UML programming with delphi: Bold for delphi

Michaël Van Canneyt

3rd December 2004

Abstract

One of the most useful extensions of Delphi has been around for some time and allows to use Delphi as a programming tool for implementing UML models. UML (Unified Modelling Language) is a high-level language to describe almost any software process or programming task in an Object Oriented way. Bold for Delphi transforms the model to a working program. In this (and possibly subsequent) articles, the use of Bold will be explored.

1 Introduction

UML (Unified Modelling Language) is a language to describe object-oriented software designs. It describes the application in functional terms (use-case diagrams), the classes involved in the application, together with the relations between the classes. There are also diagrams to describe the interactions between classes (using a timeline or not), component diagrams describing the various components (i.e. parts) of an application, and even deployment diagrams. All these provide different views of the application. In this article, the class diagrams will be considered, as that is what will be implemented in Delphi.

Bold for Delphi (now called ECO, for Enterprise Core Objects) is a way to realize UML models using delphi as the programming environment. Bold for Delphi comes with the Architect version of Delphi 7 - where it has version 4.0. Delphi 2005 comes with ECO, which is the successor of "Bold for Delphi". Since the author has version 7 of Delphi available, the terminology will be the one of Delphi 7. It should readily translate to ECO.

What does Bold do? Bold for delphi converts a class diagram to classes in object pascal. It creates also relations between the classes, and automatically performs persistence, i.e. writes the classes to disk. This can be in a database, a XML file, or can be done using a custom event mechanism. It further gives all tools necessary to manipulate the classes in a GUI fashion.

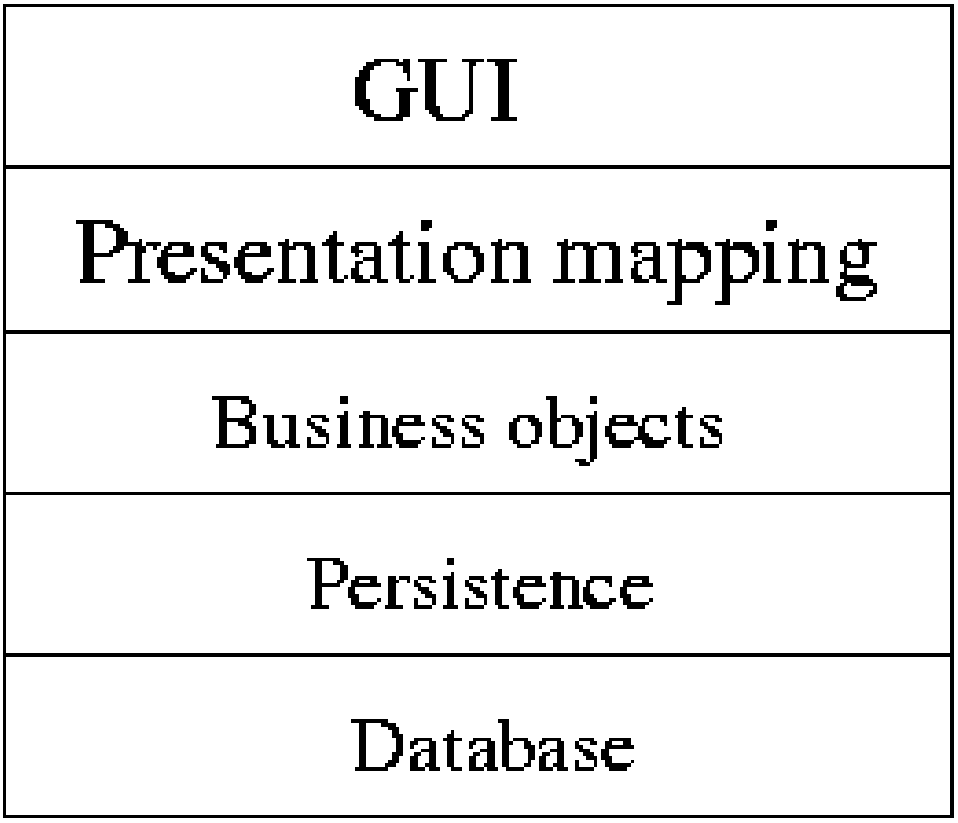
To do this, Bold for Delphi consists of a huge number of classes, separated into several layers, as shown in figure 1 on page 2.

GUI These are Bold-aware GUI controls. They are comparable to the well-known data-aware controls of Delphi, but instead of connecting them to a dataset, they are connected to the Bold model.

Presentation mapping This is a set of classes that allows to represent the various attributes in the Bold model in a GUI fashion. Mainly this means transforming attributes (and operations on attributes) to something which can be represented on screen, such as a string.

Business objects The business objects are all the classes as defined in the model: Bold generates the complete code for these classes in Object Pascal.

Figure 1: The various layers of Bold



Persistence To be able to store the state of the classes, Bold has a persistence layer. It takes care of all storage, and must be connected to some external storage mechanism (a database).

database The database is not per se a part of Bold. Bold only provides some in-between classes which allow the persistence layer to communicate with existing Delphi data storage mechanisms (IBExpress, DBX, ADO, BDE etc.)

The programmer normally only comes in contact with the GUI layer, and with the implementation of the business objects, if there are any methods that must be filled in. Bold takes care of the rest, and manages to hide this from the programmer. It can even take care of a large part of the GUI layer.

An important part of Bold is the OCL: the Object Constraint Language. It is part of UML, and is used (as the name indicates) to implement constraints, such as 'ISO code of Country is at most 3 characters, all uppercase'. It is used also to describe collections of objects and operations on objects. Bold uses it extensively, and the programmer will need to enter lots of OCL expressions in his application, because the expressions are used for many things:

- Class navigation: `Contact.AllInstances` represents the full list of contact instances.
- Access to attributes: `Contact.firstName`. They can be used to compute values such as `Contact.firstName+' '+Contact.lastname`
- Operations: `Contact.AllInstances->first` will position the current object on the first instance.

- Navigation through associations: `Address.InCity.InCountry` returns the `Country` instance that represents the country in which the address is located.

It is not possible to cover all aspects of OCL here, but the Bold installation comes with a copy of the official specifications. The language is easy to read and understand, so any programmer will quickly be quite familiar with it.

A note about the images: The images in this article were taken from the screen of the author. Bold (and in general Delphi) has problems with the 'Scaled' property of TForm, causing forms to scale wrongly on resolutions different from the resolution on which the form was designed. Therefore, some of the screenshots of the Bold editors will have parts of the screen 'cut off'.

