RTTI controls in Lazarus

Michaël Van Canneyt

June 24, 2007

Abstract

Lazarus offers a set of controls which allow to display and set the published properties of any component: these RTTI Controls can be used to create property pages and wizards in the IDE, but they can also be used in real applications. Last and maybe most importantly: they can form the basis of a MDA toolchain for Lazarus.

1 Introduction

The Free Pascal compiler generates Run-Time Type Information for any descendent of a class that was compiled with RTTI information. Run-Time Type Information describes all published properties and methods of a class: This is the information which gets displayed in the Object Inspector of Lazarus.

The `TPersistent` class was compiled with RTTI information. All components descend from `TPersistent`, and have therefore RTTI information.

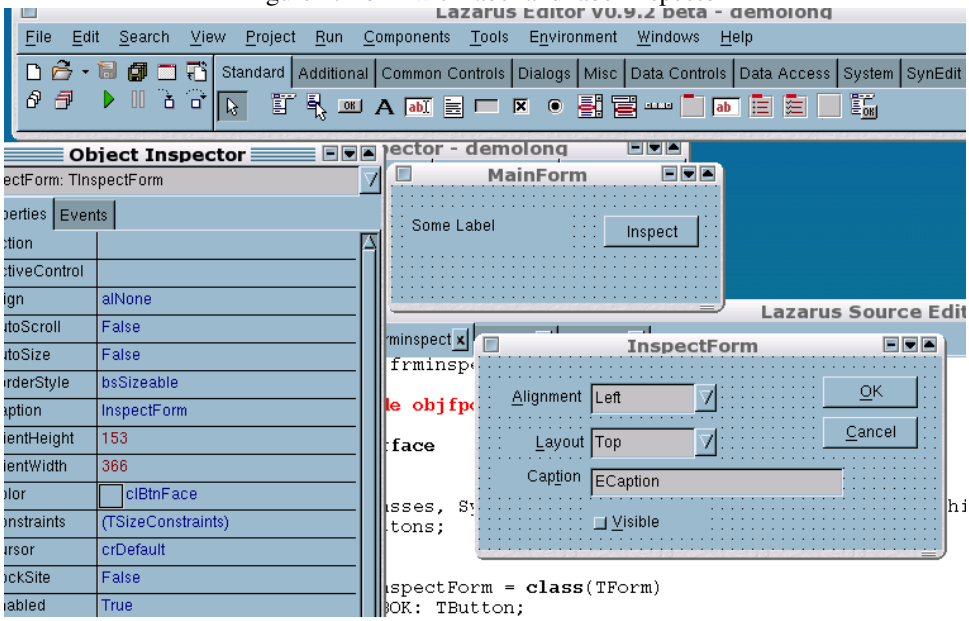
The RTTI information is not only available in design-time in the compiler, it can also be accessed during run-time using the standard typinfo and rttiutils units. The `SessionProperties` property of `TForm` uses this to save and restore the values of selected properties.

Manipulating RTTI information requires still a lot of manual coding. This has now changed with the implementation of RTTI controls. RTTI controls are regular controls, but they can be linked to a published property of a component: the control will then display the value of this property. For instance a `TICheckBox` (a RTTI aware checkbox) can be coupled to a `Boolean` property: the `Checked` property of the checkbox will reflect the property value. Checking or unchecking the checkbox will automatically set the value of the property. This is a simple example. The most advanced example is a complete grid which displays all properties of a component, just as the object inspector does.

In the remainder of this article 3 possible uses will be discussed of these controls:

- For component developers, these controls can be used to create property editors which need no code to display and set properties of a component. A `TPropertyForm` form and a `TComponentEditor` component will be developed to show how it can be used.
- For end-user applications, the components can be used to let the user set a published property of another component. This will be demonstrated by connecting a `TCheckBox` and `TEdit` to the 'Filtered' and 'Filter' properties of a dataset.
- Last but not least, the RTTI controls provide the beginning for a set of MDA (Model Driven Architecture) controls: they can be used to implement a set of model-aware components for Lazarus. This will be demonstrated by implementing a (limited) set of MDA components.

Figure 1: Form with label and label inspector



The RTTI controls are not included in the component palette of the Lazarus IDE by default: they must be installed. The package that does this is called the 'runtimetypeinfocontrols.lpk'. It suffices to install this package in the IDE to start using the controls.

2 RTTI controls for component designers

The RTTI controls are originally designed to simplify the creation of component editors: It is sufficient to drop them on a form, to set the relevant property names, and when the form is shown, the instance on which they operate is set. The components then take care of showing and modifying the component's published properties.

The demo1 application will show what the usual way is for implementing a wizard. A main form contains a button and a label. When the button is clicked, a second form is created in which some of the published properties of the label can be set, such as "Alignment", "Layout", "Visible" and "Caption". Both forms are shown in figure 1 on page 2

To do this, the comboboxes which will represent the "Alignment" and "Layout" properties, must be filled with values such as 'left', 'right' and 'center' - and in the correct order.

