from the database, using `AObjectID` as the unique `ID` for the instance. If the ID cannot be found in the database, then an empty instance is created if `CreateIfMissing` is `True`.

Clone This creates a new instance and clones all attributes from the `Source` instance. Needless to say, the classes should be compatible. Here also, a connector can be specified.

`TInstantObject` also has some interesting properties:

Id this is the unique ID of this object. By default, this is a string of 32 characters and is filled with A GUID. (although this can be changed).

IsChanged a boolean property indicating whether the object has any changes which have not yet been written to the database.

RefCount Shows how many times the object is referenced by other objects in the application. If the refcount is zero, then the object can safely be destroyed.

Owner if the object is owned (i.e. it's a part or parts attribute of another class) then this property returns the owning object.

And some methods:

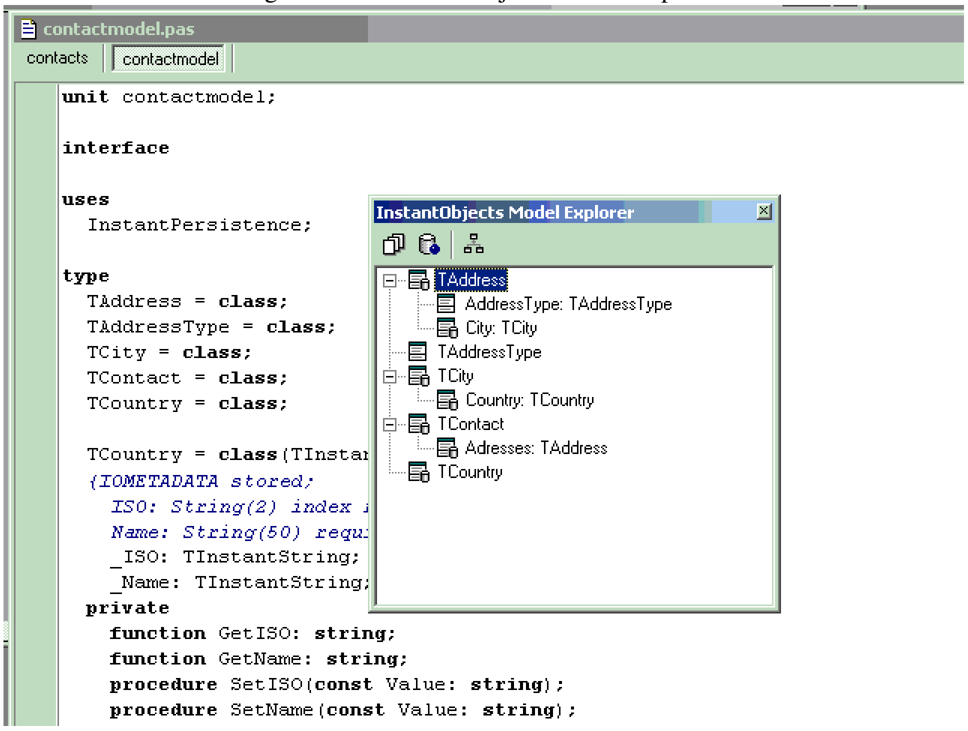
Store Saves any pending changes to the database.

Dispose Removes the instance from the database, but not from memory. This means that the object can be stored again in the database (but with a different ID). Note that Destroying or freeing the object is not the same as disposing it.

Abandon Calling this will break all connections with the database: the object can no longer be stored or disposed.

There are many more methods and properties but these are the main ones. All the others are documented in the help file of InstantObjects.

Figure 1: The InstantObjects Model Explorer



```
unit contactmodel;

interface

uses
  InstantPersistence;

type
  TAddress = class;
  TAddressType = class;
  TCity = class;
  TContact = class;
  TCountry = class;

  TCountry = class(TInstar
  {IOMETADATA stored;
    ISO: String(2) index i
    Name: String(50) requ
    _ISO: TInstantString;
    _Name: TInstantString;
  private
    function GetISO: string;
    function GetName: string;
    procedure SetISO(const Value: string);
    procedure SetName(const Value: string);
```

```
TAddress
  AddressType: TAddressType
  City: TCity
TAddressType
TCity
  Country: TCountry
TContact
  Adresses: TAddress
TCountry
```

Since each class in a model will descend from this class, it's important to know the above methods.

There are many more classes in InstantObjects which are involved in persistence and database mappings, but normally one is not confronted with them: they are used internally, and are automatically handled by InstantObjects. The classes presented here should be the only classes that one encounters in a vanilla InstantObjects application.

