**Mediator and ORM**

There are all kinds of ORM tools created for pascal to work with standard GUI-components connected with a lot of frameworks like mORMot and tiOPF. They are working only with objects. mORMot itself uses an internal SQLite database to store all kind of things. But the similar thing of these frameworks are combining non-data components to a (rest)database. No dataset is used. But the greatest problem of these frameworks is documentation. The one has less but a lot of examples and the other has more than 1000 pages in PDF. If you really want to play with these frameworks, it will take a lot of time and patient. But it works.

In my opinion the libraries are very complex, but I learned a small bit of tiOPF how to connect my standard GUI-components to an object and working with other objects to read and write data.

I want it to do it myself to see what really happens and my own framework ‘MyMeditor’ is born.

**Basics**

The best thing to save components is using objects. In this case I selected generics (Lazarus : fgl / Delphi : system.generics.collections) to accomplish it. It is a ‘static’ way to read the added object without using typecasting in the original way with T{Object}list. I also used a dictionary (Lazarus : TFPGMap / Delphi : TDictionary) to add an object with a name. It is just like TStringlist, but this components works only with generics. Whit a dictionary you can easily search with a name.

The dictionary is linked to my created class TMyCollection (see example below). TMyCollection has a property called fDisplayComponent which has TControl declared. In this case it is the closest object for all the components I want to use. For now only TEdit will be provided to the basemediator. This is the code I use for this moment:

unit clMyBaseMediator;

interface

uses classes, fgl, sysutils, controls, stdctrls;

type TMyDeclarations = (dcString, dcInteger, dcFloat, dcDate);

TMyCollection = class

fDisplayComponent : TControl;

fClassname : string;

fOldValue : string;

fReadOnly : boolean;

end;

TMyBaseMediator = class

private

fcomponentList : Tmydictionary;

function locate(const aFieldname : string; var cln : TMycollection) : boolean;

protected

procedure mdread(const aFieldname: string; const aValue: variant);

public

constructor Create;

destructor Destroy; override;

procedure AddComp(aCompname : TControl; const aFieldname: string);

end;

implementation

constructor TMyBaseMediator.Create;

begin

fcomponentList := Tmydictionary.create;

end;

destructor TMyBaseMediator.Destroy;

begin

fcomponentList.Free;

inherited;

end;

procedure TMyBaseMediator.mdread(const aFieldname: string; const aValue: variant);

var cln : TMycollection;

begin

cln := nil;

if locate(aFieldname,cln) then

begin

if cln.fDisplayComponent is Tedit then

TEdit(cln.fDisplayComponent).Text := aValue;

end;

end;

procedure TMyBaseMediator.AddComp(aCompname : TControl; const aFieldname: string);

var cln : TMyCollection;

begin

cln := TMyCollection.Create;

cln.fDisplayComponent := aCompname;

cln.fClassname := aCompName.ClassName;

fComponentlist.Add(aFieldname,cln);

end;

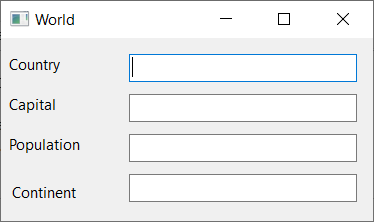
function TMyBaseMediator.locate(const aFieldname : string; var cln : TMycollection) : boolean;

begin

result := fComponentlist.TryGetData(aFieldname,cln)

end;

Later in this article I will explain the basic functions in the basemediator.

Let me connect a few TEdits to the mediator. For our example I create a demo World. I Put several TLabels (country, capital, population and continent) with the ’linked’ TEdits to fill our data. Rename the form into FrmMain. To connect the TEdits to the mediator I can use the TBaseMediator, but then I have to copy the unit several times and code will be duplicated. The best way to copy the TBasemediator is creating a new class with mainclass TBasemediator. I called this class TMainMediator (type TMainMediator = class(TBaseMediator). Now all code and definitions are inherited to use as a single class.

The main purpose is to develop an application where code is separated from the GUI. The form should not know what the database is doing with data. Advantage by separating is the form can easily modified / replaced and still working with data. That is why I create a separate unit with a class to communicate with the form. Create a new unit with name clMain and a class named TMain. This class is declared as an object with the variable name ‘flink’.

clmain:

{$mode objfpc}{$H+}

interface

uses

Classes, SysUtils;

implementation

type TMain = class

end;

end.

form:

uses ……, clmain;

type

TFrmMain = class(TForm)

………

procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);

procedure FormCreate(Sender: TObject);

public

flink : TMain;

end;

implementation

{$R \*.lfm}

procedure TFrmMain.FormCreate(Sender: TObject);

begin

flink := TMain.create

end;

procedure TFrmMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);

begin

flink.free

end;

The ‘gateway’ is created. Now TMainMediator can be declared as an object to variable fMediator (fMediator = TMainMediator) and with the variable flink in my form I can connect al my TEdits to the mediator. This is done in the procedure FormCreate of the form:

procedure TFrmMain.FormCreate(Sender: TObject);

begin

flink := TMain.create;

fLink.Mediator.AddComp(ECountry ,'countryname');

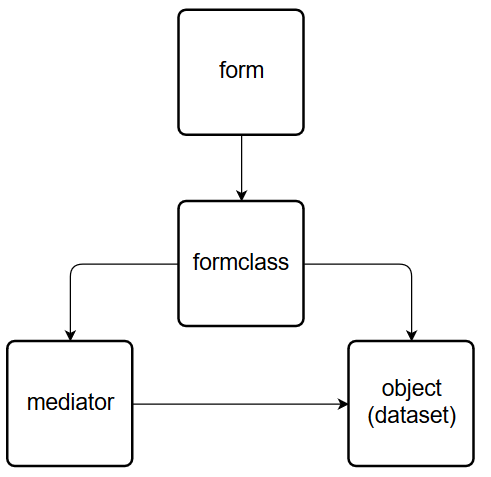
fLink.Mediator.AddComp(ECapital,'capital');

fLink.Mediator.AddComp(EPopulation,'population');

fLink.Mediator.AddComp(EContinent,'continent');

end;

