Designing non-visual objects

Michaël Van Canneyt

November 21, 2009

Abstract

The lazarus IDE is made to design forms: it manipulates in the first place GUI objects: descendents from TControl, which are painted as they look in reality. Non visual controls are represented with an icon. Recently, the IDE has been enhanced, so the IDE can be used to design - visually - components which are not descendents from TControl.

1 Introduction

The Lazarus IDE is a RAD tool: created to design GUI applications. It can handle non-visual components, but these are represented merely by an icon. It can also handle non-visual containers like TDatamodule, which is useful to group for instance database-related components, and thus separate database logic from GUI logic. Lazarus had already taken this one step further by allowing to design non-TDatamodule components as if it was a datamodule: the daemon mapper component from the daemon support is an example of this (described in Toolbox 3/2007): Basically any object that can be streamed (i.e. a .lfm file can be created for it) can be designed in the IDE. However, it will always be represented by a kind of TDatamodule-like window, and no visual components can be dropped on this, only non-visual components can be dropped on such an object, and they are represented by an icon.

A recent development (The `TDesignerMediator` interface of the IDE) has taken this one step further: it is now possible to design and draw a non-visual component as if it was a visual control: The component will no longer be shown as an icon, but it can actually be drawn and manipulated on the designer window.

Examples where this can be useful can be the following:

1. Report components: The report designers are currently totally separate designers, which mimic the Lazarus IDE designer. This was necessary, because the report's bands and printable objects are not actual TControl descendents. using this new development, reports can be designed totally in the IDE itself, using the IDE's form designer.
2. UML diagrams: Imagine a set of TComponents for creating and describing UML diagrams: objects to describe objects, tables, use-cases. If they were to be dropped on a form, they would be represented by an icon. With the new technique, the components could be drawn on the form so that they give a visual representation of what they describe.
3. Database designers. The Database desktop can be used to design a database, but this is not done visually, but rather in tree and grid-like structures. Tools like IBExpert or IBAdmin draw actual database diagrams. The new technique would enable to design these diagrams in the IDE.

4. A widget set that does not use the LCL (for example fpGUI, see the article in toolbox 3, 2008) could be designed in the Lazarus IDE itself.
5. Web-pages. There are some enhancements to Lazarus and Delphi that draw web-pages in the designer. These use TForm and regular controls as the starting point. Instead, the new technique allows to create a webpage in the designer, without any TForm or controls: instead of TForm, a TWebPage component could be used, and instead of TControls, controls that represent various HTML or Javascript elements can be dropped and designed visually.

In this article, the `TDesignerMediator` interface of the IDE will be presented. a rudimentary database designer interface will be developed as a show-case.

2 Installation

Since this is a recent development, a development snapshot of the Lazarus IDE is needed, it has not yet been included in an official release. A development snapshot can be downloaded from the Lazarus website:

```
http://www.lazarus.freepascal.org/
```

Or the source code can be grabbed from subversion:

```
http://svn.freepascal.org/svn/lazarus/trunk
```

Once the IDE was installed or recompiled from sources, the TDesignerMediator support will be present, and an example can be viewed in the examples/designnonlcl directory: there is a package that must be installed in the IDE (notlcldesigner.lpk), and then the sample project in example/designnonlcl/project/nonlcl1.lpr can be opened and manipulated. It shows how to use the new mediator to design and use an imaginary widget set. This is shown in figure 1 on page 3.