4 The InstantObjects model explorer

Once InstantObjects is installed, the model can be defined in InstantObjects. For this, the 'InstantObjects Model Explorer' should be used (depicted in figure 1 on page 4. It contains the complete model in a big tree. The model explorer can be called from the 'View' menu in Delphi, and is a non-modal window, which can be kept open all the time: if a different project is loaded, the model of this project is shown at once. The model tree in the model explorer can be built up in 2 ways: showing the inheritance of the objects (the default), or showing the classes and their relation to each other.

The model explorer can be used to maintain the classes themselves, but is also used to build the database which should contain the data of the classes. Before it can be used, it should be told which units in the project will contain the business classes: these units should be created first, and can then be selected using the 'Select units' button.

The interface of the model explorer is very straightforward: To create a new class, the 'Insert' key can be pressed (or select 'New Class' from the model tree popup menu)

This (or editing an existing class) will pop up the class editor dialog. The dialog is pretty straightforward. At least the following things must be specified:

Class Name the name of the class, obviously.

Base class is the class from which the new class should be descended. All classes used in an InstantObjects application should at least descend from `TInstantObject`, which is the base class for persistent objects.

Unit This is the name of one of the units that were selected to contain the model classes: The model editor will insert the class declaration and basic implementation in that unit.

Persistence determines how the data for this class will be stored. This can be either **Stored**, in which case the data will be stored in a table, or `Embedded`. Embedded storage means that the data will be stored with the data of the object that uses this class. For example, the `TAddressType` which indicates the type of an address (it's a property of the `TAddress` class) is a separate class, but is stored embedded, which means that the data is stored together with the `TAddress` data.

Storage Name is the name of the table in which the data for this class will be stored. Note that the same table can be used to store data of more than 1 class: InstantObjects can differentiate between the objects.

After this, the storage of the class is determined: InstantObjects knows enough to handle storage for the class. To determine which data should be stored, the attributes (or properties) of the class must be defined. This happens also in the class editor dialog, on the second tab. A simple list of attributes is displayed, and attributes can be added, modified or deleted.

For each attribute, several things must be specified:

Name The name of the attribute. this can be any valid pascal identifier.

Type The type of the attribute: this can be a basic type, or a reference to another object (more on this below). In some cases (notably strings), the maximum size must be indicated.

Storage name This is the name under which the attribute will be stored (a field name).

Storage kind determines how references to other objects are stored. More on this below.

The attribute editor can be seen in figure 2 on page 6.

When a class definition contains a reference to one (or more) other objects, there are several options which determine how the objects are related:

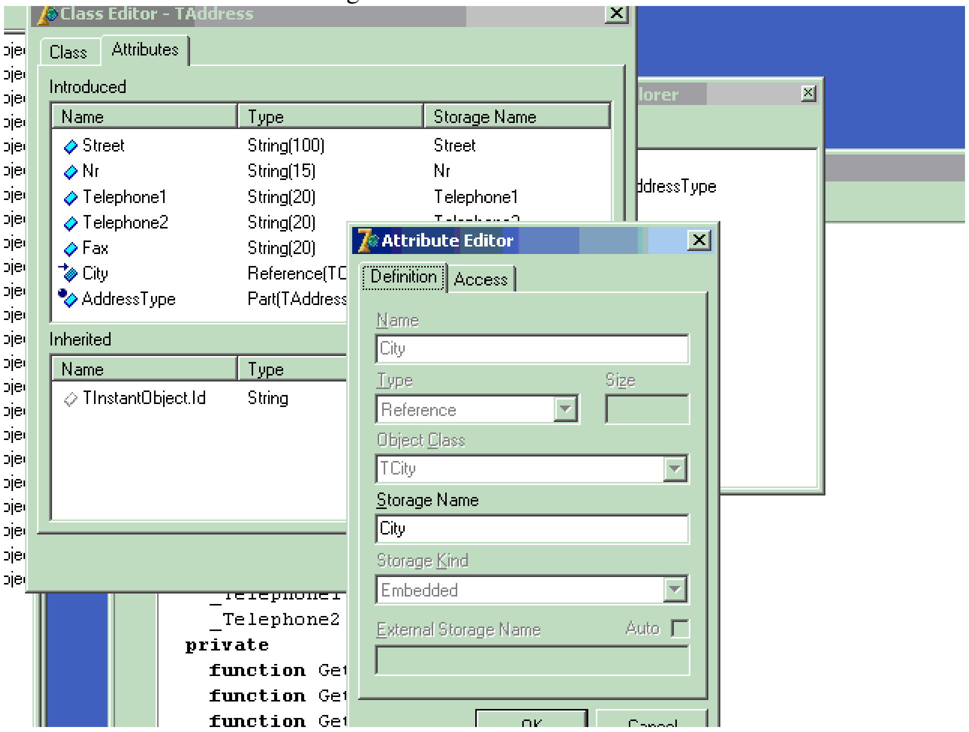
Part this is for the case when the referenced object exists only as a part of the current object, and will never be instanced by itself: it is 'part' of the owning class. For instance, in the contacts application, the `TAddressType` exists only as an attribute of a `TAddress`. (a one-way relationship)

Parts Has the same meaning as `TPart`, but there are multiple objects. In the contact application, `TAddress` exists as a `Parts` of `TContact`.

