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 Introduction

In this article, a different way of handling forms and navigation between forms is presented. The article will mainly be concerned with the question 'How to pop up a new form with information related to the currently visible form?', mainly in the setting of large database applications where lots of related information exists.

In a database application, there are lots of occasions where one must put a button on a form which will open a screen with detailed information:

- When presenting an overview of customers (e.g. in a grid), a button may appear on the form which opens a screen that allows to edit the addresses associated with this client, or some screen to manage telecom information. As the user scrolls in the overview of clients, the detail information should be refreshed to reflect the information of the currently selected client, as is customary in master-detail relations.
- In a school administration application, there may be an overview of the courses; Next to the overview, there could be a button which opens a screen which gives an overview of the pupils in the school which are currently following this course.
- A list of pupils can have many such buttons: A button to show the personal data of the pupil (date of birth, gender, name of parents etc), another button to show addresses of the pupil, a list of courses the pupil is currently following, a list of un-payed bills the pupil may have, etc.

All these situations are cases where a master-detail relationship exists, at the level of datasets, and hence at the level of the forms as well.

What is more, often the same details can be accessed from different forms: e.g. details about which courses a student follows can be accessed from the overview of pupils currently inscribed in the school, but also from the overview of pupils in a group. These two overviews are implemented in different screens, but the details are the same. The 'master view' which will provide the ID of a Pupil whose course should be shown, is simply different.

This kind of situations is common in any database program where a lot of interrelated tables are present. In one of our database programs, there are currently 297 forms, with 181 master-detail links defined between forms. Usually, creation of a form to show details of data already on screen is implemented in a `OnClick` handler of some button:

```
TMasterForm.DetailsButtonOnClick(Sender : TObject);

Var
  F : TDetailForm;
  cs : TCursor;

begin
  cs:=Screen.Cursor;
  Screen.Cursor:=crHourGlass;
  Try
    // Create new instance of form.
    F:=TDetailForm.Create(Self);
    // Set up master-detail relationship.
    F.DetailDataset.DataSource:=Self.MasterDataSource;
    // Show the form.
    F.Show;
  Finally
    Screen.Cursor:=cs;
  end;
end;
```

Additional actions can of course be taken, this depends on the case. The fact that this code is more or less the same for all cases where such master-detail relations should be made, is sufficient reason to separate out this code and put it into some central routines.

In general, it is so that the data shown on the 'detail' form will be dynamically updated when the user scrolls in the 'master' form. However, this does not always have to be so, and it can be useful sometimes not to update the details when browsing in the master data. One can take this a step further: The user should be able to decide whether or not the form with details should indeed keep track of changes in the position of the dataset on the 'master' form. This may not always be desirable:

- In an overview with available courses in a school, people may want to open 2 overviews of pupils following a course, for two different courses, to get a visual idea of which pupils are following both courses.
- In an overview of pupils, one may wish to see the address data of a brother and sister at the same time, in order to compare the addresses.

In the above cases, changing the current record in the the 'master' form, should not have any effect on the data shown in the 'Detail form', as it would never be possible to show the information of different pupils at the same time.

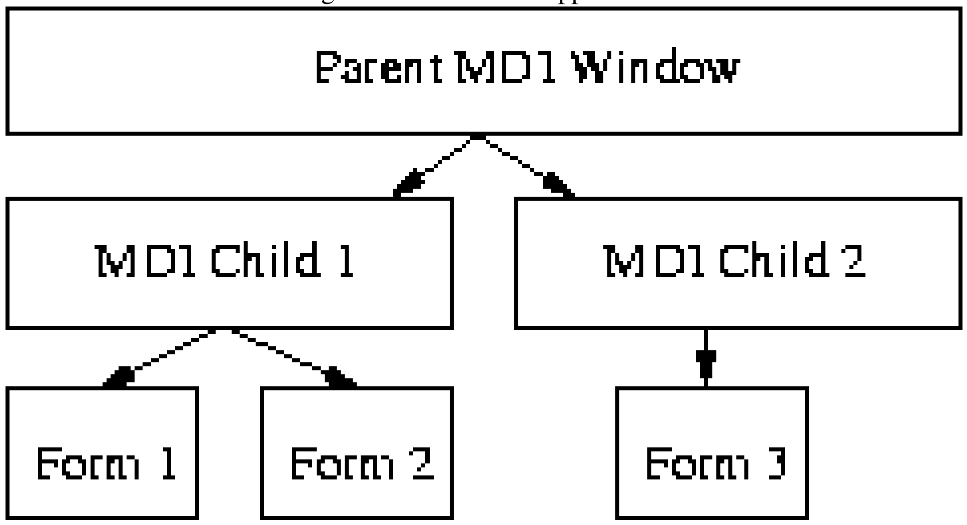
Another aspect of master/detail linked data is how the detail form should be presented on the screen: The usual approach is to show the detail form as a separate form. This form can be shown nonmodal (useful when the details should follow the master), but can also be shown modal, in which case the form showing the details must be closed first.