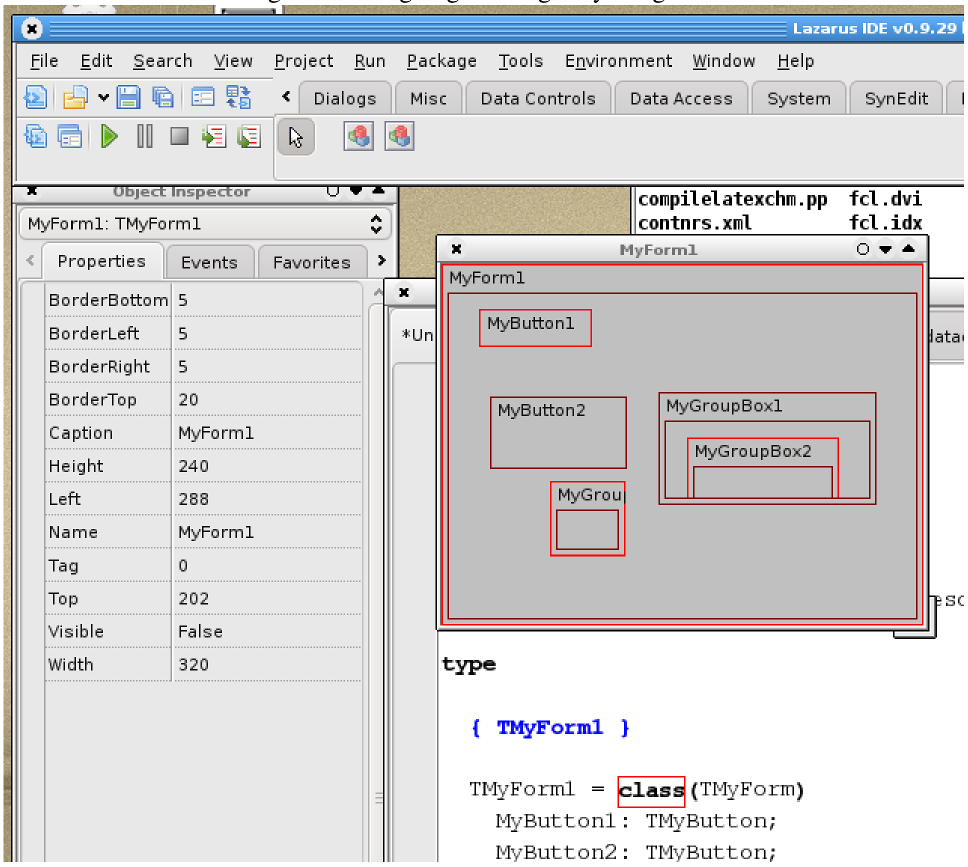
3 TDesignerMediator

The implementation of the TDesignerMediator class stems from a simple idea: if a non-visual component can be represented by an icon, it should also be possible to draw something else instead of an icon. If this icon can be moved about on the page, then the same should hold true for whatever is drawn on the page. A component has no idea that it is being represented by an icon. The IDE implements some default behaviour for non-TControl components: namely, drawing an icon.

The TDesignerMediator class simply makes this default behaviour configurable. It provides a way for a component developer to instruct the IDE not to draw an icon for some component, but to let the IDE call a special drawing routine whenever it must represent the component on a form: the TDesignerMediator contains a 'Paint' call which draws the component on a canvas, provided by the IDE.

Designing visually not only consists of drawing a component. The component can be moved about etc. The `TDesignerMediator` class has some additional methods that allow the component developer to react on clicks, keystrokes and resizing and moving of the component. It should also notify the designer when it needs to repaint the component, for instance when the properties have changed: TControl descendents automatically repaint themselves whenever something is changed, but non-visual components will need to notify

Figure 1: Designing an imaginary widget set


Lazarus IDE v0.9.29
File Edit Search View Project Run Package Tools Environment Window Help
Dialogs Misc Data Controls Data Access System SynEdit
Object Inspector
MyForm1: TMyForm1
Properties Events Favorites
MyForm1
MyButton1
MyButton2
MyGroupBox1
MyGroupBox2
MyGrou

```
compilelatexchm.pp   fcl.dvi
contnrs.xml          fcl.idx
```

BorderBottom	5
BorderLeft	5
BorderRight	5
BorderTop	20
Caption	MyForm1
Height	240
Left	288
Name	MyForm1
Tag	0
Top	202
Visible	False
Width	320

```
type

  { TMyForm1 }

  TMyForm1 = class(TMyForm)
    MyButton1: TMyButton;
    MyButton2: TMyButton;
```

the IDE when a repaint is needed, so that the IDE can instruct the mediator to repaint the control.

All this means that there is a strict separation of GUI representation and the component itself: The component itself does not need to know anything about TCanvas or and can hence be totally ignorant from the LCL. The Mediator component contains all knowledge about the component, and uses this knowledge to draw a visual representation on the LCL canvas. The component does need to be aware that it can be designed visually in an IDE: it must know how to notify the designer of changes. (Just as the concept of observers/observed in the MGM Pattern, described in Toolbox X/Y by Graeme Geldenhuys).