When the 'Inspect' button is pressed, the form is created and the label to be inspected is set:

```
With TInspectForm.Create(Self) do
  try
    TheLabel:=MyLabel;
    ShowModal;
  Finally
    Free;
  end;
```

This is nothing special and such code will be found in many component packages. In the TInspectForm, this action will trigger the loading of the properties in the various controls:

```
procedure TInspectForm.SetLabel(const AValue: TLabel);
begin
  Flabel:=AValue;
  If Assigned(Flabel) then
    LoadProps;

end;

procedure TInspectForm.LoadProps;
begin
  CBAlign.ItemIndex:=Ord(FLabel.AlignMent);
  CBLayout.ItemIndex:=Ord(FLabel.Layout);
  ECaption.Text:=FLabel.Caption;
  CBVisible.Checked:=FLabel.Visible;
end;
```

Note that for this code to work, the items in the dropdowns must be in the correct order.

When the user has set the needed properties, the OK button can be pressed, and the settings can be saved: procedure TInspectForm.SaveProps;

```
begin
  FLabel.Alignment:=TAlignMent(CBAlign.ItemIndex);
  FLabel.Layout:=TTextLayout(CBLayout.ItemIndex);
  FLabel.Caption:=ECaption.Text;
  Flabel.Visible:=CBVisible.Checked;
end;
```

The LoadProps and SaveProps can be suppressed by the RTTI controls, and the need to fill the comboboxes as well: the RTTI controls will do this all by themselves.

To demonstrate this, a `TPropertyForm` form will be made, together with a component editor, which can be used to invoke the property form in the Lazarus IDE. The idea is that this form can be used as a base for component editors: It can be invoked with the component editor popup menu. All that must be done is drop some RTTI controls on a form, and register the form in the IDE.

The `TPropertyForm` is very simple: has a property 'Instance', which will be set to the component that is being edited. It also has a panel at the bottom of the form, and this panel contains a 'Cancel' and 'OK' button. These buttons do as expected: Cancel closes the form without applying the changes, and 'OK' applies the changes that were done in the form, and closes the form.

The declaration of `TPropertyForm` is quite straightforward:

```
TPropertyForm = class(TForm)
private
  FInstance: TPersistent;
  FOnSetInstance: TNotifyEvent;
  procedure SetInstance(const AValue: TPersistent);
  Procedure SetLinks;
Protected
  FButtonPanel : TPanel;
  FOKButton,
  FCancelButton  : TButton;
  Procedure SaveProperties; virtual;
public
```

```
  Constructor Create(AOwner : TComponent);
  Procedure CreateButtonPanel; virtual;
  Procedure OnOKClick(Sender : TObject); Virtual;
  Property Instance : TPersistent Read FInstance
                                  Write SetInstance;
Published
  Property OnSetInstance: TNotifyEvent Read FOnSetInstance
                                       Write FOnSetInstance;
end;
```

In the constructor of the form, the panel and the buttons are created by calling the `CreateButtonPanel` method:

```
constructor TPropertyForm.Create(AOwner: TComponent);
begin
  Inherited;
  CreateButtonPanel;
end;
```

The creation of these buttons and panel is quite straightforward, the code will not be shown here.

After a `TPropertyForm` descendent is created, the component (or TPersistent) instance will be set. When it is set, all RTTI controls on the form will be connected to it. This is done with the following 2 methods:

```
procedure TPropertyForm.SetInstance(const AValue: TPersistent);

begin
  if (FInstance<>AValue) then
    begin
    FInstance:=AValue;
    SetLinks;
    If Assigned(FOnSetInstance) then
      FOnSetInstance(Self);
    SetPropertyCaption;
    end;
end;

procedure TPropertyForm.SetLinks;

Var
  I : Integer;
  C : TComponent;
  L : TpropertyLink;

begin
  For I:=0 to ComponentCount-1 do
    begin
    C:=Components[i];
    If IsPublishedProp(C,'link') then
      begin
      l:=TPropertyLink(GetObjectprop(C,'link',TPropertyLink));
      If l<>Nil then
        L.TIObject:=FInstance;
```

```
      end;
    end;
end;
```

The `SetInstance` property write handler will call setlinks to connect all RTTI controls on the form to the instance which should be displayed and edited in the form. This is done in the `SetLinks` method. This method simply checks all components on the form for the `Link` property. This property must be of type `TPropertyLink`. The `TIObject` property of this class is set to the instance which must be edited in the form. After the links have been set, the form's caption is set to an appropriate value in `SetPropertyCaption`. The interested reader can consult the source code that comes with this article to see how this is done.

The `TPropertyLink` class has the following published properties, which can be set in the object inspector of Lazarus:

TIPropertyName The property that the control should display. Note that the control should be able to display the property: A checkbox cannot display an integer value.

TIObject The `TPersistent` descendent whose property must be displayed.

Options a set which can contain the values `ploReadOnIdle` and `ploAutosave`: The option `ploAutosave` tells the RTTI control to automatically save the property when the user has changed it. The `ploReadOnIdle` value tells the control to re-read the value of the property when the application goes idle. This can be useful when changing one property also causes other properties to change.