Reference should be used when an object refers to another objects, which can be instantiated by itself. In the case of the contacts application, `TCity` refers to `TCountry`: the list of countries can be maintained by itself. (a two-way relationship)

References is similar to `Reference`, but more than one object can be referenced.

Figure 2: The attributes editor



Class Editor - TAddress

Class | Attributes

Introduced

Name	Type	Storage Name
Street	String(100)	Street
Nr	String(15)	Nr
Telephone1	String(20)	Telephone1
Telephone2	String(20)	
Fax	String(20)	
City	Reference(TC	
AddressType	Part(TAddress	

Inherited

Name	Type
TInstantObject.Id	String

```
    _Telephone2
  private
    function Get
    function Get
    function Get
```

Attribute Editor

Definition | Access

Name	City
Type	Reference
Size	
Object Class	TCity
Storage Name	City
Storage Kind	Embedded
External Storage Name	
Auto	☐

Which reference should be chosen depends entirely on the needs of the model. Roughly, one could say that parts are owned by the source class: if the source class instance is deleted, it's parts will also be deleted. For a reference, the referenced class will not be deleted.

Note that InstantObjects does not enforce referential integrity at the database level (although this feature is planned). This means that it is possible to delete an object while there are still references to it, leaving the database in an inconsistent state. It is of course possible to create the necessary foreign keys manually.

For regular (primitive) attributes, the storage always happens in the table of the class of which the attributes are part. For references to other objects (parts or references) the storage can be done in 2 ways. Which way is used, depends on the value of `Storage Kind`. This can have one of 2 values:

embedded the parts or references are stored in a blob field in the table of the referring class.

external here, the parts are stored in a separate table, and the links between part and owning object are stored in an intermediate table. The same is true for references: the references are stored in an intermediate table.

Other than the definition and storage options for an attribute, which constitute the main part of the attribute's definition, InstantObjects also allows to define some other characteristics of the attribute.

Visibility the pascal visibility of the attribute. To be usable in the presentation layer, the visibility should be at least 'Published'.

Options this includes Indexed, Read-Only, Required, read-only and default.

Edit Mask an edit mask to be used when editing this attribute.

Valid characters allowed characters when providing a value for this attribute.

Display width the preferred display with for display in DB-Aware grids.

Default value A default value for this property.

When all the classes and their attributes have been entered, InstantObjects can create the database. InstantObjects completely controls the creation of the database, based on the definitions of the classes, attributes and their storage properties. A database can be created for all broker types that were installed in the IDE.

For the purposes of the article, a UIB (Unified Interbase) connection is used: it's very lightweight. The following objects are defined in the modeler for the contact application:

TAddressType the type of an address, it has a single string attribute `Name`, and it's persistence type is embedded.

TCountry Has 2 string attributes `ISO` and `Name`, and is stored.

TCity Has 2 string attributes `Zip` and `Name`, a reference to a `TCountry`, an attribute of type `TAddressType` (a part) and is stored.

TAddress consists of several string attributes: `Street`, `Nr`, `Telephone1` and `Telephone2` as well as `Fax`, plus a reference to a `TCity`. it's persistence is set to stored.

TContact has several string attributes: `FirstName`, `LastName`, `Email` and `Mobile`. The `Addresses` attribute is a parts, referring to a `TAddress` object.

TBusinessContact descends from TContact and has 2 string attributes: `Title` and `Company`. The storage name is set to `Contact`

TPersonalContact descends from TContact and has 1 string attribute, `SpouseName`. The storage name is left to the default `PersonalContact`.

The objects here embed some design choices:

1. `TAddressType` is stored embedded. This means that there is no separate list of address types, which could be used for lookup values.
2. By contrast, `TCity` is stored separately: this means that each address refers to a city. The application can maintain and set a list of cities. It also means that a new TCity must be created before it can be referred to in an address.
3. Addresses are stored as parts, which means they cannot be re-used.
4. The `TBusinessContact` object has the same storage name as `TContact`: this shows that it is possible to store multiple classes in 1 table.
5. The `TPersonalContact` object has a different storage as `TContact`. This means that it's data is stored in 2 tables: the attributes it inherits from `TContact` are in the table used by TContact, and the attributes introduced in `TPersonalContact` are stored in `PersonalContact`.
6. The telephone, fax and mobile numbers could have been defined as a separate `TTelephone` class, and stored embedded in the owning classes. This was not done, below the consequences of this decision will be shown.

