In this series of articles, some programming ideas will be discussed that can be applied when making large database applications in Delphi, i.e. database applications that have many screens, which operates on a database with possibly many tables. Many aspects of such programs will be discussed; No code will be presented other than some small code snippets to illustrate the ideas.

1 Some background

In this series of articles, a series of programming ideas will be explained that were developed in the company of the author. These ideas are mainly concerned with the development of large database programs, and the problems that are faced in such programs. These ideas were developed when porting an old Visual Basic program that operated with an MS-Access database to a Delphi program which operates in a Multi-Tier architecture with Interbase as a database. At the same time, 2 new applications were developed which now operate under a similar setup.

The Visual Basic application was a program to handle all aspects of an administration of a school: Managing pupils data and personnel, plus an interface with the Ministry of Education (in Belgium). The old Visual Basic program has about 130 screens, resulting in an executable of 19 Mb, with a database of about 170 highly interconnected tables.

While the Visual Basic program works fine in a single-user environment, performance in a multi-user environment leaves to be desired at times. Furthermore, the development team had bumped into some design limits of Visual Basic and MS Access.

So it was decided to upscale this application to a full client-server environment, and at the same time develop a bookkeeping module in the program, oriented towards the needs of the schools. The bookkeeping module would be sold as a separate program as well. Since the author is thoroughly familiar with Delphi, it was decided to do this in Delphi (apart from some other arguments in favour of Delphi). These programs would need to have the same look and feel, and behave in the same way. Later, similar applications would follow.

Faced with this task, the author set about creating what was called an 'Application Framework' in Delphi: A series of Delphi components and IDE wizards that would make the development of the new applications faster, and at the same time present the user with a uniform and powerful interface. The end result is exactly what the name says: It is a framework in which an application can be developed:

- Design time, the programmer has an extended set of tools (components, wizards) to his disposal for the development of (n-tier) database programs.
- Run-time, the application framework is a finished program that features the basic functionality of all our applications; It features a client and a server application, which are capable of user and security management, plus parametrization of the application.

The application framework covers several aspects of database programs, and the approaches taken reflect in some ways the needs of the company's customers (schools), but are applicable to most large database programs. Most of the features are targeted towards quick and efficient retrieval and fast editing of large amounts of data in a multi-user setting:

Modularity The end-user programs consist of a series of modules (actually Delphi packages) that are programmed separately, and which are dynamically loaded at run time by the framework routines. This reduces program size, and reduces the size of updates, making updates also small and quick to download.

Security The framework is capable of authenticating users, and handling the following security aspects:

1. Access to screens can be restricted.
2. In screens, insert/delete/modify/print permissions can be controlled.
3. The available data can be restricted by defining filters in each screen. These filters are in effect as soon as the screen is opened, in effect restricting the data the user has access to. The filters are completely designable at run-time by the system administrator.

All this is settable on a per user-basis, with the possibility of grouping this in profiles.

Customization of data views Most screens that have an overview of data (e.g. a list of clients in a grid) on them can be customized: The user can define filters on the data using conditions on data in *any* related table, not just on the fields shown. Furthermore, any field out of any related table can be shown in the overview. The layout of the grids can be customized, and the contents can be sorted on any field in the grid.

Fast data entry Users can fully customize default values for all fields that can be edited. Furthermore, the last inserted value for a field can be retrieved when editing another record. Tab stops and tab order can be configured, combined with default values this allows for fast data entry.

Customization of drop-downs

Searching Users can perform an incremental search on almost any field displayed in an overview.

Navigation in each screen links to other screens, with related information, are presented, in a uniform manner.

Printing Each screen has by default one or more reports associated with it, which can be printed. The user can add his own reports to this.

Many other things can be customized by the user, and are stored on a per-user basis in the database.

All of the above things can be enabled and configured by the programmer with the help of design-time properties; No extra programming is required.

To be able to realize this, full use was made of the OOP features of Delphi, the RTTI (Run-Time Type Information) and of the Opentools API for customizing the IDE. The ideas used when developing all these things will be discussed in this and following articles.