2 Comparing traditional and model-driven programming

To illustrate the creation of an application using Bold more clearly, let's take a look at how a contact management application would be constructed in a traditional application, and how it is done using a model-driven approach. The application is a database application, as it is there where the strength of Bold lies; Bold can be used to program simple applications as well, but there would be little advantage to its use.

The functional analysis is the same for both approaches: What is needed is an application which manages a list of contact persons (just a first/last name, birthday), and to each contact person a series of addresses can be associated. Each address is identified by its kind (home, work, etc.). What is more, the cities and countries should be selectable through a list, to avoid double entry and differencies in spelling.

In a traditional application development cycle, one would e.g. start with a relational database design, and create 4 tables:

Country Containing e.g. the ISO code and name of the country.

Cities Containing cities as entered by the users. A foreign key establishes the relation with the country.

Addresses Containing all addresses. One foreign key establishes the relation with a city, and one establishes the relation with a contact.

Contacts A list of first and last names, and the birthdays.

How to create the primary keys and the foreign keys between the tables is a choice of the programmer:

1. The country can be identified by its ISO code, the city by its zip code and country, and the contact by the first/last name.
2. Another method would be to assign a unique ID (using some autoincremental value) to each record in each table, and use that as the primary key. A unique index can be used to ensure the uniqueness of the ISO code, ISO/Zip cobination, and first/lastname combination.

After the database is created, the application will be programmed. The details of the application depend on the choices made in the database model.

In a model-driven design, the development cycle is different. First, a model is created. It will consist of several classes:

Contact A contact person class. It will have 3 attributes: Firstname, lastname and birthday.

Country A country class, with 2 attributes: Name and ISO code.

City A city class, with 2 attributes: zip code and name.

Address An address class, with several attributes: Street, telephone, fax, mobile, email.

Then, the relations (associations) between the various classes are described:

1. Each address is associated to 1 and exactly 1 city.
2. Each city is associated to 1 and exactly 1 country.
3. Each address is associated to 1 and exactly 1 person. The association also specifies the kind of address.

In the last item, there is some choice: it could be that 1 address is related to different contacts (as in several people living in the same house, or working in the same company), thus avoiding duplication. However, it is a deliberate choice not to do so.

Note that the kind of address could also be specified in the address itself, but instead this attribute is specified in the relation. There is no pressing need to do this, but it allows to illustrate a point which is a marked difference with the relational-database design. To be able to set the 'kind' attribute in the association between city and contact, a separate class must be created to represent the association.

The above design can be made using several tools: Rational Rose or Modelmaker. Bold for delphi can import models made by these tools. However, this model can also be created in Bold for Delphi itself.

After the model is created, Bold will create a database (various components exist for this) and Bold will also create object pascal classes which are implementations in Object Pascal of the classes as described above.

All that is left to do is design the screens of the application. Bold for delphi comes with a lot of Bold-Aware components. They are the equivalent of DB-Aware components in traditional database programming. Bold even goes as far as creating complete forms to allow to edit the classes and their associations as defined in the model (something which even the Delphi data-entry form wizard cannot do). Strictly speaking, all that would be needed for the programmer would be 1 grid and a navigator. Bold can do all the rest.

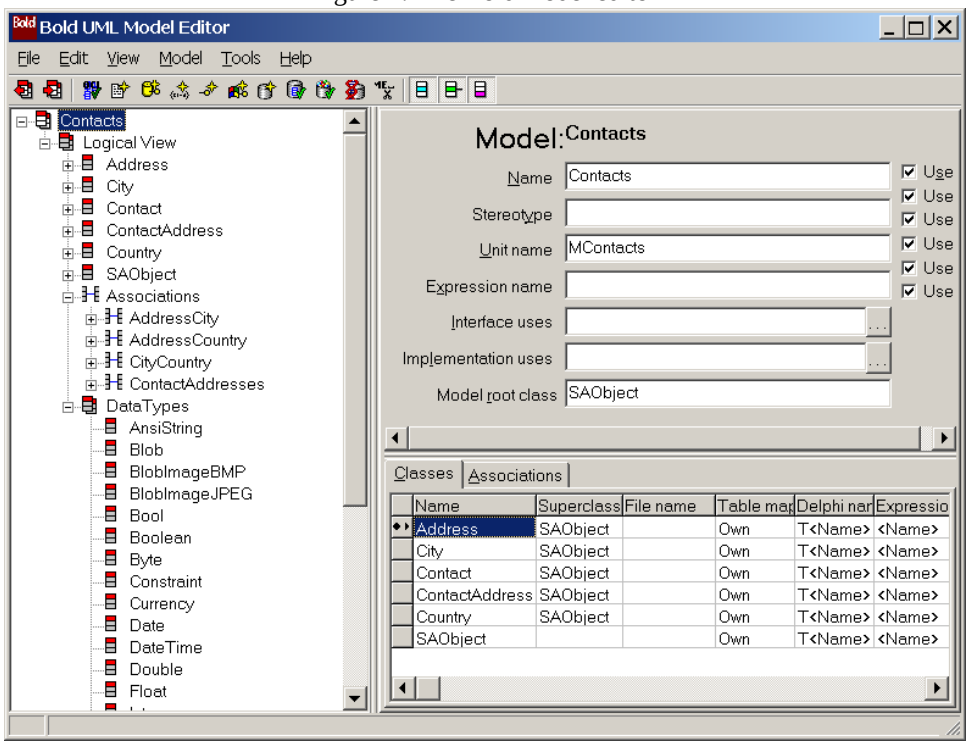
3 Creating a model in Bold

The next step is to create the above model in Bold. This can be done by implementing the model in some 3rd party tool (such as ModelMaker) and importing it in Bold, or it can be entered in the Bold Model Editor.

To create a Bold model, a `TBoldModel` component is needed. So a new datamodule is created, with name `ContactsDatamodule`, and it is saved in the file dmContacts. A `TBoldModel` component from the 'Bold handles' tab is dropped on the datamodule, and is named 'BMContacts'. Double clicking the component will bring up the Bold UML model editor.

The editor is shown in figure 2 on page 5. It has a left pane, which allows to browse through the model: all classes, associations between classes, and data types are shown in the tree. When available, attributes (properties) and operations (methods) of classes will also be shown in the tree.

Figure 2: The Bold model editor



The first thing to do is to give the model a name. To do this, the topmost node in the tree is chosen (for an empty model it is called `New_model`). On the right side, the model pane appears. In the 'Name' box, the name 'Contacts' can be entered. Note that all names for classes should be valid Delphi identifiers.

The 'Unit name' edit can be used to set the name of the unit in which to save all code which is generated by Bold (we'll use mcontacts for this). The 'Model root class' entry can be used to set the name of the root class: this class serves as the parent class for all classes in the model, unless specified otherwise. If need be, the names of units to be added to the `uses` clause in the interface and implementation sections can be set too. Bold will then add all units specified there to the uses clauses.