Let me explain. The procedure AddComp is called from the TBaseMediator and has two parameters: TEdit component and a name variable. Because TEdit derives from TControl, TEdit can saved as a pointer value. With the second variable the component can be found with a value. That is why TEdit can get his value displayed. But how does my application know which value it is. By making a new class with the variables and objectlist that holds a list of values like TDataset. I save the unit with the name objWorld. Let me show with a flowchart how the objects are working together:



Let me place some data into the TEdit with the mediator. First I fill a record to TCountrylist and call the mediator to read the data. I created a new procedure TMain.init to fill an object with the country I live:

procedure TMain.init;

var country : Tcountry;

begin

Country := TCountry.create;

Country.countryname:= 'the Netherlands';

Country.capital := 'Amsterdam';

Country.population := '14 million';

country.continent := 'Europe';

fWorld.Countrylist.Add(Country);

fMediator.ReadToComp(fWorld.fcountrylist[0]);

end;

This procedure is called in de TFrmMain.FormCreate of the form. After filling data, I ask to the Mediator to show my data into the requested components with procedure ReadToComp with the parameter where the data holds:

procedure TMainMediator.ReadToComp(aCountry : TCountry);

begin

mdread('countryname',aCountry.name);

mdread('capital',aCountry.capital);

mdread('population',aCountry.population);

mdread('continent',aCountry.continent);

end;

Procedure ReadToComp reads procedure mdread with three parameters: the name of the object stored in dictionary a value (declared as an variant type) and an optional value to tell in which declaration type the value is stored. With no third parameter, the values is always a string type.

procedure TMyBaseMediator.mdread(const aFieldname: string; const aValue: variant; aDec : TMyDeclarations);

var cln : TMycollection;

begin

if locate(aFieldname,cln) then

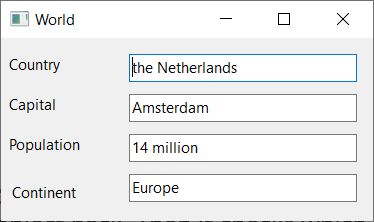
begin

if cln.fDisplayComponent is Tedit then

TEdit(cln.fDisplayComponent).Text := aValue;

end;

end;

First function mdread locates the right component on name in the dictionary. The function locate gives the founded object back. Then it checks which component belong to the object and gives the value to display. In the beginning I told the mediator to connect the TEdit from the form. It is actually a pointer to an address of TEdit and therefore a values can be displayed. And still the form doesn’t know what is really going on.

**More data**

I take a step further to add, save and scroll data with the mediator. The form will be extended with a toolbar and 4 buttons within. I connect the toolbarbuttons with a TActionlist.

Please read <https://wiki.freepascal.org/TActionList> or <http://etutorials.org/Programming/>

mastering+delphi+7/Part+I+Foundations/Chapter+6+Building+the+User+Interface/The+ActionList+Component/ if you don’t know how to work with actions. I remove the lines in procedure TMain.init to have a clear countrylist on startup. To know the record is new, I create a new property Newrecord : boolean into class Tmain. Clicking on the event onClick of button btnAdd gives a new entry in the code editor:

//fmain.pas

procedure TFrmMain.BtnAddClick(Sender: TObject);

begin

flink.AddRecord;

end;

//clmain.pas

procedure TMain.addrecord;

begin

fMediator.Clear;

fNewRecord := true;

fWorld.IncreaseID;

end;

In the basemeditor all connected components are cleared to input:

procedure TMyBaseMediator.Clear;

var cln : TMycollection;

index : integer;

begin

for index := 0 to fcomponentList.Count - 1 do

begin

cln := fComponentlist.Data[index];

TEdit(cln.fDisplayComponent).Text := '';

end;

end;

The property NewRecord is set to true to tell the procedure save te create a new object to save into the list.

Clicking on the onClick of button btnSave gives a new entry in the code editor and put this line flink.save which is linked to a procedure save in classe TMain:

The variabele fNewRecord creates a new object of TCountry and calls procedure CmpToWrite in TMainMediator to collect all connected values. I expand the TMainmediator and TBasemediator to accomplish it:

procedure TMainMediator.CmpToWrite(aCountry : TCountry);

begin

aCountry.countryname:= mdwrite('countryname');

aCountry.capital := mdwrite('capital');

aCountry.population := mdwrite('population');

aCountry.continent := mdwrite('continent');

end;

function TMyBaseMediator.mdwrite(const aFieldname : string) :variant;

var cln : TMycollection;

begin

cln := nil;

if locate(aFieldname,cln) then

begin

if cln.fDisplayComponent is TEdit then

result := TEdit(cln.fDisplayComponent).Text;

end;

end;

After saving the variable fNewRecord is set to false. Now I can add several objects to the objectlist. But if the record already created and has only to be modified, how do you know which object must be modified. I Can search on a name, but names can be modified? The easiest way to make a record identyfied is putting an extra variable in class TCountry of objcountrylist. Lets call it id : integer. After saving a record a unique number is created. In class TWorld I put three extra properties:

property LastID : integer read fLastID write fLastID;

property ID : integer read fID write fID;

property Error : boolean read fError write fError;

Now I can locate the right object to save the modified values put in TEdit:

function TWorld.FindRow: TCountry;

begin

result := nil;

if assigned(fcountrylist) then

begin

if (fcountrylist.Count > 0) and fID >= 0 then

result := fCountrylist[fID]

end;

end;

This is the whole code of procedure TMain.save:

procedure TMain.Save;

var Country : TCountry;

begin

if fNewRecord then

begin

Country := TCountry.Create;

Country.id := fWorld.LastID;

end else

Country := fWorld.FindRow;

fMediator.CmpToWrite(Country);

if fNewRecord then

begin

fWorld.Countrylist.Add(Country);

fWorld.ID := fWorld.LastID;

fNewRecord := false

end;

end;