It's instructive to have a look at the tables which InstantObjects creates based on these definitions. Opening the database in a firebird management tool (such as FlameRobin), reveals the following tables:

Address this table keeps the addresses. It contains columns `CityClass` and `CityID` which form the reference to the `TCity` instance. The `AddressType` attribute (though a class in itself) is stored in the `AddressType` field.

City this table keeps the cities. It contains columns `CountryClass`, `CountryID` wich hold the reference to the `TCountry` instance.

Contact This table holds the `TContact` and `TBusinessContact` data. The `Addresses` attribute is not stored in this table.

Contact_Adresses This table holds the references between `TContact` and `TAddress` instances.

Country This holds the `TCountry` data.

PersonalContact this table holds the attribute (spousename) of the `TPersonalContact` class. The attributes inherited from `TContact` are stored in the `Contact` table.

Each of the tables, except the `Contact_Adresses` table, contains 3 standard InstantObject fields:

Class contains the classname of the stored object. This allows to store data of multiple classes in 1 table (especially important for Descendent classes). It's length is 32 characters.

ID a unique ID, identifying the instance. Together with the `Class` field, the `ID` field forms the primary key of the table.

UpdateCount this field is incremented each time the instance writes changes to itself to disk: this way, changes to an object in another application can be detected: if the object needs to write itself to disk, it first checks the update count on disk: if it matches with the update count in memory, it's safe to write the changes to disk. If the count does not match, it knows the object's data have been changed by another application, and appropriate action can be taken.

5 Adding business logic

The model created in the previous section only specifies the attributes of the business classes, and how the classes are stored in the database. That is all what InstantObjects does for the developer. No methods have been defined, no constraints on possible attribute values and so on.

In the InstantObjects model explorer, the units in which the classes should be implemented were indicated. Whenever a class is edited, the InstantObjects model explorer adapts the source code of the class to match the new properties.

Methods which are inserted by the are left untouched. Therefore methods will not be treated here, as they have no impact on the working of InstantObjects. However, business rules do have an impact.

A closer look at the definition of the classes reveals that the fields for the attributes are not primitive pascal types. For instance, in the `TAddress` class, the Telephone1 and Telephone2 fields are defined as follows:

```
TAddress = class(TInstantObject)
  _Telephone1: TInstantString;
  _Telephone2: TInstantString;
private
  function GetTelephone1: string;
  function GetTelephone2: string;
  procedure SetTelephone1(const Value: string);
  procedure SetTelephone2(const Value: string);
published
  property Telephone1: string read GetTelephone1 write SetTelephone1;
  property Telephone2: string read GetTelephone2 write SetTelephone2;
end;
```

Note that the `_telephone1` field is of type `TInstantString`, and therefor the declared type of the `Telephone1` property (String) does not match, and a read and write specifier must be used to get or set the property value. Luckily, InstantObjects generates all needed code.

The write procedure looks as follows:

```
procedure TAddress.SetTelephone1(const Value: string);
begin
  _Telephone1.Value := Value;
end;
```

This method is generated by the InstantObjects modeler. In this method, some business-rules can be implemented. For instance, code could be inserted to check whether the value is a valide telephone number, and refuse any invalid telephone numbers:

```
Procedure CheckPhoneNumber (S : String);

begin
  If (S<>'') then
   begin
   {Insert code to check validity. Raise exception if S
    does not contain a valid telephone number}
   end;
end;

procedure TAddress.SetTelephone1(const Value: string);
begin
  CheckPhoneNumber(Value);
  _Telephone1.Value := Value;
end;
```

Note that the empty value should be accepted as a valid telephone number.

The above code presents the consequence of the decision to define all phone numbers as a string attribute, instead of defining it as a separate class: In each write specifier for an attribute (property) that represents a telephone number, the call to `CheckPhoneNumber` has to be inserted. There are 4 occurrences in the contacts model, which all need to be modified.

If, by contrast, a separate class `TTelephone` had been defined with a single attribute `Number`, and all attributes had been declared of type `TTelephone` (with embedded storage), then the check procedure could have been implemented in the write specifier for

`TTelephone.Number`, and all telephone numbers throughout the model would have automatically been checked.

Obviously, the above is a simple example. More complex examples of business logic can be found in the examples accompagnying InstantObjects. For the purpose of this article, no other business rules will be implemented.

6 Creating a presentation layer: the GUI.

Now that the model has been created, the application can be programmed. Instantobjects makes it very easy to create a GUI application for a model: it allows the programmer to use all DB-Aware controls to display and edit the objects in the model. It does this with the aid of 2 components which present objects (and their properties) as a `TDataset` descendent.