Currently, all this does not yet work on an actual TForm descendent. There must be a container class (similar to TDatamodule or so) which represents the form, which will be streamed to the .lfm file. The mediator will represent this container in the IDE using a designer window as it would represent a TForm or TDatamodule, and further handles all drawing on this container.

4 Creating a database designer: the objects

In order to demonstrate the use of the TDesignerMediator class, a small case study will be made: a set of components representing a database diagram. A database diagram should contain visual representations of tables, stored procedures and all other possible database objects. Additionaly it can contain some objects to hold comments and explanations. For simplicity's sake, the model will be limited to 2 components: a `TModelTable` and `TModelText` component, representing a table and an explanatory text. They are declared in a unit mdldatadict as follows:

```
TModelText = Class(TModelComponent)
Published
  Property Caption : String;
  Property Text : TStrings;
end;

TModelTable = Class(TModelComponent)
Published
  Property TableName : String;
  Property Fields : TModelFieldDefs;
end;
```

Both classes are descendents of TModelComponent, which will be explained below. The TModelText class has 2 properties: a caption and a text, which can be used to place a small explanatory panel with caption and some text in the model.

The `TModelTable` has 2 properties: `TableName` (obviously) and `Fields`. The `Fields` property is a collection of `TModelFieldDef` items: one item per field, describing each field in the table. When the table is selected, the collection editor of lazarus can be used to manipulate the `Fields` property. As items are added to the collection, they will be drawn in the visual representation of the table. The details of `TModelFieldDef` are not really important to the discussion. Suffice it to say that it makes use of the data dictionary support implemented in the FCL, and discussed in the article on the database desktop. For the discussion here, it is sufficient to know that TModelFieldDef has 2 properties: 'FieldName' and 'FieldType', with obvious meanings.

The container for the tables and other database objects is called TModelPage: This represents one page in the database diagram. It plays the role of the form for GUI windows, and is used to place all the database model components on:

```
TDDModelPage = Class(TModelComponent)
Protected
  procedure GetChildren(Proc: TGetChildProc; Root: TComponent); override;
  Procedure DoInvalidateRect(Const ARect : TRect; Erase: boolean); virtual;
Public
  Property OnInvalidateRect : TOnInvalidateRect;
Published
  Property PageName : String;
end;
```

The 'PageName' property is used as a caption. The GetChildren call must be overridden, and should return all components on the page: when the page is streamed to the .lfm file, this call is used to determine which components are streamed to the file.

The `DoInValidateRect` call serves only as a hook to the TDesignerMediator, to notify it when it must repaint the page: the TDesignerMediator class will set the `OnValidateRect` event, and `DoInvalidateRect` will call the event handler if it is set.

All three components are descendent from `TModelComponent`. This is done deliberately: `TModelComponent` contains all methods needed to be able to work with the model using the IDE's designer.

```
TModelComponent = Class(TComponent)
Protected
  Procedure InvalidateRect(Erase : boolean);
Public
  function HasParent: Boolean; override;
  function GetParentComponent: TComponent; override;
  Procedure SetBounds(Const ALeft,ATop,AWidth,AHeight : Integer);
Published
  Property Top : integer;
  Property Left : integer;
  Property Width : integer;
  Property Height : integer;
end;
```

The presence of the Top/Left/Width/Height properties should not come as a surprise: they represent the location and size of the component in the model page. The `HasParent` and `GetParentComponent` methods of TComponent must be overridden, and are implemented as follows:

```
function TModelComponent.HasParent: Boolean;
begin
  Result:=Owner is TModelComponent;
end;

function TModelComponent.GetParentComponent: TComponent;
begin
  if Owner is TModelComponent then
    Result:=Owner
  else
    Result:=nil;
end;
```

They do nothing spectacular, but are required for the working of the IDE: the IDE needs to know where it has to place the components on the form, and which component is the visual 'parent' of each control.