The root class (Initially called `New_Modelroot`) can be renamed. We'll name it `SAObject`. All classes will descend from this. Attributes may be associated to this class, for instance: any auditing information (such as "created by", "created on", "last modified on", "last modified by") could be defined here. If defined as persistent, the auditing information would be stored as well.

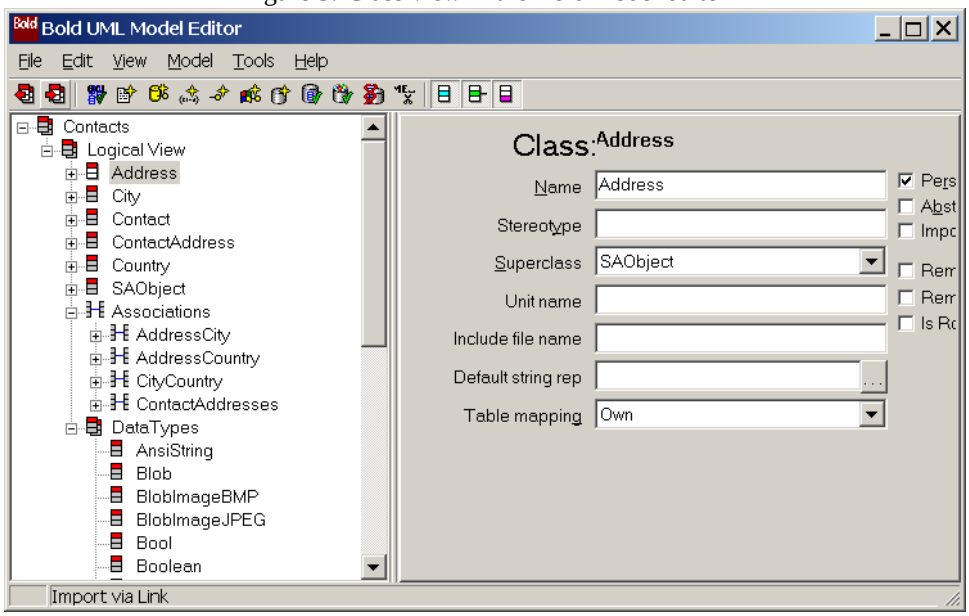
The root class can also be used to override the default behaviour of Bold methods in `TBoldObject`: the `TBoldObject` class is the parent class of the root class, and hence of all classes in the model. For instance, enforing constraints can be done in the root class. For the purpose of this article, nothing will be added to the root class.

The first class to be created is the 'Country' class. For this, 'Add new class' can be chosen from the context menu of the tree view. A new class node will be added, and the right side of the screen will switch to the class view, as in figure 3 on page 6. The following things are of interest here:

Name The name of the class.

Superclass The name of the parent class.

Figure 3: Class view in the Bold model editor



Unit name The name of the unit where the code for this class should be written to.

Include file name is the name of the include file where method implementations can be implemented.

Default string rep is the default string representation of this class: it is what will be shown by default in e.g. lookups, dropdown lists etc. The button to the right will open the Bold OCL editor. It can be used to create a correct OCL expression. In the case of the Country class, the default string representation could be

```
iso+' '+name
```

The editor will give an error if an erroneous expression is entered. By default, the value of the first attribute is used as the default string representation.

Table mapping this setting specifies where the persistent attributes of this class should be stored. This can be in a own table, or in the table associated with descendent of the class.

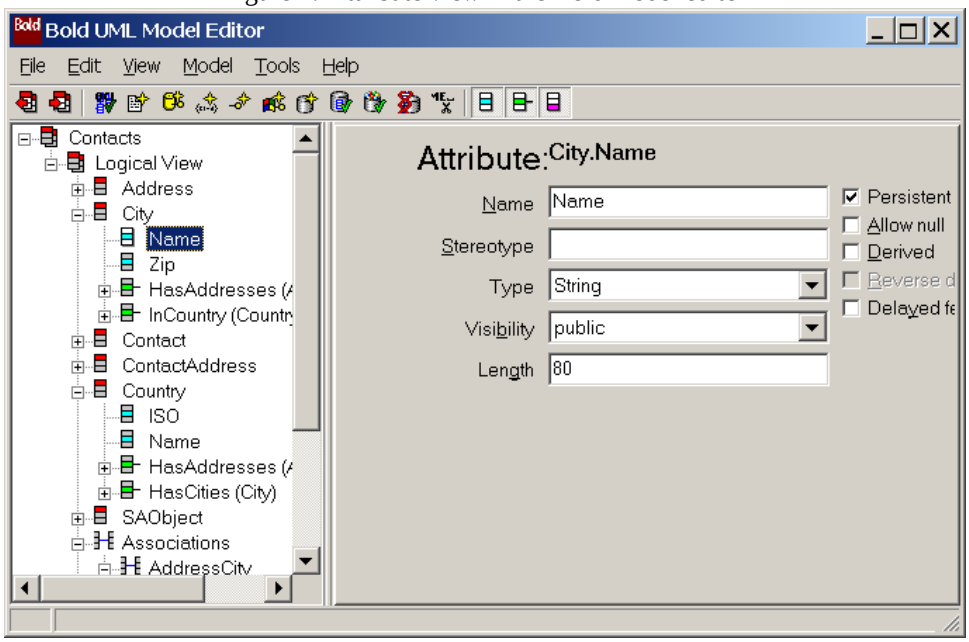
Most of the time, only the name, superclass and default string representation will be explicitly set.

The checkbox `Persistent` determines whether this class will be stored in the database or not. The `Abstract` option tells Bold that there will be no instances of this class, but that only instances of descendent classes will be created.

Now that the class is defined, some attributes (properties) can be added to it. If the class is selected in the navigation tree, the 'Add Attribute' menu item in the 'Model' menu is availabe. Clicking it will add a new attribute to the class, and the attribute editor is displayed. It looks like figure 4 on page 7. In the attribute editor, the attribute's definition is entered. This means that the following properties must be entered:

Name Name of the attribute. This should be a valid Delphi identifier.

Figure 4: Attribute view in the Bold model editor



Type The type. Most simple delphi types are available, but custom types can be added as well. They will be shown under the 'DataTypes' node. Note that no definition of custom types will be added to the code generated by Bold. These types must be made available by adding the units in which they are defined to the interface uses clause.

Visibility The visibility of this identifier.

Length this is only needed for a string type.