2 Using OOP all the way

While Delphi features a standard set of components that allow to program almost any database-oriented program, nevertheless quite often some functionality which would make the program more user-friendly is missing, and must be implemented again for each form in the application. Luckily, Delphi is object oriented, and this feature offers a simple way out of this problem.

Creating descendents of standard Delphi components is something which is usually done when new components are built. However, also for the end-developer, creating simple descendents of existing components can be useful. In fact, when developing several applications with many screens that must act and feel alike, it is a good idea to create descendents

of all basic components that will be used regularly (even when buying a set of third-party components), and only to use these descendents. There are two good reasons to do this:

The first reason is that when a new feature should be introduced, this can be done anytime by adapting the descendent component, just requiring a recompile of the application to enable the feature throughout the whole application.

This is true not only for components, but also for forms: ensuring a uniform look and feel would be a lot of work if form inheritance was not used. With form inheritance, it is quite easy. For instance, it seems logical that the `OnCloseQuery` would check whether there is any unsaved data in the form by checking all datasets on the form, and either prompt the user for some action, or post the modified data. A kind of functionality to make the form 'DB-Aware'. In the author's opinion, this kind of functionality should already be implemented by Borland, in a `TDBForm` descendent of TForm. Luckily, Delphi offers enough mechanisms to handle this inherent lack of functionality.

But form inheritance can be used even more than just for that: In cases were many similar screens should be developed, developing them as inherited screens of a single parent screen that contains the common functionality, is good practice and can save a lot of time and avoid duplicated code.

Initially, the descendent components and forms need not introduce any new functionality. Using them makes introducing new features a lot easier: Imagine that a normal `TComboBox` component is used on many places. At a later time, it is decided to add a MRU feature to the comboboxes. (i.e. the combobox keeps track of what items are used most, and presents these on top of the list)

Now two situations can occur:

1. If a plain combobox was used, this means that they must all be replaced with a new comboboxes that has the MRU functionality. It means copying all changed properties to the new combobox etc.
2. If a descendent class of the standard combobox was used, the MRU functionality can be implemented in that descendent class. Recompiling the application is sufficient to introduce the MRU functionality in all comboboxes.

Clearly, the latter option is much more convenient and saves a lot of work.

Another example would be the `TDBLookupComboBox`: When the user presses a certain key combination, the contents of the lookup dataset should be refreshed. This can be implemented in 2 ways:

1. To each `TDBLookupComboBox` in the application, a `OnKeyPress` event handler is attached, which checks for the keycombination and which then executes the following code:

   ```
   With (Sender as TDBLookupComboBox) do
      If Assigned(DataSource) and
         Assigned(DataSource.Dataset) then
        DataSource.Dataset.Refresh;
   ```

   This code must be attached to each `TDBLookupComboBox` in the project.
2. A descendent of `TDBLookupComboBox` is made, which interceps the key, and executes the above code. In all forms, only this descendent is used.

Clearly, when lots of forms are involved, the second method is less work, and less error-prone as the first.

The second reason for creating descendents of basic components is speed of development, and executable size and speed: if a component with certain property values will be used often, then creating a descendent with these values pre-set is a good idea; Setting the default property to these pre-set values reduces the size of the form files. For example:

Suppose that it is decided to give all buttons standard a certain width and height, say 100x25. Then it is good to create a descendent of TButton which pre-sets these values:

```
Type
  TMyButton = Class(TButton);
  Public
    Create(AOwner : TComponent); override;
  Published
    Property Width : Integer Default 100;
    Property Height : Integer Default 25;
  end;

Constructor TMyButton.Create(AOwner : TComponent);

begin
  Inherited;
  Width:=100;
  Height:=25;
end;
```

If all buttons are of type `TMyButton` and their height and width are not changed in the forms, then the height and width will not be written in any form file, and will not need to be loaded when the form is created. So the binary is smaller and faster.