The `InvalidateRect` call can be used to notify the designer when the component has changed, and must be drawn again:

```
procedure TModelComponent.InvalidateRect(Erase : Boolean);

Var
  P : TDDModelPage;
  R : TRect;

begin
  If Self is TDDModelPage then
    P:=TDDModelPage(Self)
  else if Owner is TDDModelPage then
    P:=TDDModelPage(Owner)
  else
    P:=Nil;
  If Assigned(P) then
    begin
    R:=Rect(Left,Top,Left+Width,Top+Height);
    P.DoInvalidateRect(R,Erase);
    end;
end;
```

It tries to determine the `TDDModelPage` instance on which it was dropped, and will then call `DoInvalidateRect` to notify the designer that a redraw is in order, passing it's own bounding rectangle as the rectangle to redraw.

Note that nowhere, a reference is made to to canvases or the LCL: the mdldatadict relies only on the Classes unit, and uses the fpDatadict unit for the TDDFieldDef class. TRect is defined in the Classes unit as well.

5 Creating a database designer: the mediator

With all the business objects in place (model page, table and text objects), it is time to turn to the mediator. For each kind of container (in our case, `TDDModelPage`), a descendent of `TDesignerMediator` must be registered. The mediator for the database diagram will be implemented in a unit `meddatadict`, and is called `TModelMediator`. Here is the declaration of the mediator, leaving away the private methods:

```
TDDModelMediator = class(TDesignerMediator)
protected
  procedure InvalidateRect(Sender: TObject; Const ARect: TRect;
                           Erase: boolean);virtual;
public
 class function FormClass: TComponentClass; override;
 class function CreateMediator(TheOwner,
    aForm: TComponent): TDesignerMediator; override;
 procedure GetBounds(AComponent: TComponent;
                     out CurBounds: TRect); override;
 procedure SetBounds(AComponent: TComponent;
                    NewBounds: TRect); override;
 procedure GetClientArea(AComponent: TComponent;
                        out CurClientArea: TRect;
                        out ScrollOffset: TPoint); override;
 procedure Paint; override;
 function ComponentIsIcon(
             AComponent: TComponent): boolean; override;
 function ParentAcceptsChild(Parent: TComponent;
             Child: TComponentClass): boolean; override;
end;
```

All the methods shown here are overrides of abstract methods in the `TDesignerMediator` class. They must be implemented by descendent classes. The purpose and implementation of each of the methods will be discussed below.

The `FormClass` method should return a class pointer to the container class. In the case of the database diagram, this is `TDDModelPage`:

```
class function TDDModelMediator.FormClass: TComponentClass;
begin
  Result:=TDDModelPage;
end;
```

The IDE uses this to determine which container class the mediator can handle.

The `CreateMediator` call is used to actually create an instance of the mediator, and should be overridden to perform additional any initialization that is required. In the case of the database diagram, this means setting the `OnInvalidateRect` event handler, so the mediator is notified when the page needs to be redrawn. A reference to the model page is also saved, for easy access:

```
class function TDDModelMediator.CreateMediator(TheOwner,
                  aForm: TComponent): TDesignerMediator;
var
  Mediator: TDDModelMediator;
  P : TDDModelPage;
begin
  Result:=inherited CreateMediator(TheOwner, aForm);
  Mediator:=TDDModelMediator(Result);
  P:=aForm as TDDModelPage;
  Mediator.FDDPage:=P;
  P.OnInvalidateRect:=@InvalidateRect;
end;
```

When the `InvalidateRect` event handler is called, it simply transfers the request to the form designer window. This window is available through the `LCLForm` property of `TDesignerMediator`:

```
procedure TDDModelMediator.InvalidateRect(Sender: TObject;
                                       Const ARect: TRect;
                                          Erase: boolean);
begin
  if (LCLForm=nil) or (not LCLForm.HandleAllocated) then
     exit;
  LCLIntf.InvalidateRect(LCLForm.Handle,@ARect,Erase);
end;
```

As a result, the designer window will then call the `Paint` method of the mediator, which should then redraw the container's surface.

6 position and size handling

The following three methods: `GetBounds`,`SetBounds` and `GetClientArea` are needed to tell the designer window how big the components are: The designer does not assume the presence of `top`,`left`, `width` and `height` properties, but leaves it up to the mediator to get/set the bounds of a component. The `GetBounds` method is called whenever the designer needs to know the location and size of a component: on return, the `CurBounds` parameter must be filled with the bounding rectangle of `AComponent`.