AliasValues This can be used to provide alternate names for the values that the property can have: a combobox which displays an enumerated type will by default display the names of the enumeration values. This list can be used to change the displayed names. They should be specified in the form "value=alias". For instance, for the `Alignment` property of a `TLabel` component, this could be of the form:

```
taCenter=Center
taLeftJustify=Left
taRightJustify=Right
```

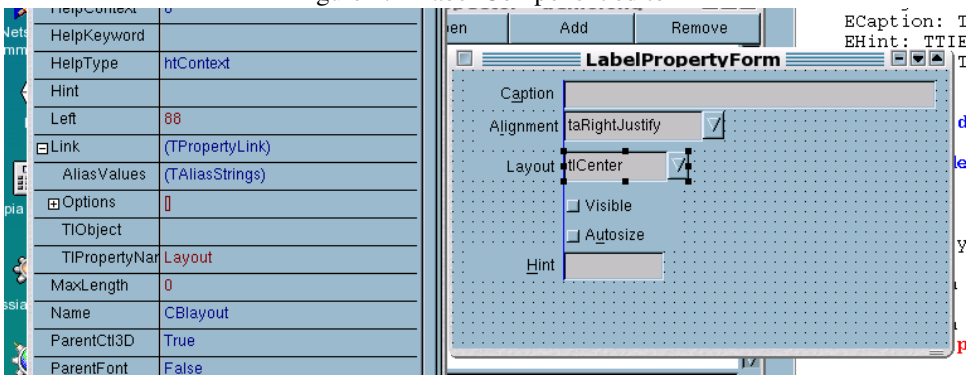
The combobox will display 'Left','Right' or 'Center', depending on what the value of the Alignment property is.

The property editor of the 'AliasValues' property can fetch all possible values and propose defaults.

When the PropertyForm is displayed, and the user presses the OK button, all properties must be saved before the form is closed. If the `ploAutosave` option is not specified, the properties have not yet been saved, which should be the case, otherwise the 'Cancel' button would have no effect. The saving of the properties happens similarly to the setting of the link:

```
procedure TPropertyForm.SaveProperties;

Var
  I : Integer;
  C : TComponent;
  L : TpropertyLink;
```

Figure 2: TLabel Component editor




```
begin
  For I:=0 to ComponentCount-1 do
    begin
    C:=Components[i];
    If IsPublishedProp(C,'link') then
      begin
      l:=TPropertyLink(GetObjectprop(C,'link',TPropertyLink));
      If l<>Nil then
        L.SaveToProperty;
      end;
    end;
end;
```

The `SaveToProperty` tells the RTTI control to save the value it is displaying to the property of the component that is linked to it.

Now the `TPropertyForm` class is all ready for use. A simple descendent will be made. For this, a normal TForm is created which contains all RTTI controls, as shown in figure 2 on page 6. For each RTTI control, the 'ploautosave' option is disabled, and the 'TIPropertyName' property is set to the name of the property that should be displayed. Note that enough free space is left to contain the panel with the buttons. This panel will be added in run-time, when the form is displayed, by the code in `TPropertyForm`.

The designed `TLabelPropertyForm` form descends of `TForm`. To make it a descendent of `TPropertyForm`, a small trick is used. In front of the `TLabelPropertyForm` declaration, the following line is inserted:

```
TForm = TPropertyForm;
```

And the '`frmproperty` unit is added to the uses list. This will trick the IDE into thinking that it is designing a regular `TForm`, but when the code is compiled, a `TPropertyForm` descendent will be made instead.

The `TLabel` component editor is ready. It can be integrated in the IDE. To make this easier, a component editor is created which 'knows' about TPropertyForm:

```
TPropertyFormClass = Class of TPropertyForm;

TPropertyPagesEditor = class(TComponentEditor)
  procedure ShowPropertyPage;
  procedure ExecuteVerb(Index: Integer); override;
```

```
  function GetVerb(Index: Integer): string; override;
  function GetVerbCount: Integer; override;
  Function PropertyFormClass : TPropertyFormClass ; virtual;
end;
```

When a descendent of this editor is registered, it will register a menu item 'Property Pages' in the IDE. When this menu item is activated, this editor will create a TPropertyForm descendent, set the instance property, and show the form modal. When the OK button is pressed, the component will be marked as 'modified' with the IDE.

The component is quite simple and it's code will not be shown here. To use it, a descendent must be made which overrides the `PropertyFormClass` function. It should return the class of the actual `TPropertyForm` that must be created.

For the `TLabel` component editor, this is particularly easy. A `TLabelPropertyPagesEditor` class is declared:

```
TLabelPropertyPagesEditor = class(TPropertyPagesEditor)
  Function PropertyFormClass : TPropertyFormClass ; override;
end;

function TLabelPropertyPagesEditor.PropertyFormClass:
   TPropertyFormClass;
begin
  Result:=TLabelPropertyForm;
end;
```

All that remains to be done is to register the component editor in the IDE:

```
procedure register;

begin
  RegisterComponentEditor(TLabel, TLabelPropertyPagesEditor);
end;
```

and it is ready for use. It is shown in figure 3 on page 8. Note that this is the only real code that must be created in order to have a working component editor: no other code is needed, unless some extra, specific code must be incorporated in the form.

All this is installed in a 'propform' package, which can be readily installed in the Lazarus IDE. It contains the following units:

frmproperty contains the `TPropertyForm` and `TPropertyPagesEditor` declaration and implementations.

frmlabelprop contains the `TLabelPropertyForm` form.