The only thing I have to do is scrolling into the countrylist and showing the right values in TEdit:

procedure TMain.recordprior;

var Country : TCountry;

begin

if fWorld.ID > 0 then

begin

fWorld.ID := fWorld.ID - 1;

Country := fWorld.FindRow;

fMediator.ReadToComp(Country);

end;

end;

procedure TMain.recordnext;

var Country : TCountry;

begin

if fWorld.ID < fWorld.LastID then

begin

fWorld.ID := fWorld.ID + 1;

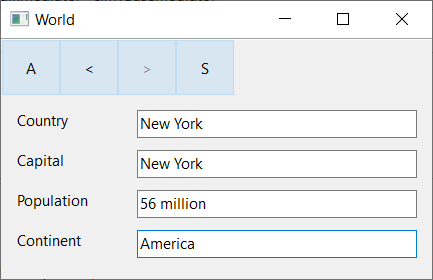
Country := fWorld.FindRow;

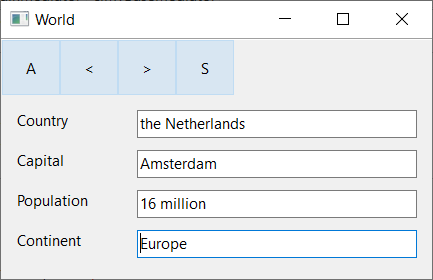
fMediator.ReadToComp(Country);

end;

end;

Now I can put new and old data to my countrylist.





But if a recorditem has to be deleted?

It is also possible with some extra code. I put an extra button in the toolbar and linked an action on it:

procedure TFrmMain.acDeleteExecute(Sender: TObject);

begin

flink.DeleteRecord;

if fLink.ID = -1 then

begin

flink.AddRecord;

ECountry.SetFocus;

end;

end;

procedure TMain.DeleteRecord;

var country : TCountry;

emptylist : boolean;

begin

emptylist := fWorld.RowDelete;

if not fWorld.Error then

begin

if emptylist then

begin

country := fWorld.FindRow;

fMediator.ReadToComp(country);

end else

AddRecord;

end else

showmessage('cannot delete item')

end;

Procedure TMain.DeleteRecord calls Tworld.RowDelete to delete an item in countrylist and reindex the id in the TCountryObject:

function TWorld.RowDelete: boolean;

var myID : integer;

begin

try

fError := false;

myID := fID;

fcountrylist.Delete(fID);

fcountrylist.Pack;

if fcountrylist.Count = 0 then

begin

result := false;

fID := -1;

end else

begin

if myID = fLastID then

begin

myID := myID - 1;

if myID = -1 then

begin

fID := -1;

result := false;

end else

begin

fID := fID - 1;

result := true;

end;

end else if myID = 0 then

begin

fID := -1;

result := false;

end else if myID < fLastID then

begin

result := true;

end;

end;

Reindex;

except

fError := true

end;

end;

**New component**

Let’s take a step further. The field ‘continent’ is repeatable when countries like United Kindom, Belgium, the Netherlands and Germany are located in Europe. You can type the word ‘Europe’ for each record, but sometimes you mistype a word (Europe, europe, Europa). It is better to select an item. For this selection I use a TComboBox.

I replace Tedit.Econtinent to TCombobox.CBContinent. I also has to tell the mediator a new component is used (fLink.Mediator.AddComp(CBContinent,'continent')). For reading and writing to the right component some modifications are made in the TBaseMediator:

procedure TMyBaseMediator.mdread(const aFieldname: string; const aValue: variant; aDec : TMyDeclarations);

var cln : TMycollection;

begin

if locate(aFieldname,cln) then

begin

if cln.fDisplayComponent is Tedit then

TEdit(cln.fDisplayComponent).Text := aValue

else if cln.fDisplayComponent is TCombobox then

Tcombobox(cln.fDisplayComponent).Text := aValue;

end;

end;

function TMyBaseMediator.mdwrite(const aFieldname : string) :variant;

var cln : TMycollection;

begin

if locate(aFieldname,cln) then

begin

if cln.fDisplayComponent is TEdit then

result := TEdit(cln.fDisplayComponent).Text

else if cln.fDisplayComponent is Tcombobox then

result := TCombobox(cln.fDisplayComponent).Text;

end;

end;

The only thing I have to do is putting items in TCombobox for selecting. If I run the application I can select a continent value for the displayed record. Scrolling to the list gives me the right and consistency values of continent.

This is the easy way. In most cases the items in TCombobox are dynamic. When using databases, TCombobox.items is listed with records from a table. The original value is a key corresponds with a value (lookup). In this example I create objContinents on the same way as objWorld.

If you can see a few properties are exactly the same in obj\* files. I can select these properties to put them in a separate unit. I call this unit objbase.pas with the following code:

type TObjectBase = class

fError : boolean;

fID : integer;

fLastID : integer;

public

property LastID : integer read fLastID write fLastID;

property ID : integer read fID write fID;

property Error : boolean read fError write fError default false;

end;

Class TWorld and TContinentTable can use this (base)class (TWorld = class(TObjectBase)). Now I save the same properties with less code. The only thing I have to is putting the file objbase into the uses in section interfaces.

Again I create the procedure init to call my object TContinentTable. I declare the object local, because only the values are needed to add in TCombobox.Items and do nothing with the test of the application.

procedure TMain.init(aContinent: TStrings);

var fContinentTable : TContinentTable;

Continent : TContinent;

begin

fContinentTable := TContinentTable.create;

try

for continent in fContinentTable.Continentlist do

aContinent.Add(continent.name);

finally

fContinentTable.free;

end;

end;

In mainform:

procedure TFrmMain.FormCreate(Sender: TObject);

begin

<some code>

flink.init(CBContinent.items);

end;

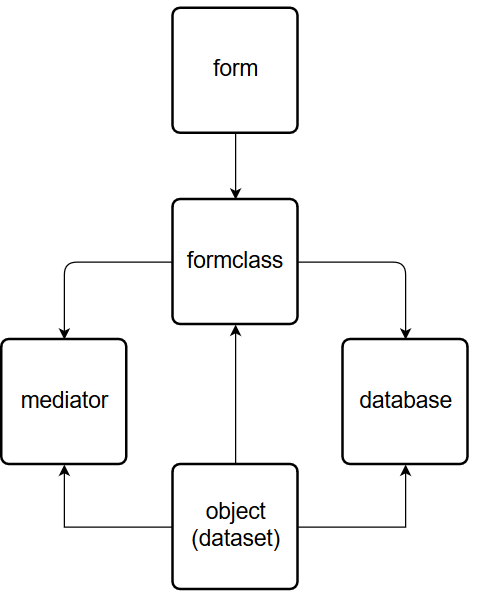
Running the application I can select one of the items without making mistakes of typing.