There is yet another approach possible: The details can be shown on the same form as the master form, but on a new tabsheet of a pagecontrol. By changing the tabs in the pagecontrol, the user can then switch from master to detail information. Here again, the choice can be left to the user: Either he opens a new form with the requested information, or the information is shown on a new tab on the same form.

Why would one want to do this ? In principle, the showing of details can continue *ad infinitum*: In the detail screen, details of the details can be asked:

- In the list of customers, one can ask the list of addresses associated with the customer. In the list of addresses, one can ask a list of telecom data for this address, etc.
- In a bookkeeping program, one can have an overview of journals. The first details screen can offer an overview of the time period covered by the journal, divided in various years. The second detail can show the months of the selected year. The third detail shows all documents in the selected month. The fourth detail shows the document header data, and the last detail shows the bookkeeping entries associated with a document.

Figuur 1: Forms in the application



In practice, this 'showing of more details' stops after at most 3 or 4 levels, but in principle it could be much more, this depends of course on the database model.

When having 3 or 4 levels of details on the screen, one can get lost between the various windows; If all details are presented simply on different tabs on the same window, the screen isn't cluttered with windows, and the user can easily switch between the various levels of details.

But how to show the same information either on a new form or on the same form without implementing the same thing twice ? Two possible approaches jump to mind:

1. Design all screens in the application as frames, and programmatically put each frame on a MDI Window at runtime. Either on a new MDI window, or on a new tabsheet on an existing MDI window.
2. Design all screens in the application as forms, and dock these forms on a MDI Window or on a new tabsheet on an existing MDI window.

In the below, the second approach was taken: it has the additional advantage that each form can be used as a stand-alone form; The former method always requires a 'host form' on which the frame should be placed. Additionally, in Delphi 5 there are some scaling issues with frames, using forms avoids these issues.

Schematically, this can be represented as in figuur 1 op pagina 3. The main application window is a MDI application which manages a series of MDI child windows. The forms as developed by the programmers (Form 1, Form 2 and Form 3) are docked on these MDI child windows. Each MDI child window can have one or more forms docked on it. The same approach can be used in an SDI application. In the below, the term *MDI Window* may be freely interchanged with the term *SDI Window*.

All this functionality can be implemented in 2 different objects:

A Formmanager is an object which handles the creation of a new form. It also handles the registration of forms as discussed in the previous article, and keeps the list of available forms, together with their menu entry points etc.

A Dockmanager is an object which takes a form and places (docks) it on a MDI Window. It also establishes (if needed) a link between a master and a detail form.

Both of these objects will be discussed below.

2 Implementing a form manager

The form manager is essentially no more than a specialised list object. Its function is quite simple:

1. Keep a list of available forms, and, if a menu item should exist which opens the form, where this menu item should appear in the menu.
2. Create an instance form based on the forms class name. When using a modular application as discussed in the previous article, the form manager should use the module manager which loads and unloads modules as needed.

To be able to do this, at least the following information about the available forms should be present:

Class name The (unique) class name of the form.

Class reference Essentially, the VMT (Virtual Method Table) pointer of the form.

Menu entry Where the form should appear in the menu of the application. This can be empty, in which case no menu entry will be created.

Module name When making a modular application, in what module the form is implemented.

The classname can of course be determined from the class reference, but if the module which contains the form is not loaded, the class reference is invalid, hence the class name must be stored as well.

The formmanager class should at least have the following methods and properties:

```
TAppFormClass = Class of TAppForm;

procedure RegisterForm (AFormClass : TAppFormClass; AModule,Menu : string);
function CreateForm(ID: Integer;AOwner : TComponent): TAppForm;
function FindFormByName (FormName : String) : Longint;
Function FindFormByClass(FormClass : TAppFormClass) : Longint;
Property FormCount : Longint Read GetFormCount;
Property FormNames [Index : longint] : String Read GetFormName;
Property FormModule [Index : longint] : TModule Read GetFormModule;
Property FormClass[Index : Longint] : TFormClass Read GetFormClass;
Property Menu : PAppMenurec Read GetMenu;
```

The `FormNames`, `FormCount` and `FormClass` methods simply return the stored information about the forms. The `FindFormByName` and `FindFormByClass` functions are used to search for the index of a form in the list of forms, based on the class name or class pointer.