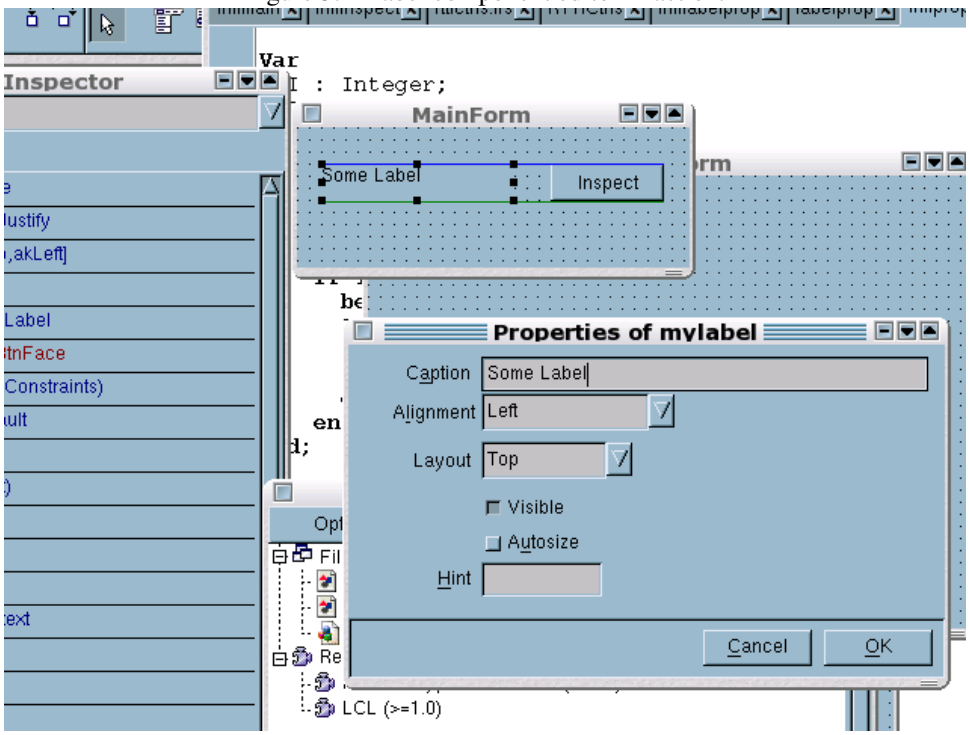
labelprop contains and registers the `TLabelPropertyPagesEditor` component editor.

By the time this article is printed, the `TPropertyForm` may be included standard in Lazarus, so descendents can be created from the 'New' menu.

3 Using RTTI controls in applications

In the previous section, the RTTI controls have been demonstrated by using them for what they were intended: to create component editors for the Lazarus IDE. However, they can

Figure 3: TLabel component editor in action.



also be used in end-user applications. In order to demonstrate this, the address book which was introduced in a previous article will be enhanced by adding filtering capabilities: The user will be given the opportunity to enter a filter on the list of entries in his address book: The `TDbf` dataset supports filter expressions, so this feature will be used.

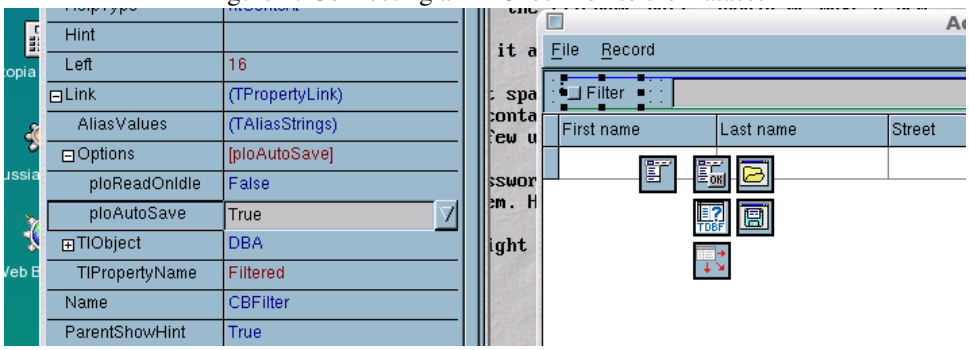
The traditional way of doing this would be to add an edit control and a checkbox: When the checkbox is checked or unchecked, the contents of the edit control is copied to the `Filter` property of the `TDBF` component:

```
Procedure TMainForm.CBFilterClick(Sender : TObject);

begin
  With DBA do
    if CBFilter.Checked then
      begin
      Filter:=EFilter.Text;
      Filtered:=True;
      end
    else
      begin
      Filtered:=False;
      Filter:='';
      end;
end;
```

In this code, `DBA` is the dataset, `CBFilter` is the `TCheckbox` component, and `EFilter` is the `TEdit` component.

Using RTTI controls, no code is necessary: A `TTICheckbox` is dropped on the form, it's

Figure 4: Connecting a TTICheckBox to the Dataset



`Link.TIObject` property is set to the `DBA` component, and the `Link.TIPropertyName` is set to 'Filtered', as shown in figure 4 on page 9. Then a `TTIEdit` component is dropped, coupled to the same `DBA` component, with the `Link.TIPropertyName` set to `Filter`. The `ploAutoSave` option is set for both components, and everything is ready.

Another case where RTTI controls can come in handy is the following case: often, a Edit control (EFile) is combined with a button (BSearch) and a Open (or save) Dialog (ODFile). When the user clicks the button, the dialog is executed to search for a filename and fill it in in the Edit control.

In the OnClick event of the button, the following code is executed:

```
Procedure TMainForm.BSearchClick(Sender: TObject);

begin
  With ODFile do
   begin
   FileName:=EFile.Text;
   If Execute then
     EFile.Text:=FileName;
   end;
end;
```

By using a RTTI TTIEdit control, which is connected to the FileName property of the ODFile dialog component, this code is reduced to:

```
Procedure TMainForm.BSearchClick(Sender: TObject);

begin
  ODFIle.Execute;
end;
```

Provided the `ploReadOnIdle` option is set for the edit control. A small demo application is provided to show this (it is called demo2).