To the right of the attribute properties, there are some options which can be set:

Persistent if this is checked, the attribute will be saved to the database. If it is not checked, it's value will be lost when the application is closed.

Allow null if this is checked, the attribute is not required. By default, all attributes are required.

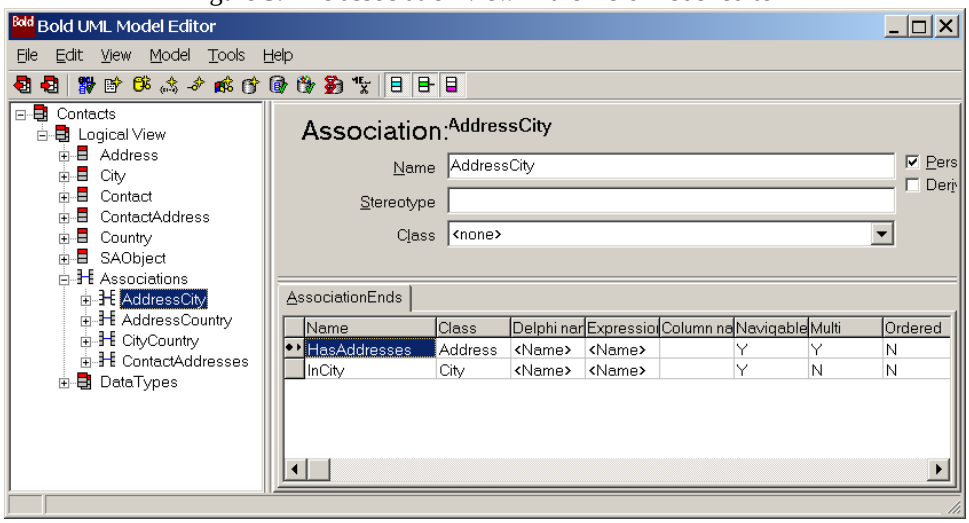
Derived A derived attribute is calculated, i.e. it cannot be set but is computed. It is comparable to a calculated field in a TDataset.

Delayed fetch if this is checked, then the value of the field will not be retrieved unless it is actually needed. This can be used to optimize loading of instances: by default all attributes of an instance will be loaded: this can cause heavy database traffic if lists of objects are loaded. If this option is set, then the attribute will be loaded only when it is actually needed.

Note that constraints on the attributes are not (yet) enforced by Bold, i.e. an object can be save with invalid attributes. The constraints must be checked manually. Methods can be entered in the same way using the 'Add Operation' menu.

With the above information, all the classes of our model can be added. What remains to be done is to define the relations between the various classes. This is done using the 'Add

Figure 5: The association view in the Bold model editor



association' menu item. When this menu item is used, the association panel is shown, as in figure 5 on page 8. The 2 relevant properties are

Name Each association has a name that describes the association.

Class An association can explictly be implemented through a class. If this is the case, the name of an exising class can be entered here.

In the case of the contacts model, the `ContactAddress` class is used to form the association between the `Contact` and `Address` class: it contains the `Kind` attribute, identifying the kind of address.

There are 2 options for an association:

Persistent This option determines whether the association should be stored in the database.

Derived If the association is derived, it is calculated; The calculation can be done in code, but can also be calculated from a OCL expression. In the Contacts model, the `AddressCountry` association is derived from the relation between `Address` and `City` and `City` and `Country`. The following OCL expression is used:

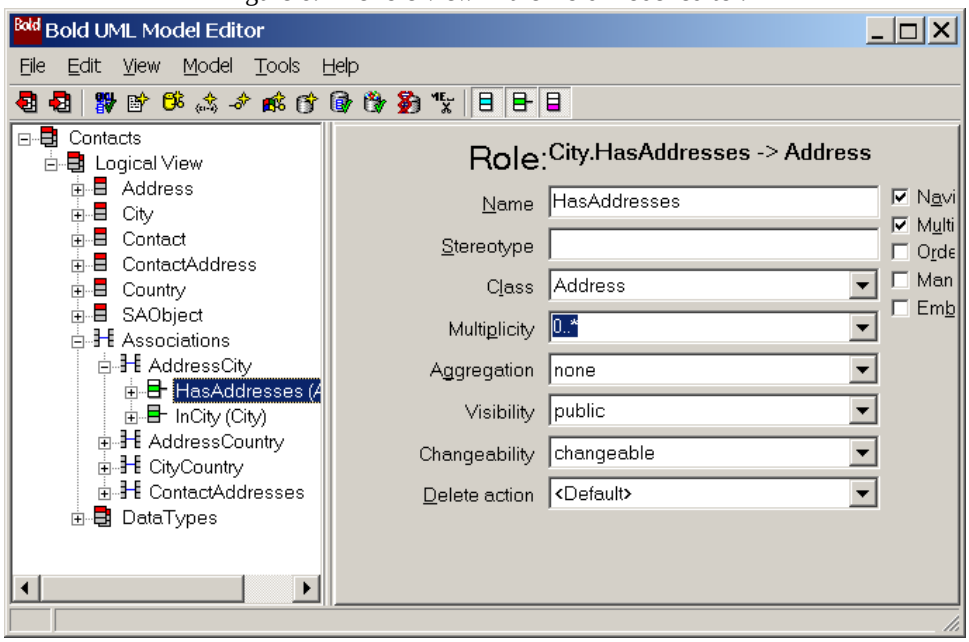
```
inCity.inCountry
```

The `InCity` part identifies the link between `Address` and `City` and the `inCountry` part identifies the link between `City` and `Country`. In terms of a relational database, this would correspond to two left joins on the foriegn keys between the various tables involved.

The association has 2 association 'ends', also called 'Roles': they define the end points of the association: They are defined in the 'Role' pane. Clicking one of the two association ends under the association node in the navigation tree will bring up the 'Role' panel. In the role panel, the end points of the association can be defined. The following parameters can be set:

Name The name of the endpoint or role.

Figure 6: The role view in the Bold model editor.



Class The class which this endpoint is connected to.

Multiplicity determines the multiplicity of this end of the NxM association. It can be any NxM value, but the dropdown will only present the most used values, of which there are 4:

- **0..1** zero or 1 connection.
- **1..1** exactly 1 connection. In this case, the connection is mandatory (the appropriate checkbox will be automatically checked).
- **0..*** any number of connections.
- **1..*** any number of connections, but at least 1.

Visibilty is the visibility of the relation.

Delete action is what should happen if the association is deleted: should the object be deleted, should nothing happen, or should an error occur (i.e. should deletion be forbidden).

Here again, there are some options:

Navigable Should it be possible to traverse the association from this end ?

Multi is automatically checked when the multiplicity can be more than 1.