**Database**

All those things are nice, but when closing the application all data is gone. It is time to make data permanent with a small database. I choose SQLite because it is very small, easy to use and has drivers for almost every OS. For Lazarus I need one DDL to connect the SQLite database. The library can be found on <https://www.sqlite.org/download.html>. For my application I need the precompiled Windows driver. Download the driver for the Lazarus you using (32 / 64 bits). Unzipping the file gives two files : sqlite3.dll and sqlite3.def. I place these files in my development directory. You can also put them in Windows\system<32>.

I reference working with SQLdb to my SQLdb article in Blaise xx or on website <https://wiki.freepascal.org/SQLdb_Tutorial1>.

I explained in the first flowchart the relation to the formclass and mediator. Now a database is included and this flowchart is expanded with dbtools (https://online.visual-paradigm.com):



The database is working together with formclass and object. Still form does not know anything about databases. Objects is the communication layer for data.

I will create a new unit called clDBWorld. This unit inherit from base object TMyDatabase. These values and properties is put to communicate with database unit:

clDatabase:

TMyDatabase = class

fSQLQuery : TSQLQuery;

fSQLTransaction : TSQLTransaction;

private

fdberror: boolean;

fRecordCount: integer;

public

constructor create;

destructor destroy; override;

procedure GetTable(const aQuery : string);

function GetLastID: int64;

function dataset : tdataset;

property recordcount : integer read fRecordCount write fRecordCount;

property dberror : boolean read fdberror write fdberror;

end;

clDBWorld:

type TDBWorld = class(TMyDatabase)

strict private

fworld : TWorld;

procedure ReadValuesToObject;

public

procedure GetValues(const aTablename : string; aWorld : TWorld); overload;

procedure GetValues(aQuery : TStrings; aWorld : TWorld); overload;

procedure SaveToDatabase(const aNewRecord : boolean);

procedure RowDelete(const aID : integer);

end;

Value fWorld is the object TWorld in objWorld. This is needed to load values from a table of a database. In the formclass ‘clmain’ I create a new object fdatabase : TDBWorld. In procedure TMain.init a new line of code is includes to load the records into object TWorld (fdatabase.GetValues('countries', fWorld)). The first parameter tells which table has to be called and the second parameter links the object TWorld to the object fdatabase : TDBWorld:

procedure TDBWorld.GetValues(const aTablename: string; aWorld : TWorld);

begin

GetTable(format('select \* from %s',[aTablename]));

fWorld := aWorld;

ReadValuesToObject;

end;

procedure TDBWorld.ReadValuesToObject;

var Country : TCountry;

begin

if not dataset.Active then

dataset.Active := true;

RecordCount := dataset.recordcount;

if RecordCount > 0 then

begin

fWorld.fID := 0;

dataset.first;

while not dataset.eof do

begin

Country := TCountry.create;

Country.id := dataset.FieldByName('id').AsInteger;

Country.countryname := dataset.FieldByName('countryname').AsString;

Country.capital := dataset.FieldByName('capital').AsString;

Country.population := dataset.FieldByName('population').AsString;

Country.continent\_id := dataset.FieldByName('continent\_id').AsInteger;

fWorld.Countrylist.Add(Country);

dataset.next;

end;

end;

fWorld.fLastID := fWorld.Countrylist.Count - 1;

dataset.Active := false;

end;

I only have to open the required table in my database to read all records. After loading the table can be closed. All data is held in TWorld.

Now if I run my application and no database is found, it creates a database with table ‘countries’:

procedure TDMConnection.CreateTables;

var MyTables : TStrings;

begin

MyTables := TStringlist.create;

try

SQLConnector1.GetTableNames(MyTables);

if MyTables.IndexOf('countries') = -1 then

begin

SQLConnector1.ExecuteDirect(

'CREATE TABLE "countries" (' +

' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +

' "countryname" VARCHAR (30) NOT NULL,' +

' "capital" VARCHAR (40) NOT NULL,' +

' "population" VARCHAR (20) NOT NULL,' +

' "continent\_id" INTEGER NOY NULL DEFAULT 0);');

SQLConnector1.ExecuteDirect('CREATE UNIQUE INDEX "countries\_id" ON "countries" ("id");');

SQLTransaction1.commit;

end;

finally

MyTables.free

end;

end;

In the previous example I put continent directly in combobox ‘CBContinent’. If you can see in the table I can not save text into the table, only an integer to ‘continent\_id’. It is always better to save the ‘id’. It use less memory in table and the description connected to the ‘id’ can easily change without changing the id in the maintable. To let the application find the right ‘id’ for text in ‘CBContinent’, I create a new unit called clLookup. Now I have only one lookuptable, but the object is created for multiple tables. Therefore I use this construction:

Type TLookupItem = class

id : integer;

value : string;

end;

TLookupItemList = specialize TFPGObjectlist<TLookupItem>;

TLookupList = specialize TFPGMap<string,TLookupItemList>;

TLookuplist : TFPGMap (Delphi TDirectory) holds the name of the lookuptable with its items put in TLookupItemlist. Object TMain gets a new object (fLookup : TLookup) In TMain.init I call procedure fLookup.LoadContinent(fContinentTable.Continentlist, aContinent). The first parameters has all the records of continent put in object TContinentTable and the second parameter holds the items of ‘CBContinent’ I still have to fill the items to select one of them. One other thing I have to do is telling my object TWorld an extra value will be used to accomplish my lookup feeling of CBContinent:

type TCountry = class

id : integer;

countryname : string;

capital : string;

population : string;

continent\_id : integer; //for database/lookup purpose

continent : string;

end;