TInstantSelector this is the equivalent of a `TQuery` object. It allows to specify a SQL-SELECT like statement which returns a series of objects: each object is a row, and the attributes of the objects are a column. Obviously, all objects in the query should have the same base type. The `TInstantSelector` has some properties that control which attributes are loaded (discussed below). The command statement can specify conditions on the returned objects, and can be parametrized as a `TQuery`, although master-detail relations are not possible (although they can be coded)

TInstantExposer This class can be used to represent a single object as a `TDataset`, or a series of objects if an attribute of type 'Part(s)' or 'Reference(s)' is assigned to it. Which of the 2 is used, depends on a `Mode` parameter. The `TInstantExposer` class can be used to construct Master/Detail relationships. The object which is being exposed is available through the `Subject` property.

Both classes have a property 'ObjectClassName' which is the classname of the object which they will edit. (i.e. the name of a single instance). The two classes will be described more in detail:

The `TInstantSelector` can be used whenever a selection of instances of a certain object must be shown to the user: in a grid, in a lookup list. It allows to enter a SQL like statement. It must be of the form

```
SELECT * FROM [ANY] CLASSNAME [WHERE FILTER-CLAUSE]
```

The names used must be a class name (not the storage name) and the attributes in the class. The `ANY` keyword tells InstantObject that also descendent classes must be included. For instance:

```
SELECT * FROM TCONTACT
```

Will retrieve all `TContact` instances, but not `TBusinessContact` or `TPersonalContact` instances. On the other hand, the following statement:

```
SELECT * FROM ANY TCONTACT
```

will also retrieve `TBusinessContact` and `TPersonalContact` instances.

Note that it is not allowed to specify which attributes should be fetched, the fieldlist is always '*'. InstantObjects always fetches all attributes. Exactly which fields are created for the attributes depends on the `FieldOptions` property. This is a set property, which can include the following elements:

foObjects this will create a field for object references (be it parts or references) which represents the 'Self' of the object. This field will be of type integer, and contains the pointer to the object. (note that this is not correct for 64-bit platforms). The `Self` of an exposed class will also be included.

foThorough specifying this option will include fields of referenced objects in the fieldlist. For instance. Note that this is independent of the `foObjects`

foRecurseNesting This option controls the recursive nesting of TDatasetField fields. Parts and references are included in a TDatasetField containing a TDataset of their own: if `foRecurseNesting` is specified, this process continues recursively (and may lead to a lot of fields and data)

The instantselector is a Read/Write dataset: all objects in the dataset can be modified, and the changes will be written back to the dataset. The dataset can be put in read-only mode, but the mechanism in which changes are handled can also be controlled with the Options property. It's again a set with the following items:

eoAutoApply Changes made to the exposer's current object (Subject) or objects contained in the current object should be applied automatically. For `TInstantObject` descendants this means that `Store` is called.

eoAutoRemember Automatically applies `Remember` and `Revert.` functionality to the current object: this means that changes can be cancelled. Since this requires extra memory, this functionality can be switched off.

eoSyncEdit If this option is specified, then any changes that are made to an object directly, are immediatly reflected in the dataset, even if it is in edit mode. (The default behaviour is not to modify the contents of the dataset buffer if it is in edit mode). This option will be demonstrated later.

The `TInstantExposer` can be used to display a single `TInstantObject` or a parts or `References` attribute of a single `TInstantObject`. In fact, it can be used to represent any `TCollection` or `TList` descendent as a dataset, as long as the members of the collection or list have published properties. This makes it a very powerful component.

The `TInstantExposer` class can work in one of 2 modes, determined by the `Mode` property:

amContent in this mode, a parts or references attribute of an object is the source of the data. each object in the parts or references list is a row in the dataset. The attribute which should be examined is determined by the `Containername` property. For example, in case the addresses of a `TContact` instance should be exposed as a TDataset, the `Subject` property of the `TinstantExposer` would be set to the `TContact` instance, the `ContainerName` would be `Addresses`. The `ObjectClassName` would be set to `TAddress`, the class of 1 element in the list of addresses.

amObject in this mode, the subject is treated as an object (or list of objects) which should be displayed: in case of a list, the

In the `amContent` mode, the following additional `Options` can be used:

eoNotDisposeReferences if this is specified, referenced objects of references attributes are not disposed when deleting the Subject of the exposer.

eoDeferInsert By default, objects that are contained within another object (i.e. parts or references) are immediatly inserted in the owning class as soon as a new row is inserted in the dataset. If this option is on, then the objects are only inserted in the owning class when the row is posted.

These options are not valid in `amObject` mode, or for a `TInstantSelector`.

The `TInstantExposer` can be used in a Master-Detail relationship: setting the `MasterSource` to a `DataSource` instance which is coupled to a `TInstantSelector` or `TInstantExposer` will fetch the `Subject` (the object which is being edited) from the datasource.

Armed with these objects, an application can be programmed.