In the case of the database diagram components, this is just the rectangle defined by the `top`,`left`, `width` and `height` properties.

```
procedure TDDModelMediator.GetBounds(AComponent: TComponent;
                                     out CurBounds: TRect);

begin
  if Not (AComponent is TModelComponent) then
    inherited GetBounds(AComponent,CurBounds)
  else
    With TModelComponent(AComponent) do
      begin
      CurBounds:=Bounds(Left,Top,Width,Height);
      end;
end;
```

Note that the location is always relative to the parent component, not necessarily relative to the container. Reversely, the `SetBounds` call must set the new bounds of `AComponent` as passed in `NewBounds`:

```
procedure TDDModelMediator.SetBounds(AComponent: TComponent;
                                     NewBounds: TRect);

Var
  MC : TModelComponent;

begin
  if Not (AComponent is TModelComponent) then
    inherited SetBounds(AComponent,NewBounds)
  else
    begin
    MC:=TModelComponent(AComponent);
    With NewBounds do
      MC.SetBounds(Left,Top,Right-Left,Bottom-Top);
    end
end;
```

By default, the `GetBounds` and `SetBounds` as implemented in TDesignerMediator can determine the `Left`,`Top` properties from the `TComponent` properties, but the width and height must always be specified.

Last but not least, the `GetClientArea` is needed to tell the designer what the client area is of a component that acts as a parent to other components: this is the area of the component, available to child components. It is also used when determining what component lies at a certain position on the container:

```
procedure TDDModelMediator.GetClientArea(
   AComponent: TComponent;
   out CurClientArea: TRect;
   out ScrollOffset: TPoint);
Var
  MC : TModelComponent;
begin
  if Not (AComponent is TModelComponent) then
    inherited GetClientArea(AComponent,
                            CurClientArea,ScrollOffset)
  else
    begin
    MC:=TModelComponent(AComponent);
    With MC do
      CurClientArea:=Rect(0,0,Width,Height);
    ScrollOffset:=Point(0,0);
    end;
end;
```

The `ScrollOffset` is an offset that should be used to offset positions inside the client area if the client area has been scrolled somehow - this is needed to support components that mimic a scrollbox-like.

Obviously, it should still be possible to drop normal non-visual components on a TDDModelPage, which should be represented by an icon, as on regular forms. The `ComponentIsIcon` call can be used for this. In the case of the database diagram, only descendents of `TModelComponent` can be drawn, and any other component must be represented by an icon:

```
function TDDModelMediator.ComponentIsIcon
     (AComponent: TComponent): boolean;
begin
  Result:=not (AComponent is TModelComponent);
end;
```

Lastly, the designer needs to know where a component can be dropped. For this, the `ParentAcceptsChild` call must be implemented; it should return `True` if a component of class `Child` can be dropped on component `Parent`. In the case of the database diagram components, all components can only be dropped on the model page itself:

```
function TDDModelMediator.ParentAcceptsChild(Parent: TComponent;
  Child: TComponentClass): boolean;
begin
  Result:=(Parent is TDDModelPage);
end;
```

7 The real work: painting the components

The methods till now served to give the IDE's form designer all information it needs to place components on the container, to move these components around, and to resize them.

The form designer can also be told when it should redraw a portion of the form. This leaves only one task to be completed: the actual drawing of the components. This must be handled in the `Paint` method. The `Paint` method should draw all components on the canvas of the design form: the design form is available through the `LCLForm` property. Prior to calling the `Paint` method, the LCL will have set a Clipping rectangle on the canvas of the LCLForm, so only components that fall within the clipping rectangle must be redrawn.