The application comes with no records found, so it turns directly to ‘append mode’. I make one record of country England with continent ‘Europe’. By saving the record I also have to call a procedure in object fDatabase to physically save the values in the database:

Unit clmain:

procedure TMain.Save;

var Country : TCountry;

begin

if fNewRecord then

begin

fWorld.IncreaseID;

Country := TCountry.Create;

end else

Country := fWorld.FindRow;

fMediator.CmpToWrite(Country);

Country.continent\_id := fLookup.KetItemKey('continent',Country.continent);

if fNewRecord then

begin

fWorld.Countrylist.Add(Country);

fWorld.ID := fWorld.LastID;

fDatabase.SaveToDatabase(true);

fNewRecord := false

end else

fDatabase.SaveToDatabase(false);

if fdatabase.dberror then

showmessage('record nod saved')

end;

Unit clDBWorld:

procedure TDBWorld.SaveToDatabase(const aNewRecord: boolean);

var Country : TCountry;

obJID : integer;

MyQuery : TStrings;

begin

Myquery := TStringlist.create;

dberror := true;

try

if aNewRecord then

begin

MyQuery.AddText('INSERT INTO countries (countryname, capital, population, continent\_id)');

MyQuery.AddText('VALUES (:name, :capital, :population, :continent\_id)');

if fWorld.Countrylist.Count = 0 then

objID := 0

else

objID := fWorld.Countrylist.Count - 1;

end else

begin

MyQuery.AddText('UPDATE countries SET');

MyQuery.AddText('countryname = :name, capital = :capital,');

MyQuery.AddText('population = :population, continent\_id = :continent\_id');

MyQuery.AddText('WHERE id = :id');

objID := fWorld.fID;

end;

Country := fWorld.Countrylist[objID];

fSQLQuery.SQL.Text := MyQuery.Text;

fSQLQuery.Params[0].AsString := Country.countryname;

fSQLQuery.Params[1].AsString := Country.capital;

fSQLQuery.Params[2].AsString := Country.population;

fSQLQuery.Params[3].AsInteger := Country.continent\_id;

if not aNewRecord then

fSQLQuery.Params[4].AsInteger := Country.id;

fSQLQuery.ExecSQL;

dberror := fSQLQuery.RowsAffected = -1;

if DMConnection.SQLConnector1.ConnectorType <> 'SQLite3' then

fSQLTransaction.Commit

else

dmConnection.SQLTransaction1.Commit;

if aNewRecord then

Country.id := GetLastID;

finally

MyQuery.free;

end;

end;

If I close the application and look with a SQLite editor (<https://sqlitestudio.pl>) I see my record is saved:



Now running the application again I have data to show. By displaying the continent a conversion from ‘ID’ to text is needed to show the continent on screen properly. This is done with this procedure:

procedure TMain.init(aContinent: TStrings);

begin

………….

if fWorld.Countrylist.Count > 0 then

Display;

………….

end;

procedure TMain.Display;

var Country: TCountry;

begin

Country := fWorld.FindRow;

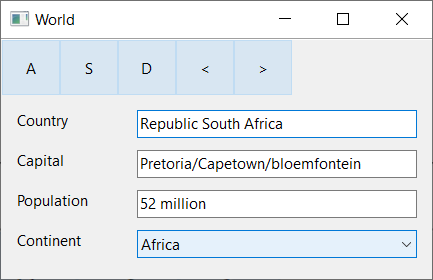
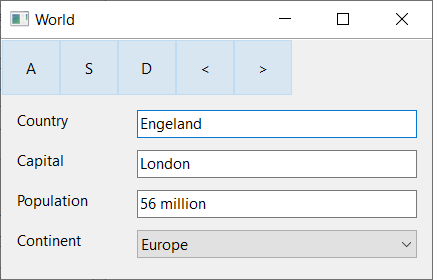
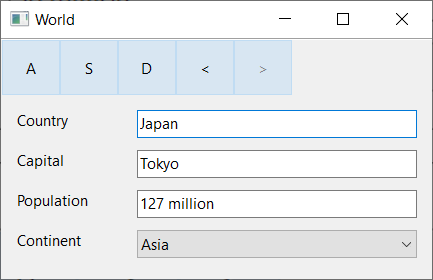
//lookup

Country.continent := fLookup.GetValueFromIndex('continent',Country.continent\_id);

fMediator.ReadToComp(Country);

end;

When scrolling in the object procedure Display is searching in object fLookup to display the right continent on screen.



The only thing I have to do is telling the application a record can be deleted physically from a database. If you know how to save it, it is easy to use the code almost the same:

procedure TMain.DeleteRecord;

var country : TCountry;

emptylist : boolean;

begin

country := fWorld.FindRow;

fDatabase.RowDelete(country.id);

if fdatabase.dberror then

showmessage('record not deleted')

else begin

………

end;

end;

procedure TDBWorld.RowDelete(const aID: integer);

begin

dberror := true;

fSQLQuery.SQL.Text := 'DELETE FROM countries WHERE id = :id';

fSQLQuery.Params[0].AsInteger := aID;

fSQLQuery.ExecSQL;

dberror := fSQLQuery.RowsAffected = -1;

if DMConnection.SQLConnector1.ConnectorType <> 'SQLite3' then

fSQLTransaction.Commit

else

dmConnection.SQLTransaction1.Commit;

end;

**TLookup advanced**

It is nice to load the lookup items from an object, but it is better to load the items from a database. It makes the items more flexible to add. The class Tlookup gets baseclass TMydatabase as base to load data from a table. Now I have TSQLQuery to load data into my items:

TLookup = class(TMyDatabase)

fLookupList : TLookupList;

private

procedure GetItems(const aItems: TStrings);

public

…………

procedure LoadContinent(aItems : TStrings);