7 The contacts application

In fact, the contacts application can be programmed much as one would program a traditional database application. It's not necessary to choose a particular kind of application, any Delphi application is suitable. So a new project can be started, with a main form much as in the traditional approach: it has a menu and a large grid which will display the contacts. For good measure, a DBNavigator component can be added. After this (this could be done first as well), the model can be defined in the InstantObjects model explorer. Once the model is made, a database for it should be built. (a Firebird database was created.)

In good Delphi tradition, all datasets and related routines will be separated out in a datamodule (`ContactsDataModule` in unit dmContacts). On this datamodule, the following components are dropped:

UIDB a `TjvUIBDatabase` component. This is the UIB database connection component. The `Databasename` should be set to the name where the database resides - this should be the same database as the one which was built when the model was defined.

ICContacts a `TInstantConnector` descendent: `TInstantUIBConnector`. It's `Database` property should be set to `UIDB`.

ISContacts a `TInstantSelector` which has the following command:

```
SELECT * FROM ANY TCONTACT
```

this will fetch a list of all contacts. The `foObjects` option should be set.

IEAddresses this is a `TInstantExposer` instance which is coupled with it's `MasterSource` property to `ISContacts`. Its `ObjectClassName` property is set to `TAddress` and its `ContainerName` is set to `Adresses`. The `Options` property should include `eoSyncEdit`.

ISCountries is a simple `TInstantSelector` with command

```
SELECT * FROM TCOUNTRY
```

Here also, `foObjects` should be included in the fieldoptions

ISCities is a simple `TInstantSelector` with command

```
SELECT * FROM TCITY
```

again, `foObjects` should be included in the fieldoptions.

ISCitySearch is a simple `TInstantSelector`, again with command:

```
SELECT * FROM TCITY
```

and again, `foObjects` should be included in the fieldoptions. This dataset will be used to define the fields.

For all these datasets, persistent fields can be created in the fields editor; All display properties can be set; No database connection is needed for this: InstantObjects can retrieve all needed information for this in the model.

With this, all is ready to code the forms. For the main form, there is not much to code, all that needs to be done is to hook up the grid to the `ISCOntacts` dataset.

Furthermore, a method to open (and close) the database connection should be made:

```
procedure TMainForm.OpenDatabase;
begin
  ContactsDataModule.ICContacts.Connected:=True;
  ContactsDataModule.ISContacts.Open;
end;
```

This method can be called from the `OnShow` event handler of the form.

To maintain the list of cities and countries, 2 menu entries are made under the 'System' menu. The `OnClick` handler of these menu entries have almost the same code: they open a new form (modal) which shows the list of countries and cities:

```
procedure TMainForm.MICountriesClick(Sender: TObject);
begin
  With TCountriesForm.Create(Self) do
    try
      ShowModal;
    Finally
      Free;
    end;
end;

procedure TMainForm.MICitiesClick(Sender: TObject);
begin
  With TCitiesForm.Create(Self) do
    try
      ShowModal;
    Finally
      Free;
    end;
end;
```

The countries and cities forms hold no surprises. In fact, they can be copied almost entirely from the traditional contacts application. All that must be done is to set the datasource property of the grids to the correct datasources on the datamodule.

In the cities form, however, a problem which also existed in the traditional approach appears: in the grid showing the cities, a way must exist to choose the country in which the city is located: obviously, entering the ID of a `TCountry` object will not do. In the traditional approach, this was solved with a lookup field: Delphi then knows how to display a dropdown with the elements from the lookup dataset.

For InstantObjects, the same technique can be applied. It also reveals the reason why the foObjects option had to be specified in the ISCities and ISCountries InstantSelectors: In the ISCities a lookupfield is created (let's call it DisplayCountry) which uses as the KeyFields property the Country field, and has as LookupKeyField the Self field from the ISCountries dataset. Both these fields are included thanks to the foObjects fieldoption. The LookupResultField is set to Name, the name of the country.

By displaying the DisplayCountry field instead of the Country.Name field in the grid with fields, the grid allows to select a country from the list of countries: This is completely similar to the technique used in the traditional database programming. Note that it does violate the idea that the data layer and GUI layer should be separated.

Double-clicking in the grid with contacts should open a new form which allows to edit contact data and which allows to enter several addresses for the contact person. The same action can be performed from a menu, with the following simple method:

```
Procedure TMainForm.ShowContacTDetails;

begin
  If Assigned(AddressesForm) then
    begin
    AddressesForm.show;
    AddressesForm.SetFocus;
    end
  else
    begin
    AddressesForm:=TAddressesForm.Create(Application);
    AddressesForm.Show;
    end;
end;
```

The addresses form looks exactly the same as the one in the traditional approach, and is hooked up to the ISContacts dataset on the datamodule. The grid is hooked up to the IEAddresses dataset, and will show 1 row per address of the contact person.