Another example is a Combobox: If a lot of boolean fields are used in the database, then it may be good idea to create a descendent of `TDBComboBox` e.g. `TBooleanCombobox` which, upon creation, adds to its items 2 strings: 'Yes' and 'No', and which sets an appropriate width and dropdowncount. Adding a `Stored False` modifier to the `Items` property will cause the items not to be streamed. Instead, they are filled in when the component is created at runtime. The binary becomes smaller and faster, and the developer has less work: These standard properties need not be set.

Many more examples can be given. Deciding which components to supply with some preset values, depends on the nature of the project. In general any component that will appear with the same properties on many forms, is a good candidate.

3 Writing Modular Applications

When confronted with a big monolithic application of 20 Mb, the problem of distribution of updates arises. As an example: An update of the current Visual Basic application in our company is about 9 Mb. and is available for download from our website; For many of our customers, their infrastructure is not adapted to such large downloads; many schools still have a 14.4K modem, making such a download a long process, which is often interrupted. As a result, they request a CD-ROM containing the update; Updates come at intervals of a month, making this an expensive procedure.

Making a modular application is a solution for this: When updates are needed, only the changed module needs to be sent to the clients. The size of a module is typically less than 200Kb, making a download very fast; By regular mail a diskette can be used, which is less expensive than a CD-ROM.

Making a modular application has also other advantages: The functionality of the application can be split over several modules. Clients that require only a limited functionality, pay for the modules that they use. If at a later time they decide that they need more functionality, the remaining modules can be quickly downloaded and installed.

For example, the bookkeeping program has a module 'Invoicing' which is separate from the basic bookkeeping module. Clients that do not need the Invoicing module pay a smaller price, but can decide at any time to pay more and install the invoicing module.

This can of course also be accomplished by creating separate executables, but then the user's notion of a single, extendable MDI application is lost.

When creating a modular application, one has the choice: The separate modules can be developed as dynamically loadable libraries, or as Delphi packages. Delphi packages are of course a special kind of loadable library, but have some more functionality and solve some issues that exist with normal libraries.

First of all, when using reference counted ansistrings, all libraries must be compiled with the `sharedmem` unit, which replaces the normal memory manager of borland with another memory manager that uses shared memory. This is not a real problem, but is something that must be taken care of.

A more serious problem is the fact that when passing classes between libraries, typecasts and the `is` or `as` operators don't work properly.

Consider the following simple library:

```
library testdll;

uses SysUtils,Classes,Libc;

Function getcomponent : TObject;

begin
  Result:=TComponent.Create(Nil);
end;

exports
  getcomponent;

end.
```

And the following simple testprogram:

```
program dlltest;

uses Classes;

Function GetComponent : TObject ; external 'libtestdll.so' name
'getcomponent';

Var
  O : TObject;

begin
  O:=GetComponent;
  Writeln (O is TComponent);
end.
```

After compilation of the library and the program, running the program will output `FALSE` instead of the expected `TRUE`. After all, The result of the `GetComponent` call is a `TComponent` !

The problem is in the implementation of the `is` (and consequently, `as`) operator in Object Pascal. The `is` operator checks whether the `VMT` pointer (Virtual Method Table pointer) of the object equals the VMT pointer of the class it tests on.

In the program, the VMT of the class to test on is the VMT of `TComponent` *in* the application. The VMT of the object is the VMT of `TComponent` but *in the library*. These two tables are on different locations in memory, so the `is` operator decides that `O` is not of type `TComponent`.

This presents a more serious problem of libraries than the need for a shared memory manager: It means that classes cannot be interchanged between libraries. Only simple data can be passed. This is of course not very convenient, and severely limits the possibilities of using libraries.

Luckily, when using packages, this restriction is lifted completely; basically because the VMT of each class (whether used in a package or in the executable loading the package) will reside in 1 and only 1 package. For instance, the VMT of `TComponent` resides (for Delphi 5) in vcl50.bpl. Since both the program and package use the `vcl50.bpl` package (obviously, the program should be compiled with run-time packages), the VMT's will be equal.

So, using run-time packages turns out to be a conventient way to write modular programs. The structure of our program will look as in figure figuur 1 op pagina 7.