Many other cases can be thought of: Checkboxes can be connected to the 'Visible' property of parts of the screen to show/hide certain parts without using additional code: The RTTI controls can be used to reduce the writing of code.

4 MDA programming with RTTI controls

In the previous 2 sections, RTTI controls were used for simple things, and the main benefit of these controls is to reduce the writing of code by letting the controls themselves act on properties of other components. However, they offer much bigger possibilities.

Some years ago, Borland acquired BoldSoft, and incorporated their 'Bold For Delphi' product in the Architect version of Delphi 7. Bold for Delphi was - and still is - a way to program using a Model Driven Architecture (MDA). It implements UML models in Delphi. UML stands for Unified Modeling Language: A language to describe application models and architectures. Using tools such as Rational Rose, ModelMaker, and Bold For Delphi, a model is converted to classes, and the application is implemented using these classes.

5 A jump-start in MDA and UML

MDA and UML is a large and complex field, which cannot be explored in the context of this article, also the author is by no means an expert in this field. Generally speaking, MDA driven programming breaks down in 2 parts: The first is designing a model, and the second is implementing the model in some development tool, such as VB, Delphi, Lazarus. The model does not say anything about the tool to be used. Instead, it simply describes the data structures in general terms, and processes: things that the application should finally do.

To describe this, UML is used: Unified Modeling Language. This language describes the data one wishes to treat in the program, and the things that should be done. For example, for an invoicing program, the UML model will describe what a customer is. It does this by describing the characteristics of 'Customer' (it is called 'defining an Entity'):

- Company name
- Customer ID
- VAT Number
- Address
- etc.

The model describes the restrictions on these various characteristics (properties), rules that should be checked. It would then continue to describe what an invoice is, it will descibe all properties of an item which appears in an invoice's details. It will also decribe the relation between the item and the invoice, and the relation between the customer and the invoice.

Once all entities and their relations are defined, the processes can be defined: how an invoice is produced: The company makes an offer, the client responds with an order, and the company makes a shipment, and finally sends an invoice.

Not a single line of code has been written. Only the objects, their relations, and what should be done with these objects is described. No mention is made of database storage (how to store the customers, invoices etc).

All these descriptions form the model: It can be defined using a tool such as ERWin or Rational Rose. From this model, a database and an application must be somehow be generated. For this, tools as ModelMaker and "Bold for Delphi" exist.

6 MDA: Generating the application

The second step in MDA is creating the application. For example using Bold for Delphi. Bold for Delphi will take all entities of the model, and will convert them to classes: For each entity (Customer), a corresponding Delphi class is generated (TCustomer). All defined properties will be added. The classes can then be used to create a program: Bold takes care of adding all code to store the properties of the classes in a database: it will also define all needed tables, foreign keys etc. The programmer no longer needs to be bothered by this: Bold does this for him.

To define a new customer in the application, one would simply do something like:

```
MyCustomer:=TCustomer.Create();
With MyCustumer do
  begin
  LastName:='Van Canneyt';
  FirstName:='Michael';
  CustomerNo:=3123;
  Address:='Gemeentestraat 163';
  end;
```

Once the customer is no longer needed, one does simply:

```
MyCustomer.Free;
```

And Bold for Delphi would have saved all properties in the database. If, later on, the customer's data is needed again, one simply recreates it:

```
MyCustomer:=TCustomer.CreateFromID(3123);
...
```

Properties can be changed (e.g. the address) and when it is freed, everything is stored again (automatically) in the database.

The programmer can now concentrate on the application itself. Bold offers a whole range of controls, which can be used to display entities' properties (e.g. a form describing customer data) or lists of entities: the list of items in an invoice. It does this by making the controls aware of the properties that can be displayed. This is the part where, for a tool such as Lazarus, RTTI controls jump in.

7 RTTI classes and Models

It is not the intention of this article to implement a modeling tool. Rather, it will be assumed that a modeling tool (such as Rational Rose) exists, that a model is available, and that it has been converted to classes by some other software, not discussed here.

Where this article starts is the controls needed to display the classes in the model. Suppose that an instance of `TCustomer` was created, as described above. How can it be shown on the screen ? The straightformward answer would be to copy all `TCustomer` properties to the various checkboxes, edit fields and dropdown lists on a form. When the user has finished editing, all properties are copied back to the instance. So showing it would be something like this:

```
With MyCustomer do
```

```
  begin
  Edit1.Text:=FirstName;
  Edit2.Text:=LastName;
  Edit3.Text:=Address;
  // etc.
  end;
```

Obviously, this is a lot of work. Same when the user is done editing, and all edits must be saved:

```
With MyCustomer do
  begin
  FirstName:=Edit1.Text;
  LastName:=Edit2.Text;
  Address:=Edit3.Text;
  // etc.
  end;
```

Again, a lot of work. If a property is added (e.g. birthday) , 2 lines of code must be added as well to be able to show it on the screen, and it must not be forgotten.

And here is the link with the RTTI controls: if all the relevant properties of the model are published, then the RTTI Controls can do this loading and saving automatically. In fact, they can be made 'Model aware', similar to the way the Bold controls are 'model aware'. How to do this, is described in the below sections.