To select the city of an address, the same technique could be used as for the reference to a country in the cities form: using a lookup field. However, this time the problem will be solved slightly different: this will allow to demonstrate 2 important aspects of the fact that the application is actually programmed using objects, and is not simply a database application.

For the column of the City.Name field, the Buttonstyle property is set to cbsEllipsis. This has the following effect (standard for a Delphi DBgrid): If the City field is edited, a button with an ellipsis on it will be displayed. When the button is clicked, the DBGrid's OnEditButtonClick event handler is called. In the event handler, the following code is entered:

```
procedure TAddressesForm.GAdressesEditButtonClick(Sender: TObject);
begin
  If (GAdresses.SelectedField=
     GAdresses.DataSource.Dataset.FieldByName('City.Name')) then
    ShowSelectCityForm;
end;
```

The ShowSelectCityForm will show a new form in which a city can be selected, or a new city can be defined if the city is not yet present in the database. The procedure is coded as follows:

```
procedure TAddressesForm.ShowSelectCityForm;

Var
  CA : TAddress;

begin
  ContactsDataModule.DSAddresses.Edit;
  CA:=ContactsDataModule.IEAddresses.CurrentObject as TAddress;
  With TSelectCityForm.Create(Self) do
    try
      if ShowModal=mrOK then
        CA.City:=SelectedCity;
    Finally
      Free;
    end;
end;
```

First, the dataset is set in editmode (this could cause an insert in the dataset). Then the address object which is being edited, is fetched and stored in a local variable. The `TSelectCity` form is created and shown. If the user closes that form using the OK button, then the `SelectedCity` property of that form is stored in the `City` property of the currently edited Address object.

This shows that

1. The datasets do actually represent objects which are actually being edited. It is easy to forget this.
2. The objects can be manipulated without using the dataset mechanism. This explains also why the `eoSyncEdit` option had to be set for the `IEAddresses` dataset: Otherwise, the changed city would not be visible at once in the dataset ! (it's in edit mode)

The `OnClose` handler of the AddressForm contains the following code:

```
procedure TAddressesForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action:=caHide;
  ContactsDataModule.ISContacts.PostChanges;
end;
```

The `PostChanges` is needed for the ISContacts to post the changes to the addresses, which are stored in a `TDatasetField` of ISContacts: While the Contact data itself was not changed, the addresses have changed. The call to `PostChanges` makes sure that these changes are also posted to the database.

Remains to code the `TSelectCityForm`. This form contains a page control with 2 tabsheets: one which allows to search for a city based on name and zip code, and the other which allows to create a new city. When the user clicks OK, the currently active tab determines which city is returned.

The code for the select tab is quite simple. It contains 2 edit fields, a 'Search' button, and a DBGrid. The DB grid is coupled to the `ISCitySearch` dataset on the datamodule. When the user clicks the `Search` button, the following code is executed:

```
procedure TSelectCityForm.BSearchClick(Sender: TObject);
begin
  DoSearch;
end;

procedure TSelectCityForm.DoSearch;

Var
  S : String;

begin
  With ContactsDatamodule.ISCitySearch do
    begin
    Close;
    Command.Clear;
    Command.Add('SELECT * FROM TCity');
    Command.Add('WHERE');
    S:='';
    If (EZip.Text<>'') then
      S:='(Zip like ''%'+EZip.Text+'%'')';
    If (EName.Text<>'') then
      begin
      If (S<>'') then
        S:=S+' AND ';
      S:=S+'(Name like ''%'+EName.Text+'%'')';
      end;
    Command.Add(S);
    Open;
    If Not (EOF and BOF) then
      begin
      GSearch.SetFocus;
      BSearch.Default:=False;
      BOK.Default:=True;
      end;
    end;
end;
```

This code is quite clear: depending on the values the user has entered, the `Command` property of the `ISCitySearch` selector is coded with a simple search mechanism. If the query returns a result, the focus is set on the grid (so the user can select a city in the resulting set) and the OK button is made default (allowing the user simply to hit the enter key to select his city).

The second tab allows to enter a new city, in case the search has not yielded any results. It contains no code, only some data-aware edit fields: one for a ZIP code, one for a name, and a lookup combobox to select a country. These 3 controls are hooked to a `TInstantExposer` instance: `IENewCity`, which is in `amObject` mode. It's `ObjectClassName` property is set to `TCity`.