The main program is nothing but a MDI parent form, which features a menu, toolbar and a statusbar. It knows of no other forms, i.e. in the uses clause of the main form, no units with other forms are present. The main form just knows what is called the 'form manager' and the 'module manager'. The module manager and form manager reside in the application packages: These 2 managers are responsible for loading modules, and creating forms.

Secondly, there exists a series of modules, which make up the actual application: They contain the MDI child forms that the main program will manage. A system of form registration enables the main program to ask for a list of available forms, with which it creates a menu. (this will be discussed later).

Each module registers a series of forms with the form manager: At program startup, all known modules are loaded and unloaded once. The loading of the module will register all forms that the module contains. So after all known modules have been loaded, the main application has a list of forms it can show. Each form has a unique menu entry associated with it, and this is used to construct a menu, dynamically.

When the user chooses a menu entry, the module in which the form is located is loaded, and the form is created and shown. After the form has been closed and destroyed, the module can be unloaded again.

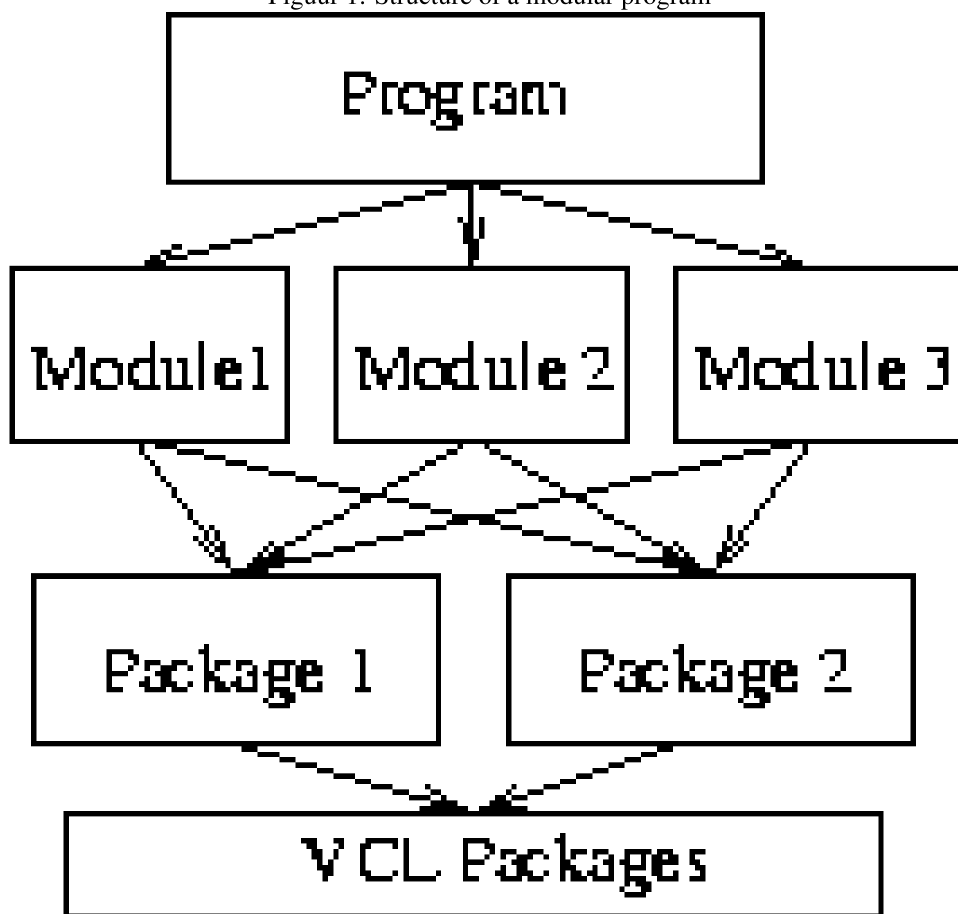
Lastly, the application packages contain all the common code for all the modules: specialized forms, components and so on.

All this is of course built on top of the VCL packages.

4 Managing modules

The task of loading and unloading modules (i.e. Delphi packages) is delegated to an object, the Module Manager. The module manager is nothing but a list of module names, and with each module a handle is associated and a reference count.