8 Model classes

To simulate a complete set of MDA classes, 2 basic classes will be introduced which represent the model. They are far from complete, they just serve to illustrate the ideas for the RTTI controls, but they are functional. The following 2 classes are used:

TEntity This base class represents an entity of the model. It is responsible for saving itself to storage and so on. This is accomplished by writing the published properties to the section of an ini file. Each entity is uniquely identified by an ID, and this ID is used as the name of a section in the ini file.

TModel This component represents the logic of the model. It is responsible for creating entities, and will serve as a representation of the model the IDE. In the current implementation, it has only 2 functions: instantiate a new entity, and notify components of this. In a real model component, it would also provide lists of known entities, property lists of entities etc.

The `TEntity` class is responsible for loading itself from storage, and saving itself to storage. This is accomplished in the following two methods:

```
procedure TEntity.LoadFromStorage(AID: String);

Var
  L : TStringList;
  PS : TPropsStorage;

begin
  FID:=ID;
```

```
  L:=TStringList.Create;
  Try
    GetPropertyList(L);
    PS:=CreatePropStorage;
    Try
      FIni:=TInifile.Create(ClassName+'.dat');
      Try
        PS.LoadProperties(L);
        FIni.UpdateFile;
      Finally
        FreeAndNil(FIni);
      end;
    Finally
      PS.Free;
    end;
  Finally
    L.Free;
  end;
end;
```

This method is quite simple: The ID from which to load itself is stored, and then a list of published property names is constructed via `GetPropertyList`. The `CreatePropStorage` function creates a `TPropsStorage` instance: this class is defined in the standard rttiutils unit. It can save and load properties from arbitrary storage locations, in this case the .ini file (appropriate callbacks for this are set in the `CreatePropStorage` function, they are not discussed here). In the above code, the `LoadProperties` call will accomplish this.

Saving all properties is done with `SaveToStorage`, which is quite similar:

```
procedure TEntity.SaveToStorage;

Var
  L : TStringList;
  PS : TPropsStorage;

begin
  FID:=ID;
  L:=TStringList.Create;
  Try
    GetPropertyList(L);
    PS:=CreatePropStorage;
    Try
      FIni:=TInifile.Create(ClassName+'.dat');
      Try
        PS.StoreProperties(L);
        FIni.UpdateFile;
        FModified:=False;
      Finally
        FreeAndNil(FIni);
      end;
    Finally
      PS.Free;
    end;
  Finally
    L.Free;
```

```
  end;
end;
```

Note the line which sets the `FModified` flag to `False`: it is the responsability of descendent classes to set this flas to `True` when a published property changes. Normally, a toolchain which generates code for the model, automatically takes care of this.

When a descendent of TEntity is created, used, and then freed, it should save itself if it was modified. This is done in the destructor:

```
Destructor TEntity.Destroy;
begin
  if Modified then
    SaveToStorage;
  inherited Destroy;
end;
```

The main methods of the `TModel` class are used to create instances of entities:

```
function TModel.CreateNewInstance(EntityName: String): TEntity;
begin
  Result:=TPerson.Create(Self);
  Result.FID:='Person'+IntToStr(GetNewID('TPerson'));
  NotifyAgents(Result);
end;
```

This dummy implementation of `CreateNewInstance` creates always an instance of `TPerson`, discussed later. In reality it would look up the real class from the `EntityName` property, and create an instance of that class. It then assigns a unique ID for the instance. In the above case, a dummy algorithm will assign IDs as 'Person1', 'Person2' etc, depending on the number of instances stored in the ini file. In a real model, some real autonumbering scheme would be used, like the value of an autoincremental field of some database.

To instantiate an existing entity, the `CreateInstanceFromID` method is used, which is quite similar:

```
function TModel.CreateInstanceFromID(EntityName, ID: String):
TEntity;
begin
  Result:=TPerson.Create(Self);
  Result.LoadFromStorage(ID);
  NotifyAgents(Result);
end;
```

The `NotifyAgents` method will be discussed later. It serves to notify any visual components that there is a new instance, which can be used. In a real entity factory, the code would look first if there is not already an instance of this class with the requested ID, and return that instance.

For simplicity, an extremely simple example class will be used. Starting from the `TEntity` class, a class `TPerson` was generated, which has some published properties:

```
TPerson = Class(TMDAClass)
Published
  Property FirstName : String;
  Property LastName : String;
```

```
  Property DateOfBirth : TDateTime;
end;
```

Many more properties can be added. The major point is that the relevant properties of the model are published.

The use of the TPerson class is quite simple:

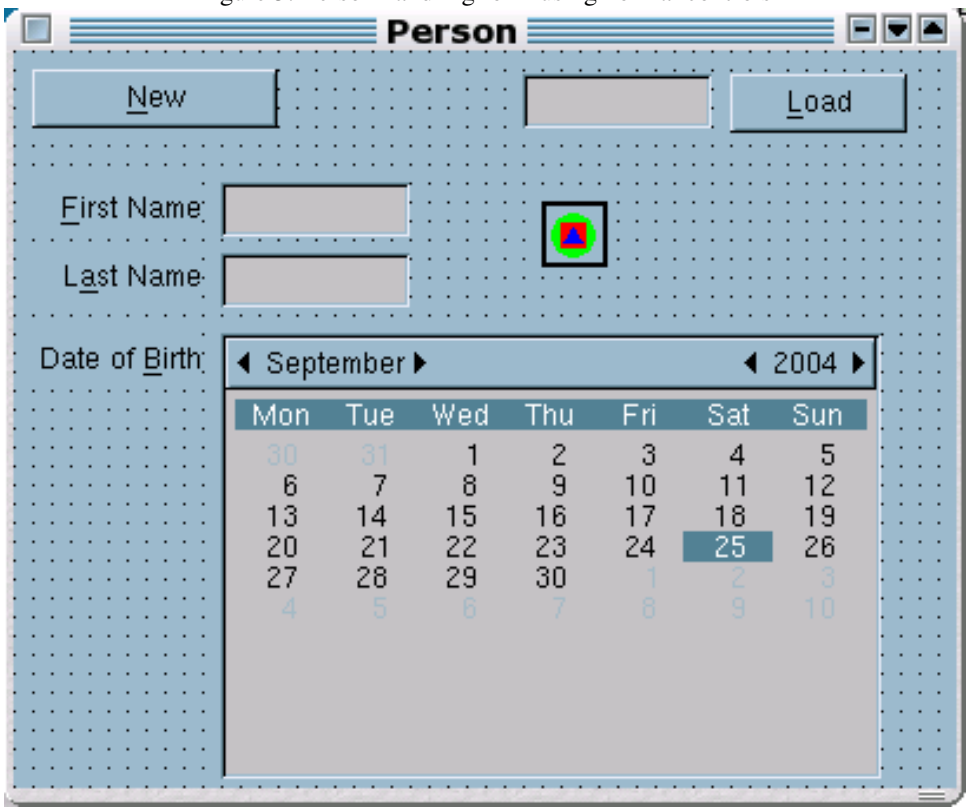
```
program testm;
{$mode objfpc}
{$h+}

uses classes,sysutils,model;

Var
  F : TModel;
  P : TPerson;

begin
  F:=TModel.Create(Nil);
  Try
    P:=F.CreateNewInstance('TPerson') as TPerson;
    Try
      Writeln('New ID :',P.ID);
      P.FirstName:='Michael';
      P.LastName:='Van Canneyt';
      P.BirthDay:=EncodeDate(1970,7,7);
    Finally
      FreeAndNil(P);
    end;
    P:=F.CreateInstanceFromID('TPerson','Person1') as TPerson;
    Try
      Writeln('First name : ',P.FirstName);
      Writeln('Last name  : ',P.LastName);
      Writeln('Birthday   : ',DateToStr(P.BirthDay));
    Finally
      FreeAndNil(P);
    end;
  Finally
    F.Free;
  end;
end.
```

The first part of this code will create a `TPerson` instance: It calls the entity factory, and gets a new instance of TPerson. It fills the needed properties, and then destroys the instance when it is no longer needed. When the instance is destroyed, any changes to it are automatically saved.

The second part of this code loads an existing TPerson from the list.

For easier manipulation of the classes in the Lazarus IDE, a lazmda package was created, which places a descendent of `TModel` called `TModel` on the component palette (on the 'MDA' tab). In the below examples, this component will be used.

Figure 5: Person handling form using normal controls



9 MDA Controls

Given now this `TPerson` class , an application must be programmed to display and manipulate 'Persons', and all other information. This is where the RTTI components come into play.

To demonstrate how the RTTI controls are used, first a program will be created which allows to load and edit instances of `TPerson`. The program is shown in figure 5 on page 16. The colorful icon is the `TModel` instance.

The 'New' and 'Load' buttons are used to create a new instance of 'TPerson', or load an existing one, based on its ID. The code to create a new instance is as follows:

```
If Assigned(FPerson) then
  begin
  SavePerson;
  FreeAndNil(FPerson);
  end;
FPerson:=FModel.CreateNewInstance('TPerson') as TPerson;
ShowPerson;
```

The `FPerson` variable contains the instance of `TPerson` that is currentlybeing edited. It is freed (thereby saving any modifications that wer made to it. And a new instance is requested from the `FModel`, which is an instance of `TModel`. After that the `ShowPerson` method is called, which will show the properties of `FPerson` in the various controls in the form:

```
procedure TMainForm.ShowPerson;

begin
  If Assigned(FPerson) then
    begin
    EFirstName.Text:=FPerson.FirstName;
    ELastName.Text:=FPerson.LastName;
    CBirthDay.DateTime:=FPerson.BirthDay;
    end
  else
    begin
    EFirstName.Text:='';
    ELastName.Text:='';
    CBirthDay.DateTime:=date;
    end;
end;
```

Before destroying the current instance of `TPerson`, all modifications made in the various controls of the form must be saved. This is done in the saveperson method:

```
procedure TMainForm.SavePerson;

begin
  If Assigned(FPerson) then
    begin
    FPerson.FirstName:=EFirstName.Text;
    FPerson.LastName:=ELastName.Text;
    FPerson.BirthDay:=CBirthDay.DateTime;
    end;
end;
```

Likewise, when an existing person needs to be edited, the var 'load' button can be used. The edit control 'EID' contains the ID of the person that should be instantiated, so the `OnClick` event of the 'Load' button looks as follows:

```
procedure TMainForm.BLoadClick(Sender: TObject);
begin
  If Assigned(FPerson) then
    begin
    SavePerson;
    FreeAndNil(FPerson);
    end;
  FPerson:=FModel.CreateInstanceFromID('TPerson',EID.Text) as TPerson;
  ShowPerson;
end;
```

As can be seen, the saving and loading of all the properties of a person instance needs to be done each time, in each form of an application.