Mandatory is automatically checked if the minimum multiplicity is 1.

Embed if this is checked, a field (property) will be added to the class which represents this end of the association. Note that this makes only sense for roles of multiplicity less than or equal to 1.

Note that for derived associations, the roles must also be defined.

When all classes, attributes, associations and roles have been entered, the model can be saved to disk. It is not necessary to save the model to disk, because the model definition is saved in the `TBoldModel` component that was used to start the editor. However, saving the model on disk allows to re-use the model for another application.

Before saving the model, it may be a good idea to check the model. This can be done with the 'Model Validation' menu item in the 'Tool' menu. Bold will then verify whether all needed parameters have been filled in, and whether the chosen options make sense (for instance, a role marked as 'multi' and 'Embed' does not make sense). The model editor can also be used to create a database for this model. This subject will be treated a little later.

4 Preparing to build the GUI for the model

Now that the model is defined, the application can be built. The `TBoldModel` component that was used to define the model is not enough to start working: it is only a location where the model is stored, and has no actual functionality.

To make a functional application, at least 3 more components are needed:

TBoldPersistenceHandle or rather, a descendent of this class. This class takes care of persistence of the objects created in the Bold model: It can store the objects to an XML file (the class `TBoldPersistenceHandleXML` does this), or can store them in a database (using `TBoldPersistenceHandleDB`). For the contacts model, a `TBoldPersistenceHandleDB` will be used (named BPHDContact).

TBoldSystemTypeInfoHandle this component (named BSTIHContact) manages all type information that is stored in the `TBoldModel` component. It is normally not necessary to explicitly reference this component, but it must be present to enable the following component to work:

TBoldSystemHandle This is the root component for the whole system: all references to objects classes, OCL expressions evaluations, start here: it is the 'root' of the "Bold object space". All other Bold handles (and there will be a lot) will directly or indirectly refer to this handle (named BSHContact).

In this list, one can note that each component is a 'handle': Bold makes extensive use of handles: They are anchor points to which objects or lists of objects are connected in some way. Many more handles exist. They all have a common point: they will all fall back to the `TBoldSystemHandle` component indicated above.

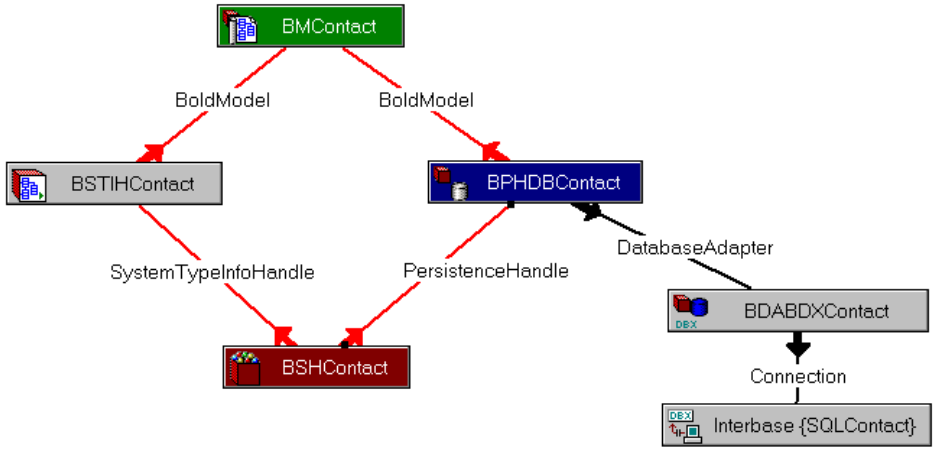
The 4 components are connected with each other as shown in figure 7 on page 11. On the diagram, 2 other components are shown. One is a `TBoldDatabaseAdapterDBX` (BDABDXContact), which connects the `PHDBContact` persistence handle to a database, in this case a `TSQLConnection` component (named SQLContact).

The DatabaseAdapter component hides the database specifics from the persistence handle: besides the used component (which connects to a DBExpress connection component), other DatabaseAdapter components exist (for ADO, IBX, BDE).

If the SQLConnection component is properly set up to connect to a database, then the Bold UML editor can be told to create a database for the model. This can be done using the 'Tools' menu in the model editor. Prior to generating a database, the Bold Model editor will validate the model, so no corrupt database can be generated.

A ready-made and partially filled Firebird database comes with the sources for this article. Re-creating the database in the existing database will destroy all data, Bold will warn if this is about to happen.

Figure 7: The 4 base components for the application



After the database has been created, the Object Pascal code for the classes must still be generated. This also can be done from the Bold editor, using the 'Generate Code' menu item in the 'Tools' menu. Bold will validate the model prior to generating code, so no invalid code is produced.

Now that the database is created, and the model code is generated, the model can be activated. To do this, the SQL connection component must be set to 'connected', and the system handle must be set to 'Active'. It is also a good idea to set the 'Autoactivate' property to 'True': if the model becomes inactive for some reason during the design fase, it will be activated when the datamodule is created at run-time.

Everything is set to create the application: The main form of the application is made a MDI parent form, and 3 MDI child forms will be created:

CountryForm Which is a simple form to manage the list of countries.

CityForm A second simple form which manages the list of cities.

ContactsForm This is the actual form to manage the contacts.

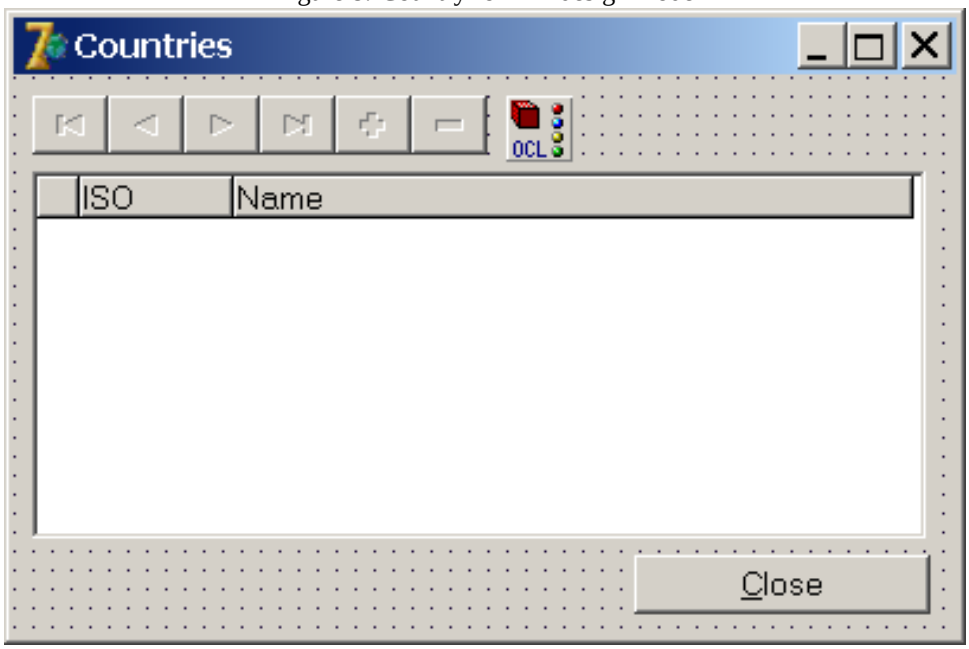
In the main form, a main menu is placed, with 3 menu items on the 'Maintenance' menu: one menu item per form. Each menu item will activate the approriate form.