…………

end;

implementation

{ TLookup }

procedure TLookup.LoadContinent(aItems : TStrings);

begin

try

GetTable('SELECT id, continent FROM continents');

GetItems(aItems);

TableCommit;

finally

fSQLQuery.Active := false;

end;

end;

procedure TLookup.GetItems(const aItems: TStrings);

var

Item: TLookupItem;

begin

while not fSQLquery.eof do

begin

Item := TLookupItem.Create;

Item.id := fSQLquery.Fields[0].AsInteger;

Item.value := fSQLquery.Fields[1].AsString;

Add('continent', Item);

aItems.Add(Item.value);

fSQLquery.Next;

end;

end;

Procedure TableCommit() from TMydatabase is called to free the transactions after read and close the dataset (table). The rest is exactly the same way of working with TLookup as before. In this case TContinentTable not needed anymore. Later on I still use it to read the data for adding / modifying / deleteing items of the lookuptable. The only thing I have to do is creating a table with records to a database in unit

if MyTables.IndexOf('continents') = -1 then

begin

SQLConnector1.ExecuteDirect(

'CREATE TABLE "continents" (' +

' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +

' "continent" VARCHAR (30) NOT NULL);');

SQLTransaction1.commit;

SQLConnector1.ExecuteDirect(

'INSERT INTO continents (continent) VALUES ' +

'("Asia"),("North America"),("South America"),("Autralia"),' +

'("Europe"),("Africa"),("Antartica");');

SQLTransaction1.commit;

end;

**Expanding**

To see if it really works I expand the application with a new form. Countries has cities with its characteristics. A new form has its own units (formclass, objectclass, dbobjectclass). I will expand the database with a view tables (the text within brackets is already put in code):

Procedure TDMConnection.CreateTables;

var MyTables : TStrings;

begin

MyTables := TStringlist.create;

try

SQLConnector1.GetTableNames(MyTables);

if MyTables.IndexOf('countries') = -1 then

begin

{countrysettings}

end;

if MyTables.IndexOf('cities') = -1 then

begin

SQLConnector1.ExecuteDirect(

'CREATE TABLE "cities" (' +

' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +

' "country\_id" INTEGER NOT NULL,' +

' "cityname" VARCHAR (30) NOT NULL,' +

' "major" VARCHAR (30) NOT NULL,' +

' "capital" BOOLEAN NOT NULL DEFAULT FALSE,' +

' "sightseeing" VARCHAR (100), ' +

' "square" INTEGER NOT NULL DEFAULT 0,' +

' "poprange\_id" INTEGER NOT NULL,' +

' "township\_id" INTEGER NOY NULL DEFAULT 0);');

SQLConnector1.ExecuteDirect('CREATE UNIQUE INDEX "city\_id" ON "cities" ("id");');

SQLTransaction1.commit;

end;

if MyTables.IndexOf('continents') = -1 then

begin

{continent settings}

end;

if MyTables.IndexOf('popranges') = -1 then

begin

SQLConnector1.ExecuteDirect(

'CREATE TABLE "popranges" (' +

' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +

' "rangename" VARCHAR (30) NOT NULL);');

SQLTransaction1.commit;

SQLConnector1.ExecuteDirect(

'INSERT INTO popranges (rangename) VALUES ' +

'("1 - 1.000"),("1.001 - 10.000"),("10.001 - 25.000"),("25.001 - 50.000"),' +

'("50.001 - 75.000"),("75.001 - 100.000"),("100.001 - 250.000"),("250.001 - 500.000"),' +

'("500.001 - 1.000.000"),("1.000.001 - 2.500.000"),("2.500.001 - 5.000.000"),("5.000.000 and more");');

SQLTransaction1.commit;

end;

if MyTables.IndexOf('townships') = -1 then

begin

SQLConnector1.ExecuteDirect(

'CREATE TABLE "townships" (' +

' "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,' +

' "townshipname" VARCHAR (30) NOT NULL);');

SQLTransaction1.commit;

SQLConnector1.ExecuteDirect(

'INSERT INTO townships (townshipname) VALUES ' +

'("Village"),("Town"),("City");');

SQLTransaction1.commit;

end;

finally

MyTables.free

end;

end;

I create objcities as my new object to the form. Most of the code will be copied from objworld. The only thing really changed is this peace of code:

type

TCity = class

id : integer;

country\_id : integer; //for database purpose

cityname : string;

major : string;

capital : boolean;

sightseeing : string;

square : integer;

poprange\_id : integer; //for database purpose

population : string;

township\_id : integer; //for database purpose

township : string;

end;

TCitylist = specialize TFPGObjectList<TCity>;

I rename the class to TCities and change all TWorld in procedures, change @countrylist to @Citylist and @Country to @City (@ means al characters before the word).

Create file clDBCity and copy all code from clDBWorld and change the name of the class from TDBWorld to TDBCity and all related objects and variables related to file objCity.

As you can see in class Tcity, variable ‘square’ is typed as an integer. Some variables ‘string typed’ has unlimited characters, but the fields in a database are always limited to store. When a query is called from a database, a lot of extra’s options are also extracted from the tabledefinition like length and datatype ((var)char, integer, boolean, float). It should be nice if some of the GUI components knows about this atributes. I expanded the cldatabase file (related to cldbcity) to load these attributes:

interface

uses

Classes, SysUtils, sqlDB, dConnection, sqlite3conn, DB, fgl, typinfo;

type

TobjFielddef = class

fieldname : string;

varlength : integer;

datatype : string;

end;

TObjFielddeflist = specialize TFPGObjectList<TobjFielddef>;

{ TMyDatabase }

TMyDatabase = class

{previous code}

Private

{previous code}

fObjFielddeflist : TObjFielddeflist;

protected

procedure LoadAttributes;

public

{previous code}

property ObjFielddeflist : TObjFielddeflist read fObjFielddeflist write fObjFielddeflist;

end;

implementation

{ TDatabase }

constructor TMyDatabase.create;

begin

fObjFielddeflist := TObjFielddeflist.create;

{previous code}

end;

destructor TMyDatabase.destroy;

begin

fObjFielddeflist.free;

fSQLQuery.free;

inherited destroy;

end;

procedure TMyDatabase.LoadAttributes;

var index : integer;

Fielddef : TobjFielddef;

begin

fSQLQuery.First;

for index := 0 to fSQLQuery.Fields.Count -1 do

begin

Fielddef := TobjFielddef.Create;

Fielddef.fieldname := fSQLQuery.Fields[index].FieldName;

Fielddef.varlength := fSQLQuery.Fields[index].DataSize;

Fielddef.datatype := GetEnumName(TypeInfo(TFieldType), Ord(fSQLQuery.Fields[index].DataType));

ObjFielddeflist.Add(Fielddef);

end;

end;

procedure TDBCity.ReadValuesToObject;

var City : TCity;

Fielddef : TobjFielddef;

begin

if not dataset.Active then

dataset.Active := true;

{previous code}

//this peace of code must be after reading the records.