When the user changes the active tabsheet, the following code is executed:

```
procedure TSelectCityForm.PCCityChange(Sender: TObject);
begin
  If PCCity.ActivePage=TSSelect then
    begin
    If Assigned(IENewCity.Subject) then
      begin
      IENewCity.Close;
      IENewCity.Subject.Destroy;
      end;
    end
  else
    begin
    IENewCity.Subject:=TCity.Create;
    IENewCity.Open;
    CBCountry.ListSource.Dataset.Open;
    end;
end;
```

If the active page is the `TSSelect` page, then the `IENewCity.Subject` property is examined. If it contains a `TCity` instance, this instance is destroyed. In the case the active page is `TSNew`, then the `Subject` property is filled with a newly created `TCity` instance, and the dataset is opened, together with the lookup dataset.

When the user presses the OK button, the following code is executed:

```
procedure TSelectCityForm.BOKClick(Sender: TObject);
begin
  if PCCity.ActivePage=TSSelect then
    FCity:=ContactsDatamodule.ISCitySearch.CurrentObject as TCity
  else
    begin
    IENewCity.PostChanges;
    FCity:=(IENewCity.Subject) as TCity;
    end;
end;
```

If the active page is the `TSSelect` page, then the currently selected city object is stored for later reference. If the active page is the `TSNew` page, the newly created city object is stored in the database by means of a `PostChanges` call, and is stored for later reference.

At that point, the form will close, and it's `SelectedCity` property (which reads from the `FCity` field) will be used to set the City attribute of the address being edited.

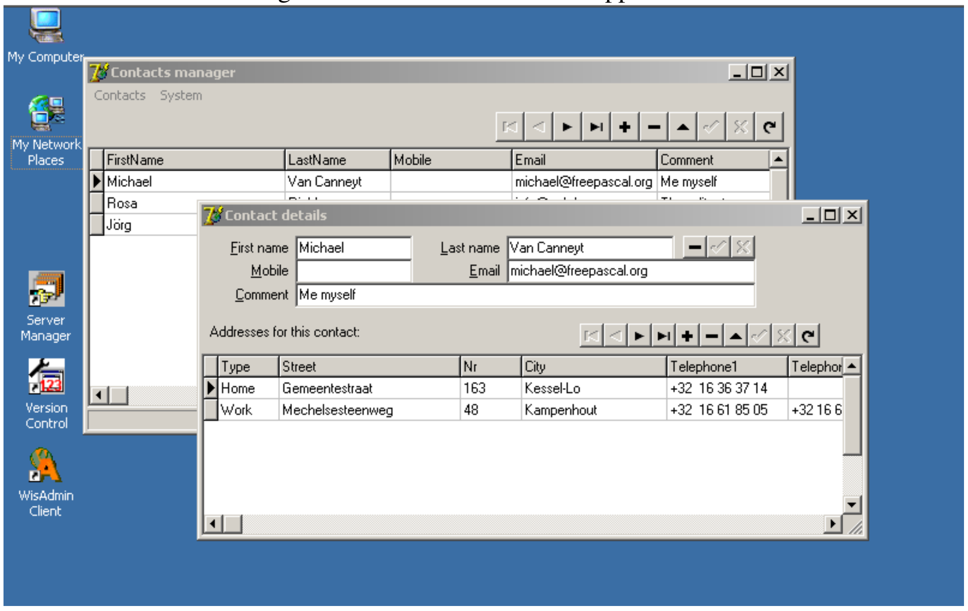
This shows that new objects can be created in code, and can be assigned to properties of other objects: the resulting changes will be saved in the database, even if everything is edited through the dataset exposers of `InstantObjects`.

With this, the contacts application is complete: it has the same functionality as the contacts application created with traditional methods. The end result is visible in figure 3 on page 18

8 Conclusion

InstantObjects is a quite lightweight Object Persistence framework. Nevertheless, it is fully functional, and is easy to understand; the gory details of persisting all objects are nicely hidden from the programmer. It's use of DB-Aware controls for the presentation layer makes it extremely easy to create GUI's for any model one wishes to create with it. It also puts a lot of 3rd-part controls at one's disposal. It's simplicity comes with a price: it currently lacks certain optimizations (loading partial property lists) which may cause some

Figure 3: The finished contacts application



	FirstName	LastName	Mobile	Email	Comment
▶	Michael	Van Canneyt		michael@freepascal.org	Me myself
	Rosa				
	Jörg				

First name	Michael
Last name	Van Canneyt
Mobile	
Email	michael@freepascal.org
Comment	Me myself

	Type	Street	Nr	City	Telephone1	Telephor
▶	Home	Gemeentestraat	163	Kessel-Lo	+32 16 36 37 14	
	Work	Mechelsesteenweg	48	Kampenhout	+32 16 61 85 05	+32 16 6

performance degradation when large amounts of data must be treated. The fact that it needs complete control over the database is easy, but can lead to problems when legacy data must be used. However, for someone starting with OPF, InstantObjects is definitely worth a try; any Delphi programmer should be very quickly up-to-speed with this nice tool.