5 Simple lists in bold: The country form

The first form, `TCountryForm` is very simple. From the 'Bold Handles' tab of the component palette, a `TBoldListHandle` is dropped on the form, and named `BLHCountries`. Then, a `TBoldNavigator` and `TBoldGrid` are dropped from the 'Bold controls' page on the component palette, and they are connected to the list handle through their `BoldHandle` property. The list handle must be connected to the bold system handle on the datamodule: to do this, the dmContacts unit must be added to the uses clause, after which the `RootHandle` property of the `BLHCountries` list handle can be set to `ContactsDatamodule.BSHContact`.

These actions are quite similar to the actions that are performed when creating a data-entry screen in a traditional application: the listhandle takes the place of the TDataset, and the BoldSytemHandle takes the place of the database connection component. Instead of setting a SQL property, the list handle gets an OCL expression in its `Expression` property:

Figure 8: Country form in design mode




```
Country.allInstances
```

This tells the list handle that it should retrieve all instance of the `Country` class.

The context menu of the grid can now be used to create some columns in the grid: choosing 'Create Default columns' will populate the grid with a column for all simple attributes in the `Country` class.

Finally, a button can be added to close the form.

The result should look more or less as in figure figure 8 on page 12.

Now, by default, Bold does not save modifications to classes to the database. If a class was modified during the run of a program, and the program is closed, then Bold will raise an exception, saying that there are still unsaved modified ('dirty') objects. To avoid this, the following code is added to the 'OnClose' handler of the form:

```
procedure TCountriesForm.FormClose(Sender: TObject;
                                   var Action: TCloseAction);
begin
  Action:=caFree;
  With ContactsDatamodule.BSHContact do
    If Active then
      System.UpdateDatabase;
end;
```

The 'UpdateDatabase' method will save any pending changes to the database. By calling this method when a form is closed, we make sure that all data which was modified when the form was open, is saved. However, if multiple forms were open and modifications were made, then these modifications will also be saved: all objects in the application which are marked 'dirty' will be saved. It is possible to save only selected objects, but the discussion of that technique is left for a later contribution.

6 Extending the grid with dropdown lists: the cities form

A similar form can be made for the Cities class. Here again, a BoldNavigator, BoldListHandle and Grid are used, and connected to each other. The BoldListHandle gets the following expression:

```
City.allInstances->orderby(name)
```

This will order the list of cities in the grid, by their name attribute. The problem with the above is that the result of this expression is 'Immutable', i.e. cannot be modified. In particular, no cities can be added to the list. To solve this, there is the `MutableListExpression` property: it should contain an expression which results in a 'mutable' list of (the same) objects, but which can be added to. This property can be set to

```
City.allInstances
```

And cities can then be added to the list.

Creating the default columns in the grid will create columns for all attributes, except for the `InCountry` role. This must be added manually to the grid columns. To do this, first a second BoldListHandle must be dropped (`BLHCountries`), with the following `Expression` property:

```
Country.allInstances->orderby(iso)
```

Now, we can add a new column to the grid. This is done in the columns editor. For a grid column, the `BoldProperties` property determines what is shown in this column of the grid, in this case the 'inCountry' expression. This will show the country which the city is connected to. It will use the 'DefaultRepresentation' expression of the City class to display.

To be able to set the country in the grid, The `LookUpHandle` property of the column can be set to the `BLHCountries` list handle. The `LookUpProperties` property can then be used to control the lookup. The 'Expression' subproperty can be set to

```
iso+' '+name
```

Which will display the iso code and name of each country in the dropdown list of the 'InCountry' column in the grid. Now the form is ready to go.

7 Drag and drop, and automatic class editor forms

Now that we have 2 forms ready to go, it is possible to show one of the advantages of Bold. The first one is the possibility for Bold to create class editor forms, i.e. a form which allows to edit a class. To enable this, a 'TBoldPlaceableAFP' component can be dropped on the datamodule, or the `BoldAFPDefault` unit may be added to a uses clause of the program.

After doing this, a double click on any of the Bold grids will open a form which allows to edit the current bold object displayed in the grid.

For the MDI contacts program, we add a menu to the main form of the program, and add 4 menu items to the first menu (Maintenance). The last menu item quits the program, all other menu items (Countries, Cities and Contacts). Each menu item is connected to an action, which will open the corresponding form when executed, as in the following code:

```
procedure TMainForm.ACountriesExecute(Sender: TObject);
```

```
begin
  With TCountriesForm.Create(Self) do
    Show;
end;

procedure TMainForm.ACitiesExecute(Sender: TObject);
begin
  With TCitiesForm.Create(Self) do
    Show;
end;

procedure TMainForm.AQuitExecute(Sender: TObject);
begin
  close;
end;
```

Compiling all this, the program can be run. The countries form can be opened, and some countries can be entered in the list. When this is done, the list of cities can be opened, and some cities can be added too. Note that the list of countries is immediatly available in the Cities form, and that when a country is added, it is readily visible in the Cities form.

Bold for Delphi will immediatly propagate a change to a class throughout the whole application. In a traditional database application, this would need a manual refresh of the list of countries in the Cities form, and would require careful management of transactions in the database. Bold takes care of this automatically.

Now, when a row in the Country grid is double clicked, bold opens the default class editor form, as in figure 9 on page 15. This editor form allows to completely edit the Country class that was double clicked on, including any relations that it may have. With some trouble, this could also be achieved in a traditional database application. What would be very hard to accomplish is Bold's capability to open any number of editor forms for any of the country classes shown in the grid: Bold has no concept of 'current record' in a list as it exists in the TDataset of a traditional Delphi database applications. What is more, the edits made in the editors will immediatly be reflected in all places where the class is shown.