//If there are no records the definitions can still read

if dataset.active then

LoadAttributes;

{previous code}

end;

I can changed this also in unit clDBWorld. The next file to create is clCity.pas. As in cldbCity I copy all code from clMain and change the classname to TFCity and change all reladed variables and objects to the objects of objcity.pas and cldbcity.pas. There is one property extra in clCity.pas : country\_id. All created records has to be related to a country. Otherwise all records are shown for every country.

To give extra functionality to the GUI-components the fielddefinitions must be read. I put this extra code to procedure init() of the formclass:

if fdatabase.ObjFielddeflist.Count > 0 then

begin

for Fielddef in fdatabase.ObjFielddeflist do

fMediator.SetAttributes(Fielddef.fieldname, Fielddef.datatype, Fielddef.varlength);

end;

As you can see, a procedure in Mediator is called. This can be in the baseobject of the mediator. Now every time a new Tmediator is created the attributes can give to the components:

procedure TMyBaseMediator.SetAttributes(const aFieldname, aDatatype: string;

const aVarlength: integer);

var cln : TMycollection;

begin

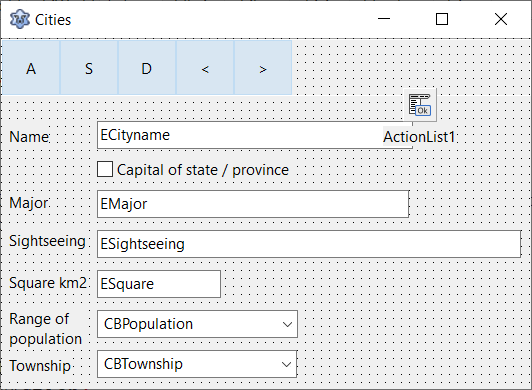
cln := nil;

if locate(aFieldname,cln) then

begin

if cln.fDisplayComponent is TEdit then

begin

 TEdit(cln.fDisplayComponent).MaxLength := aVarLength;

if aDatatype = 'ftInteger' then

begin

TEdit(cln.fDisplayComponent).NumbersOnly := true;

if aDatatype = 'ftInteger' then

cln.fDatatype := dcInteger;

End;

end;

end;

end;

This change gives the program the opportunity to use the right datatype of an variable in object and database without giving mdread / mdwrite which datatype must be used.

Now the form can be created. I put some labels, Edits, comboboxes, toolbar and actionlist on screen.

I copy the code in section ‘implementation’ of fmain.pas to fcyty.pas and link the action events to an action.

In fmain.pas I create a new button spCities and open event ‘Onclick’ In clmain.pas a new procedure LoadCityform is created:

procedure TMain.LoadCityForm;

var frm : TFrmCity;

Country : TCountry;

begin

frm := TFrmCity.Create(nil);

try

Country := fWorld.FindRow;

frm.Link.country\_id := Country.id;

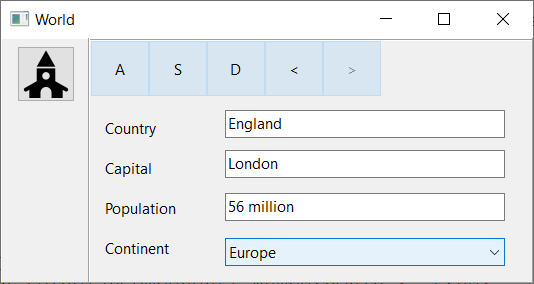
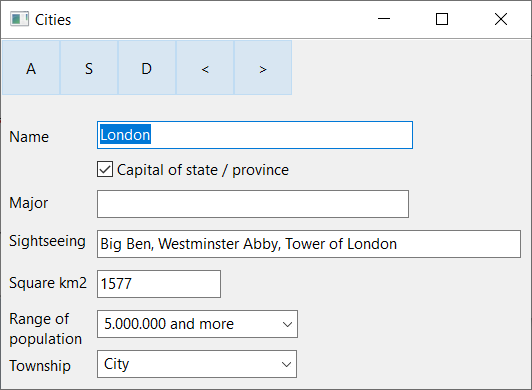
frm.ShowModal;

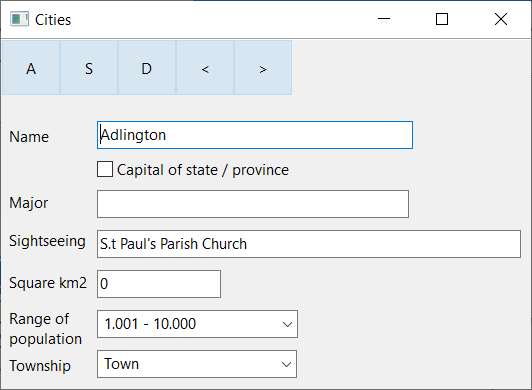
finally

frm.free;

end;

end;





**Master / Detail**

It is nice to see the countries of the world, but I want to see the cities as well. That is why I create a view in the first form with all related cities. It can be done with a grid or listview. Listview is the easiest way. In report mode it can hold multiple text and as reference I can save a pointer on each node as reference.