Since all relevant properties are published, this can be done using the RTTI controls. The `TModel` control has an idea of 'Agents': components which want to be notified when a `TEntity` class is created. When it creates a new instance, then it notifies all 'Agents', and passes the new instance. Since all relevant properties of the `TEntity` are published, it makes sense to use RTTI controls: All they need is the name of the property they need

to display. The model will notify the RTTI controls when a new TEntity is created, and the RTTI controls will do the rest.

To demonstrate this, a RTTI Control descendent will be made which is 'model aware', that is, they can be used with a `TModel`.

The descendent will be a simple Edit control. It is defined as follows:

```
TMDAEdit = class(TTICustomEdit)
published
  Property Model : TModel Read GetModel Write SetModel;
  Property PropertyName : String Read GetPropertyName
                                  Write SetPropertyName;
end;
```

In this declaration, `TTICustomEdit` the same as `TTIEdit`, but without published properties (notably, the 'Link' property). Note the 2 properties `Model` and `PropertyName`. They can be set in the Lazarus IDE. The value of `PropertyName` is written to the `Link.TIPropertyName` of the parent class. It is used to decide which property of a `TEntity` must be displayed.

The Model property is set through the `FModelLink`, which is a `TDataAgent` class: this class is introduced by `TModel`, to communicate with 'agents' that need to be notified of creation of entities, such as the `TMDAEdit` control above. It is similar in function to the `TDatalink` class used by `TDataset`. It is declared as follows:

```
TDataAgent = Class(TObject)
  Constructor Create(AOwner : TComponent);
  Destructor Destroy; override;
  Procedure SetEntity(AEntity : TEntity); virtual;Abstract;
  Property Model : TModel Read FModel Write SetModel;
  Property Owner : TComponent Read FOwner;
end;
```

The `Model` refers to a `TModel` instance. The `Owner` is the component that needs to be notified when an entity is created or destroyed, in this case the `TMDAEdit` control. The `SetEntity` is an abstract method: it must be implemented by descendent classes. `TModel` keeps a list of registered agents, and notifies all agents when an entity is created, by means of the `NotifyAgents` call which simply calls `SetEntity` for all `TDataAgents` in it's list of agents.

The `TMDAEdit` class creates a `TDataAgent` descendent called `TRTTIAgent` to link to the TModel:

```
constructor TMDAEdit.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FModelLink:=TRTTIAgent.Create(Self,Self.Link);
end;
```

Note that it passes its `Link` property to the `TRRTIAgent` class, which is a descendent of `TDataAgent` that knows how to set the `Link` property of the RTTI control:

```
TRTTIAgent = Class(TDataAgent)
  Constructor Create(AOwner : TComponent; Link : TCustomPropertyLink);
  Procedure SetEntity(AEntity : TEntity); override;
end;
```

Figure 6: Model driven form

The `Link` argument of the creator is stored in the `TRTTIAgent` class, and is used when setting the entity:

```
procedure TRTTIAgent.SetEntity(AEntity: TEntity);

begin
  If Assigned(Flink) then
    FLink.TIObject:=AEntity;
end;
```

Apart from some housekeeping code, needed to notify when a model or an agent disappears, the component is now ready to be used: The 'lazmda' package can be compiled and installed, and the 'MDA' tab of the component palette will show a TMDAEdit.

A new form is created similar to the previous one. Instead of simple edits, now TMDAEdit is dropped on the form. It's 'Model' property is set to the `FModel` on the form, and the 'PropertyName' is set to 'FirstName', 'LastName' or 'BirthDay'.

The 'New' and 'Load' buttons now have very simple 'OnClick' events:

```
procedure TMainForm.BNewClick(Sender: TObject);
begin
  If Assigned(FPerson) then
    FreeAndNil(FPerson);
  FPerson:=FModel.CreateNewInstance('TPerson') as TPerson;
end;

procedure TMainForm.BLoadClick(Sender: TObject);
begin
  If Assigned(FPerson) then
    FreeAndNil(FPerson);
  FPerson:=FModel.CreateInstanceFromID('TPerson',EID.Text) as TPerson;
end;
```

There are no other methods in the form. Everything is now handled by the `TModel` and `TMDAEdit` controls. The form can be seen in action in figure 6 on page 19. The form could be made even simpler by creating some buttons which are model aware, similar to the navigator buttons for a TDataset: the 'New' button could have 2 properties 'EntityName' and 'Model' and, when pressed, would tell the model to create a new instance of the asked Identity type.

10 Conclusion

RTTI controls are useful in their own right: They can be used for component editors without any code, they can be used in simple applications to reduce code. But the most important feature may well be that the RTTI controls in Lazarus may be the start of a powerful MDA toolchain for Lazarus. The simple (toy) toolchain presented here is fully functional, and was created using very little code. It showed that the RTTI controls have the potential to be used in a real MDA toolchain, such as Bold for Delphi.

Obviously, a lot is still missing in Lazarus, both simple and difficult:

- Other controls.
- Property editors for the 'PropertyName' property.
- Real storage, using a database, instead of ini files.
- Lists of entities must be presentable in listboxes and grids.
- Erasing of entities must be handled.
- A UML model editor which creates all classes (such as ModelMaker) must be created.

But creating all this was not the purpose of this article: The purpose was to show - in a convincing way - that using simple techniques, the RTTI controls can be turned into powerful MDA tools. Maybe the further creation of a MDA toolchain can be the subject of another article.