A last point which is worth mentioning about the default editor form is the drag point which it contains: starting from the drag point, the class can be 'dragged' to any grid to be added to the grid, thus establishing a connection between what is shown in the grid and the class being dragged. e.g. one could drag a city to a country grid, and the City's `InCountry` role would be updated to reflect the new country it is in. The same is true for any 2 grids.

By default, the editor forms will also display a 'History' tab. This can be used to show the history of the class. This feature will be discussed in a future contribution, for now the display of the 'History' in the default class editor forms will be disabled by adding the following line to the program source:

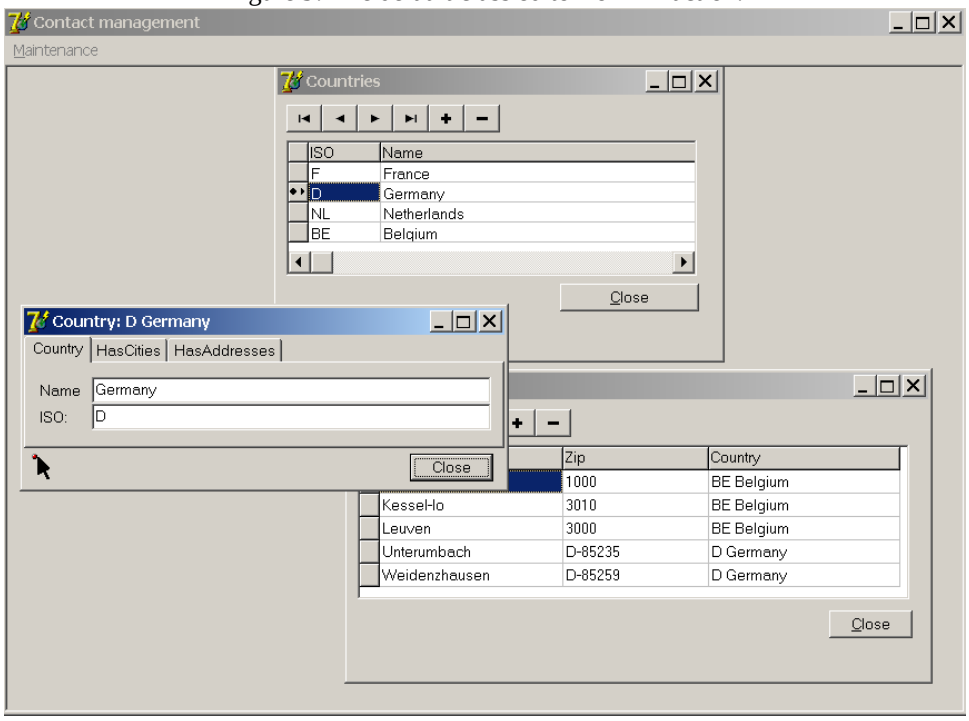
```
  BoldShowHistoryTabInAutoForms:=False;
```

Note that this variable is in the BoldAFPDefault unit, so it should be added to the used clause.

8 Other editing controls: the contacts form

There is of course more to Bold than just grids and navigators. The `Contacts` form will contain some extra Bold-Aware controls, such as a `TBoldEdit` and a `TBoldComboBox`.

Figure 9: The default class editor form in action.



The contacts form is shown in figure 10 on page 16. It contains 2 grids, 2 navigators, and 2 list handles. The first list handle (`BLHPersons`), is linked to the `BSHContact` Bold System handle, and has the following expression:

```
Contacts.allinstances
```

Which means it will display all contact persons. The second list handle (`BHLAddresses`), has its `RootHandle` set to the `BLHPersons` list handle. Its expression is

```
adresses
```

This expression will be evaluated relative to the `RootHandle` of the `BHLAddresses` list, i.e. relative to the current element in the Contacts list. As a result, when the active element in the contacts list changes, the addresses expression will be re-evaluated, and will display the list of addresses of the new current Contact class

Obviously, this mechanism is similar to the master-detail relations that can be made between various TDatasets in traditional database programming.

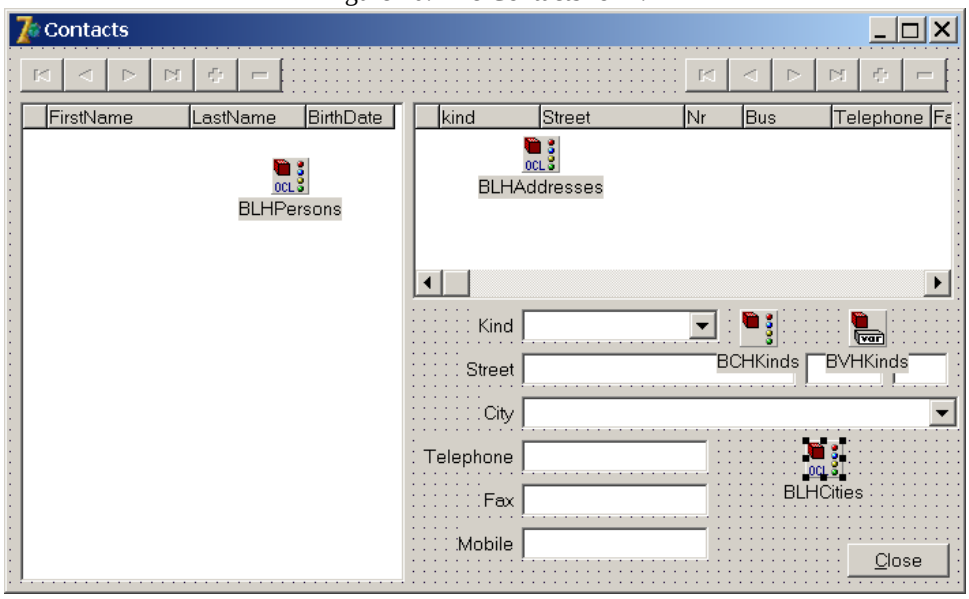
The grids can be filled with the default columns. Note that the 'Kind' attribute of the association between `Address` and `Contact` is not added to the grid. It is stored in the `ContactAddress` class, and this is not part of the expression entered in the list handle.

To be able to display the address kind in the grid, a new column must be added to the grid. The `Expression` property of the column must be set to

```
contactAddress.kind
```

It is obvious what this will do: for the displayed address row, the `ContactAddress` class' `Kind` attribute will be fetched and displayed. Similarly, the city and country of the address can be added to the grid.

Figure 10: The Contacts form.



Now some edits for the address can be placed on the form. The `Kind` and `InCity` attributes will be entered by means of a combobox, and the other attributes by simple bold-aware edits. Each of the controls is linked via its `BoldHandle` to the `BLHAddresses` list handle, similar to the way a DB-aware control is linked to a dataset via their `DataSource` property. The `BoldProperties.Expression` property controls which attribute the control displays and edits (similar to the `DataField` property of a DB-aware control). figure 10 on page 16.