Let’s put a listview on the mainscreen. All GUI components are controlled by the mediator. That’s why I tell the mediator to link the listview:

fLink.Mediator.AddComp(LVDetail,'list1');

I add a new component in TbaseMediator to find component listview:

uses classes, fgl, sysutils, controls, stdctrls, comctrls;

The listview is blank, and to see which fields are used I created a new procedure in Tbasemediator to show the givven fields. This procedure can used in any form linked to the mediator.

procedure TMyBaseMediator.LVHeader(const aFieldname: string; aHeader: array of string);

var cln : TMycollection;

LV : TListview;

Column : TListColumn;

index : integer;

fielddef : TStringArray;

begin

cln := nil;

if locate(aFieldname,cln) then

begin

LV := TListview(cln.fDisplayComponent);

LV.ViewStyle := vsReport;

for index := 0 to length(aHeader) - 1 do

begin

fielddef := aHeader[index].split(';');

Column := LV.Columns.Add;

Column.Caption := fielddef[0];

if length(fielddef) > 1 then

Column.Width := StrToIntDef(fielddef[1],0);

end;

end;

end;

I Call the procedure fMediator.LVHeader ('list1', ['cityname;200', 'population;150','township;150']) in procedure init() of unit clmain. The first parameter tells which component is called. The second parameter is an open array. This handles to functions: a fieldname and a field length separated by ‘;’. If the field length is not used the base length of the column is used.

Now I need data. As for all data I use an object to hold the data from a query. Object TWorld in unit objWorld is used for my data on mainform. It is also the place for my data of listview :

TCityDetail = class

country\_id : integer; //for database purpose only

cityname : string;

population : string;

townrange : string;

end;

TCityDetaillist = specialize TFPGObjectList<TCityDetail>;

{ TWorld }

TWorld = class(TObjectBase)

fcountrylist : TCountrylist;

fCityDetaillist : TCityDetaillist;

private

procedure Reindex;

public

constructor create;

destructor destroy; override;

function FindRow : TCountry;

function RowDelete : boolean;

procedure IncreaseID;

property Countrylist : Tcountrylist read fCountrylist;

property CityDetaillist : TCityDetaillist read fCityDetaillist;

end;

Now I can call fdatabase.GeDetaillist(fWorld) for all data represented from my query:

procedure TDBWorld.GeDetaillist(aWorld: TWorld);

var CityDetail : TCityDetail;

begin

try

if fSQLQuery.Active then

fSQLQuery.Active := false;

fSQLQuery.SQL.clear;

with fSQLQuery.SQL do

begin

Add('SELECT c.id, c.country\_id, c.cityname,');

Add(' r.rangename, t.townshipname FROM cities c');

Add(' LEFT OUTER JOIN popranges r on r.id = c.poprange\_id');

Add(' LEFT OUTER JOIN townships t on t.id = c.township\_id');

Add(' ORDER BY c.country\_id, c.cityname');

end;

fSQLQuery.Active := true;

aWorld.CityDetaillist.Clear;

while not fSQLQuery.EOF do

begin

CityDetail := TCityDetail.create;

CityDetail.country\_id := fSQLQuery.Fields[1].AsInteger;

CityDetail.cityname := fSQLQuery.Fields[2].AsString;

CityDetail.population := fSQLQuery.Fields[3].AsString;

CityDetail.townrange := fSQLQuery.Fields[4].AsString;

aWorld.CityDetaillist.Add(CityDetail);

fSQLQuery.Next;

end;

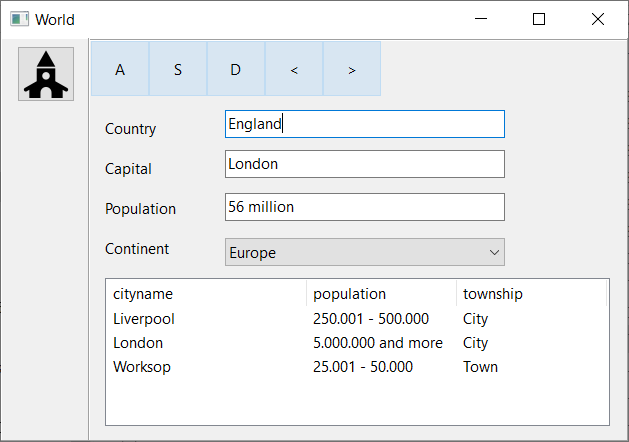
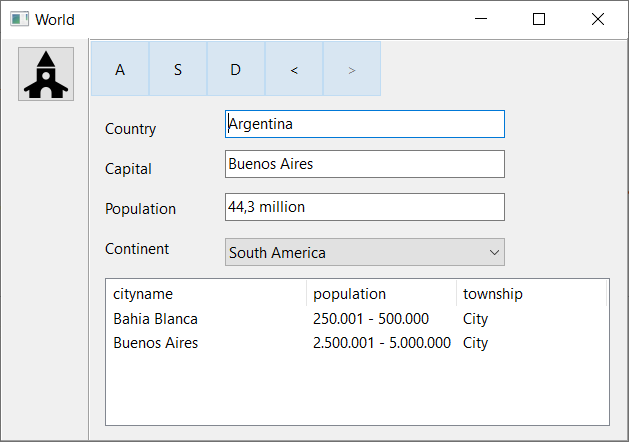
finally

fSQLQuery.Active := false;

end;

end;

The last thing I have to is showing my detail data in listview for each country with procedure fMediator.ReadDetail('list1', Country.id, fWorld.CityDetaillist). The first parameter tells which component will be used. The second parameter is the filter which data is connected to the country. The last parameter holds all data from the query.



**Conclusion**

It is a lot of work but objects are working. As far as I know I have manage to separate the form from all data. This works on every OS where Lazarus is running. The only thing to think is a connection to a database. On Unix other client libraries are used than Windows.

This way of working can also be used by SOAP / Rest servers. When data is called from these servers, it can be put in the objects and your application is still working. I managed to read a spreadsheet to object to see what’s happening. Once a spreadsheet is open, it is read only for other users, but I created an extra object that’s hold extra data outside the spreadsheet.