For the database diagram, the Paint method looks like this:

```
procedure TDDModelMediator.Paint;

Var
   DC : HDC;
   I : Integer;
   Comp : TModelComponent;

begin
  PaintBackground;
  DC:=LCLForm.Canvas.Handle;
  For I:=0 to FDDPage.ComponentCount-1 do
    if FDDPage.Components[i] is TModelComponent then
      begin
      Comp:=TModelComponent(FDDPage.Components[i]);
      LCLForm.Canvas.SaveHandleState;
      try
        MoveWindowOrgEx(DC,Comp.Left,Comp.Top);
        if IntersectClipRect(DC,0,0,Comp.Width,Comp.Height)
           <>NullRegion then
          PaintComponent(Comp);
      finally
        LCLForm.Canvas.RestoreHandleState;
      end;
      end;
end;
```

After painting the background (implemented in `PaintBrackground`, it does a simple loop. For all components dropped on the model page it does the following things:

1. It saves the canvases' state using `SaveHandleState`, a standard LCL Call.
2. It moves the origin of the canvas to the component's position.
3. If the component intersects with the clipping rectangle by means of the `IntersectClipRect` call, it paints the component.
4. It restores the canvas to it's previous state.

That's all there is to the paint method, the actual work is done in the `PaintBackground` and `PaintComponent` methods. The `PaintBackground` call is a simple call:

```
procedure TDDModelMediator.PaintBackground;

begin
  with LCLForm.Canvas do
    begin
    Brush.Style:=bsSolid;
    Brush.Color:=clLtGray;
    FillRect(0,0,FDDPage.Width,FDDPage.Height);
    end;
end;
```

As can be seen, it simply paints the background using a pre-defined color. Since a clip rectangle is in effect, only the area that needs painting will be actually painted. The `PaintComponent` method is a simple dispatching method:

```
procedure TDDModelMediator.PaintComponent(AComponent :
   TModelComponent);

begin
  if AComponent is TModelTable then
    PaintTable(TModelTable(AComponent))
  else if AComponent is TModelText then
    PaintText(TModelText(AComponent))
end;
```

The `PaintTable` and `PaintText` methods of `TDDModelMediator` will draw the actual `TModelTable` or `TModelText` component. The `PaintText` method is the simplest of these two, as it only has to draw 2 texts in a uniformly colored rectangle. It starts by setting the colors (using `SetTextColors`, not shown here) and drawing a rectangle which is the background of the text panel. After that it draws the caption:

```
procedure TDDModelMediator.PaintText(AText : TModelText);

Const
  LineHeight = 19;

Var
  R : TRect;
  S : String;
  TS : TTextStyle;

begin
  SetTextColors(LCLForm.Canvas);
  LCLForm.Canvas.FillRect(0,0,AText.Width,AText.Height);
  R.Top:=3;
  R.Left:=3;
  R.Bottom:=LineHeight;
  R.Right:=AText.Width-3;
  LCLForm.Canvas.TextRect(R,3,3,AText.Caption);
  LCLForm.Canvas.Line(0,LineHeight,AText.Width,LineHeight);
  R.Top:=LineHeight+3;
  R.Bottom:=AText.Top+AText.Height-3;
  S:=AText.Text.Text;
  If (S<>'') then
    begin
    // initialize textstyle.
    FillChar(TS,SizeOf(TS),0);
    TS.Alignment:=taLeftJustify;
    TS.Layout:=tlTop;
    TS.Wordbreak:=True;
    TS.SingleLine:=False;
    LCLForm.Canvas.TextRect(R,R.Left,R.Top,S,TS);
    end
end;
```

When the caption is drawn, a line is drawn under the caption, and then the `TextRect` call is used to draw the text of the text panel: it is by no means a difficult routine.

The `PaintTable` is quite similar to `PaintText`. It does pretty much the same: as caption text the table name is used, and below that, for each field in the table a line is painted with the name and type of the field. The interested reader can consult the sources of the mediator to see how it is done.

To get everything to work, it is still necessary to register the mediator, and to register the `TDDModelPage` class So it can be selected from the File-New menu. This is done in the `Register` procedure in the `meddatadict` unit:

```
procedure Register;
begin
  FormEditingHook.RegisterDesignerMediator(TDDModelMediator);
  RegisterComponents(SDataModeling,[TModelTable,TModelText]);
  RegisterNewItemCategory(TNewIDEItemCategory.Create(SDataModeling));
  RegisterProjectFileDescriptor(TDDModelPageDescriptor.Create,SDataModeling);
end;
```