Figuur 1: Structure of a modular program



The handle of the module is the handle returned by the `LoadPackage` call:

```
function LoadPackage(const Name: string): HMODULE;
```

This function is used to load a given module. It will take care of all possible module initialization (i.e. the initialization code of all units in the module will be executed.

When unloading a module, the `UnloadPackage` call is used:

```
procedure UnloadPackage(Module: HMODULE);
```

This will execute all finalization code of the units in the package, and unload the package from memory.

When the program needs a form, the module in which this form resides is 'loaded'. If it already was loaded, the reference count of the module is raised. When the form is destroyed, the module is 'unloaded': The reference count is decremented. If the reference count reaches zero, i.e. no more forms of this module are used, the module is unloaded.

At program startup, the module manager reads from the Windows registry, or from a ini-file the locations of several known modules.

What happens then depends on how the functionality of the modules is built up: Somehow, the application must know what is in each of the modules that it has at its disposal. In the case of our application, it is the list of forms that is contained in each of the modules. In order to build this list, there are 2 possibilities:

1. Read a prepared list from some file.
2. Dynamically load the module, and let all forms register themselves, after which the module is unloaded again.

The former approach is faster, the latter approach is less error-prone, since the list of available forms is always up-to-date.

The following unit implements a basic 'form manager':

```
unit frmmgr;

interface

Uses Classes;

procedure Registerform (Name,Description : String);
Procedure UnregisterForm(Name : String);

Var
  FormList : TStringList;

implementation


procedure Registerform (Name,Description : String);

begin
  FormList.Values[Name]:=Description
end;
```

```
Procedure UnregisterForm(Name : String);

Var
  I : Integer;

begin
  I:=FormList.IndexOfname(Name);
  If I<>-1 then
    FormList.Delete(i);
end;

Initialization
  FormList:=TstringList.Create;

Finalization
  FormList.Free;
end.
```

It supports 2 methods, just enough to register a module, and unregister it.

This unit goes into a separate package formmanage. Then, 2 packages are built. The first package contains 2 'forms' and registers them:

```
unit form1;

interface

implementation

uses
  frmmgr;

Initialization
  RegisterForm('Form 1','Description 1');
  RegisterForm('Form 2','Description 2');
end.
```

It goes into a package PForm1. This package should have the 'formmanage' package in its list of required packages.

A second unit contains and registers 1 'form':

```
unit form2;

interface

implementation

uses frmmgr;

Initialization
  RegisterForm('Form 3','Description 3');
end.
```

It goes into a second package PForm2.