The `TBoldCombobox` is a bit special. The TBoldCombobox is actually quite similar in use to the DB-aware `TDBLookupComboBox`: It needs a Boldhandle and expression for the attribute that it will edit, but also needs a `BoldListHandle`, similar to the `ListSource` property of the DB-aware combobox. The following are the key properties of the combobox:

BoldHandle The handle to the class being edited.

BoldProperties.Expression The expression that is shown in the combobox.

BoldSelectChangeAction What to do when a value is selected in the list. There are several possibilities. The action that is needed here is `bdcsSetValue`, which means that the selected element is used to set the value of an expression.

BoldSetValueExpression is the expression that will be set when the `BoldSelectChangeAction` equals `bdcsSetValue`. In the case of the address, this is `inCity`: When a city is selected from the list, the association is made between the address and the city. Since the Address' side of the association end is `inCity`, this is the expression that must be set. Obviously, not any expression can be entered in this property.

Note that there is no 'Items' property in the `TBoldCombobox` control. Yet that is what would be needed to set the 'Kind' attribute of the `ContactAddress` class using the `BCKind` combobox: a limited couple of distinct values. Instead, the Bold combobox always needs a handle. Luckily there are a couple of Bold handles which can be used to mimic a stringlist. The first of the two is a `TBoldVariableHandle`. This handle can be used to keep some static (or computed) value. This value can be of any type known to

the bold model: a string, an integer, a class, but, most importantly for the case needed here: A collection (a list). The type of the value must be set in the `ValueTypeName` property: This must be an OCL expression that results in a type. For the combobox, a collection of strings is needed to represent the various values for the `Kind` attribute, so the following OCL expression is used for the `ValueTypeName` property:

```
Collection(String)
```

The `InitialValue` property can then be used to enter the various values for the collection. For the case of the `kinds` attribute, the following strings will be entered:

```
Home
Work
Other
```

and the variable handle will be called `BVHKinds`.

The `TBoldVariableHandle` cannot be connected directly to the combobox, because it cannot be navigated. For this, a `TBoldCursorHandle` is used, which we'll name `BCHKinds`. This handle has a `RootHandle` property, which is set to `BVHKinds`. The `BCHKinds` handle can now be connected to the `BCKind` `BoldListHandle` property. By setting the `BoldSelectChangeAction` property to `bdscSetText`, the combobox will set the value of the `BoldProperties.Expression` property,

```
contactAddress.kind
```

to the value selected in the combobox.

Recapitulating, the `Kind` attribute is set by the `BCKind` combobox using the following properties:

BoldHandle refers to the current address in the address list handle `BLHAddresses`.

BoldProperties.Expression is the expression which refers to the `Kind` property.

BoldSelectChangeAction set to `bdscSetText`, will cause the combobox to set the `Kind` property to the selected text.

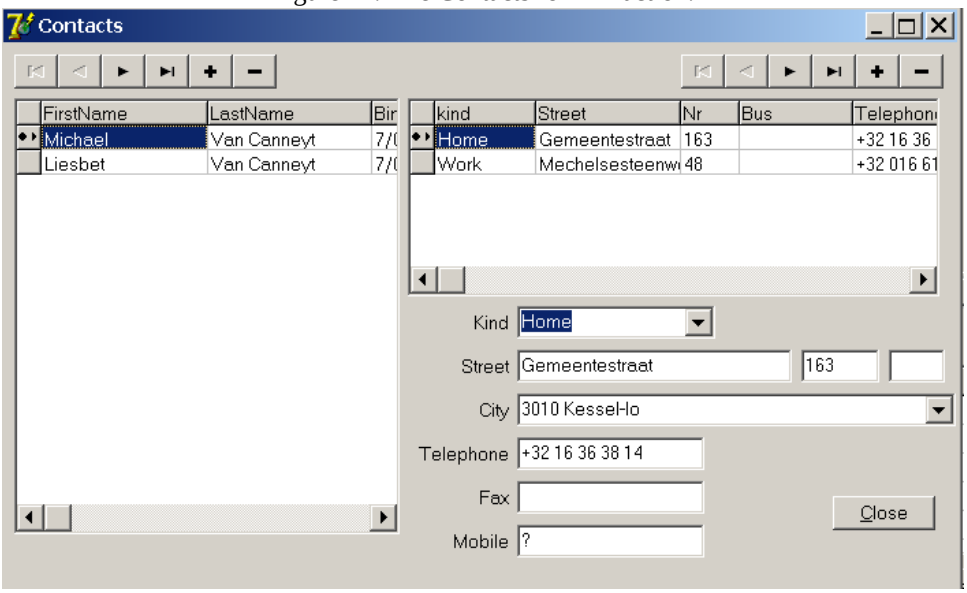
BoldListHandle refers to the `BCHKinds` handle, which is a cursor handle. The cursor handle allows to navigate through the various items in the collection defined in the `BVHKinds` handle.

This may seem a bit daunting at first, but will come natural after some time, and is in fact a very powerful construction. Also, it should be quite easy to create a `TBoldComboBox` descendent which has an `Items` property, which emulates all the above...

The `BCHKinds` handle can also be used to let the user select the `Kind` attribute in the addresses grid, by setting the `LookupHandle` of the `Kind` column in the grid to `BCHKinds`. Likewise can the `City` column be linked to the `BLHCities` list.

After all this is done, the contacts form can be used, and should look as in figure 11 on page 18. Note that while no contact person is entered, the addresses navigator is disabled: Bold has detected the absence of a contact, and hence no addresses can be added. Something that would need to be done manually in a DB-aware application, and which, if forgotten, would lead to errors.

Figure 11: The Contacts form in action.



9 Conclusion

Although the application is not yet much to look at, it is already fully functional, and has all the functionality required of it, with only very few lines of code. To achieve the same level of functionality, quite some more lines of code would be needed in a traditional Delphi database application. Bold for Delphi hides a lot of the complexities involved in maintaining the proper state of all classes, and offers a rich GUI which can be used to handle most, if not any, tasks at hand. Not all controls can be handled in the scope of an article, but the Bold sources come with a lot of examples. The Delphi/Simple/GUIControls subdirectory of the examples contain demo applications for most of the GUI controls that come with Bold, they are a valuable source of information about the possibilities of the controls.

In this article, only the surface has been scratched of Bold's capacities: Not a single line of code was added to the classes, no use was made of the versioning capabilities of Bold, the database update capacity was not used, and many other aspects have been left uncovered. These will hopefully be touched upon in future contributions.