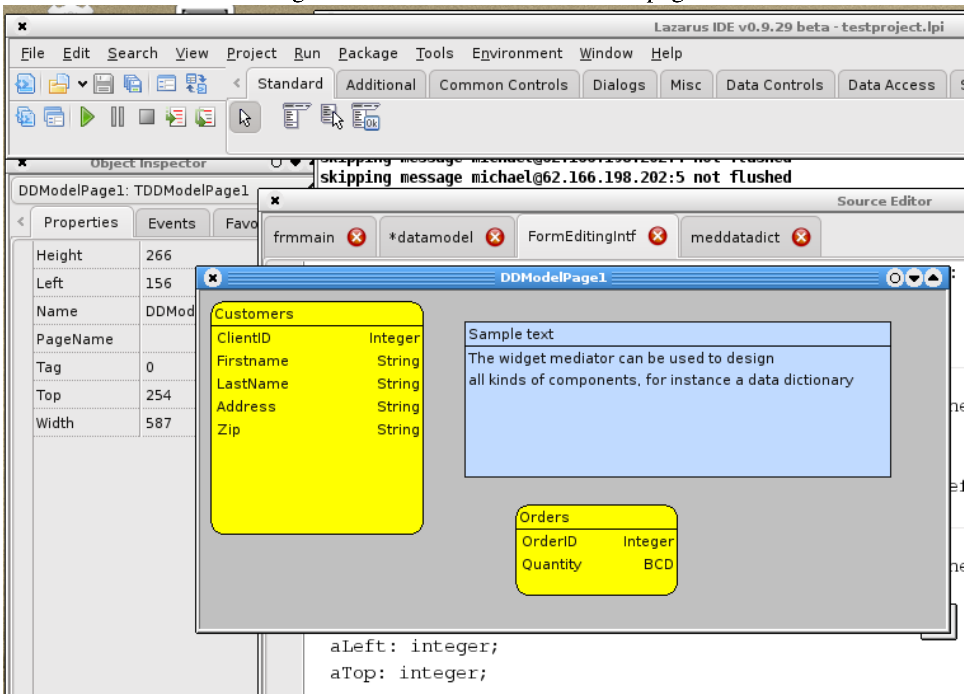
The first line registers the model mediator: it is sufficient to pass the class of the implemented mediator to the `RegisterDesignerMediator` call of the `FormEditingHook` instance (in the FormEditingIntf unit). To be able to pick some diagram components from the component palette, they must be registered: this is done in the second line. The last 2 lines introduce a new category and item in the File-New dialog: the details of the `TDDModelPageDescriptor` class are in the sources accompagnying this article. The mechanism for creating new files has been describes in detail in Toolbox 4/2005.

The result of all this can be seen in figure 2 on page 13: From the file-new menu, a new 'Data model page' was chosen, and the resulting new unit was saved as 'datamodel'. On the data model page, 2 tables and a text panel were dropped. As can be seen the tables have been drawn using a different color than the text, and have rounded corners. More effects can be added: the model page could contain some properties that determine the look of the objects dropped on it, or the look could be determined from some global IDE settings.

8 conclusion

The database diagram implemented here is of course far from finished, but it shows how to use the new mediator; A UML Diagram designing mediator could be designed using the same approach. This new technology is not just a theoretical toy, because as this article is written, an effort is underway to enhance fpWeb using the new mediator class: It should make it possible to design a webpage visually in the Lazarus IDE: various approaches for doing this are possible: as soon as it is ready for production use, it will be discussed in detail in a future contribution.

Figure 2: The finished Data Model page


Lazarus IDE v0.9.29 beta - testproject.lpi
File Edit Search View Project Run Package Tools Environment Window Help
Standard Additional Common Controls Dialogs Misc Data Controls Data Access
Object Inspector
Properties Events Favo
Source Editor
frmmain *datamodel FormEditingIntf meddatadict
DDModelPage1

```
skipping message michael@62.166.198.202:5 not flushed
```

DDModelPage1: TDDModelPage1

Height	266
Left	156
Name	DDMod
PageName	
Tag	0
Top	254
Width	587

Customers

ClientID	Integer
Firstname	String
LastName	String
Address	String
Zip	String

Sample text

The widget mediator can be used to design
all kinds of components, for instance a data dictionary

Orders

OrderID	Integer
Quantity	BCD

```
aLeft: integer;
aTop: integer;
```