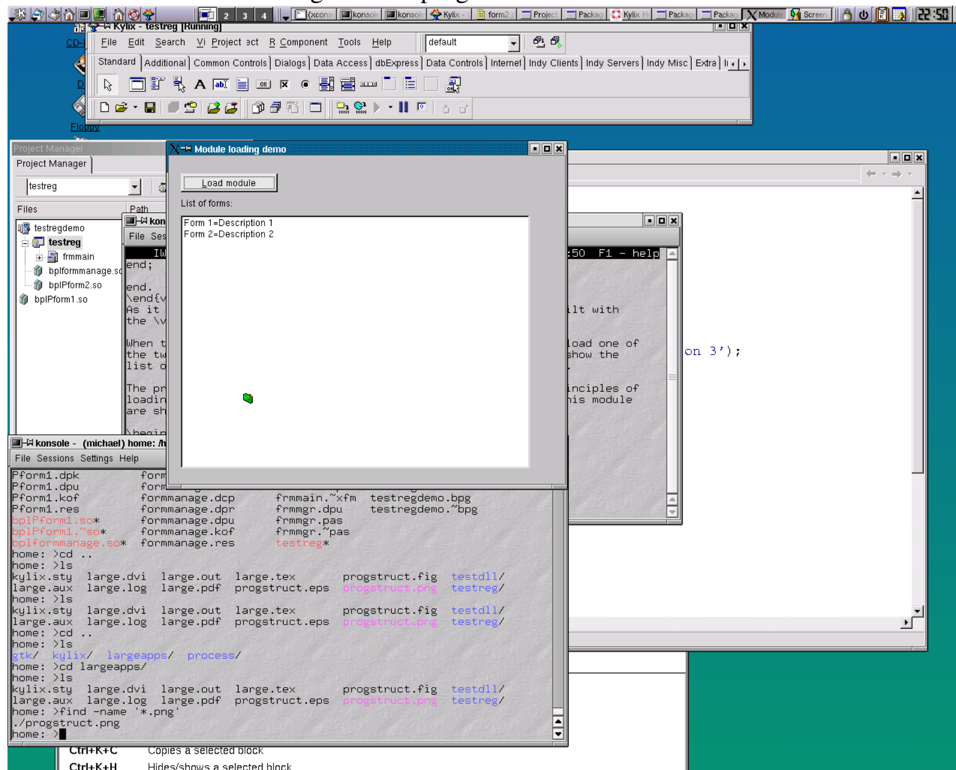
After that, a program is made which will load these 'packages' and which will show the registered forms:

```
unit frmmain;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls;

type
  TMainForm = class(TForm)
    BLoad: TButton;
    ODpackage: TOpenDialog;
    Mforms: TMemo;
    LMForms: TLabel;
    procedure BLoadClick(Sender: TObject);
  private
    procedure ShowForms;
  end;

var
  MainForm: TMainForm;

implementation

uses frmmgr;

{$R *.xfm}

procedure TMainForm.BLoadClick(Sender: TObject);
begin
  With ODPackage do
    If Execute then
      begin
      If LoadPackage(FileName)<>0 then
        ShowForms;
      end;
end;

procedure TMainForm.ShowForms;

begin
  MForms.Lines:=FormList;
end;

end.
```

As it can be seen, this is a very simple program. It should be built with the `formmanage` run-rime package.

When the program is run, the button 'Load Module' can be used to load one of the two 'modules', PForm1 or PForm2. The memo will show the list of registered forms after the module was successfully loaded.

Figuur 2: A program to load modules



The program in action can be seen in figuur 2 op pagina 11. All the principles of loading modules and obtaining a list of objects (e.g. forms) in this module are shown using this little demo program.

In our real-world application, we have a form manager in the client application, a module manager in our server application (it registers datamodules which contain data providers for our client application) and a frame manager which registers frames that are used when managing preferences (later more about this subject). These managers have basically the same functionality as the simple manager shown above. They also keep the VMT pointers of the registered forms, so the formmanager can create a form based on its class name. Additionaly, a menu location is registered.

At startup, our application loads all known modules one by one, and unloads them again. As a result, all forms are registered in the application, and a menu can be built. In the server application, a list of all available providers is built in the same manner.

There are still three things to be noted:

The first is that it is imperative that a module is not unloaded as long as an object (form, datamodule etc.) which was obtained from this module, is in memory. If a module is unloaded and a form method is called of a form that has its code in this module, then an access violation will occur since the code is no longer in memory.

Secondly, It is absolutely necessary that the main application and all other modules are built using packages, and that the form manager resides in a package which is used by all. If not, the application will have a copy of the form manager, and each module as well. The forms in the module will be registered with the copy of the form manager in the module, which is obviously not what is needed.

Third and lastly, the approach taken here leaves the task of registering available functionality to the module: At loading time; the module registers all available forms. A different approach can be taken: The module could register a series of callbacks, which are then used by the calling application to 'question' the module about the available functionality. Which approach should be taken is probably largely a matter of taste. The approach taken here is the same as is used in Delphi itself when creating and registering components in the IDE...

5 Conclusion

In this article a beginning was made with the development of large database applications. It was argued how the object-orientedness of Delphi can be put to use to reduce the application size and increase loading speeds. Secondly, it was shown how to build a modular program which loads its modules dynamically at runtime (a technique, similar to plug-in functionality). In the next article, a closer look will be taken at the form manager, and how it can be put to use to give the end-user a powerful navigation tool.