The `TAppForm` is a descendent of `TCustomForm` which is used when designing a form; All forms in the application are of this class.

The interesting calls are the `RegisterForm` and `Createform` calls, and the `menu` property is also worth a closer look.

The `RegisterForm` call registers a new form; Typically this will be called by the initialisation routines of modules as they are loaded. Its arguments are straightforward:

AFormClass The form class reference. From this, the class name can be retrieved, and stored as well.

Module The module in which the form resides. This is needed to be able to notify the module manager that a module should be loaded, or that the reference count of a module should be increased.

Menu The menu entry (if any) where the form should appear. This is a string, which contains the menu entry in encoded form:

```
CustomersMenuEntry='&Customers|&Overview';
```

This would cause a main menu item 'Customers' to be created, and under that the menu item 'Overview' would be created. Clicking this menu item could create a form which shows an overview of all known customers.

It is obvious that each form which should be created by the formmanager must be registered first.

Note that the register call should overwrite any existing entry: When the application initially loads a module (so that all forms contained in the module are registered), it unloads the module at once (there should be *no* unregistering of the forms).

When the form manager then loads a module again because a form must be created, then all forms are registered again (because the initialisation code of the module is executed again). Since the module may be loaded in a different location in memory, the VMT pointers for the various forms may be different, so the values stored by the form manager must be updated.

This is of course not necessary if the modules are not unloaded, or if no modules are used.

The `RegisterForm` call also builds a menu as it goes. The `Menu` property at all times reflects the menu structure as seen by the formmanager. It consists of a linked chain of `TMenuRec` records:

```
PAppMenuRec = ^TAppMenuRec;
TAppMenuRec = Record
  Caption : String;
  FormID : Longint;
  Next,
  Sub : PMenuRec;
end;
```

The `FormID` is the index (in the formmanager's list of known forms) of the form that should be created when the menu item is clicked on. The ID rather than the class name is used, for performance reasons. The `RegisterForm` call should analyse its `Menu` parameter in order to determine where in the menu tree the form should be inserted.

From this linked list, the real menu structure can then be created at runtime with a simple recursive loop. The reason that this happens separately is because then the menu can be created *after* the user has logged in to the application, while the registration of the forms happens before the user is logged in, at program startup. Separating these two moments allows some customisation of the menu (e.g. enable/disable items) based on security profiles and possible preferences of the user.

To be able to determine the form which must be created when a particular menu item is clicked, the index of the form is stored in the `Tag` property of the menu item when it is created. In the `OnClick` handler, the `Tag` property can then be used to determine which form must be created.

The formmanager can now be used to create a new form when the user clicks some menu, or when details of some data are requested. In general, it can be used whenever some form must be created. It can also be used to centralise user messages: While creating a new form, some kind of status bar message can be displayed, or a progressbar can be displayed. By centralising the creation of forms, the adding of such effects can be implemented in a general way, without having to rewrite the same code over and over again.

3 The DockManager

The dockmanager object is responsible for 3 things:

1. Creating a form using the form manager. Forms are never created by directly accessing the form manager, but always by using the dockmanager.
2. Docking a newly created form on an MDI window. This can either be an existing or a new MDI Window, depending on what the user wanted.
3. Establishing a master-detail relationship between the data on an existing form and the data on the newly created form.

The Dockmanager has the following overloaded methods:

```
Function NewAppForm(AppParent : TCustomForm;
                    ID : Longint;
                    MasterDS : TDataSource) : TAppForm;
Function NewAppForm(AppParent : TCustomForm;
                    FormClass : TAppFormClass;
                    MasterDS : TDataSource) : TAppForm;
Function NewAppForm(AppParent : TCustomForm;
                    FormName : String;
                    MasterDS : TDataSource) : TAppForm;

Function AddAppForm(DockForm : TAppDockForm;
                    ID : longint;
                    MasterDS : TDatasource) : TAppForm;
Function AddAppForm(DockForm : TAppDockForm;
                    FormClass : TAppFormClass;
                    MasterDS : TDatasource) : TAppForm;
Function AddAppForm(DockForm : TAppDockForm;
                    FormName : String;
                    MasterDS : TDatasource) : TAppForm;
```

The various `NewAppForm` calls do nothing but create an MDI window (of class `TAppDockForm`) and then call the corresponding `AddAppForm` with as `DockForm` argument the newly created dockform. The various overloaded forms of the call are mainly for convenience reasons, a new form can be created and docked based on the class, the classname, or the ID of the form.

Docking the form on a MDI window is quite easy; the `TAppDockForm` is a descendent of `TForm` which has its `FormStyle` property set to `fsMDIChild`. On this form, a `TPageControl` is located. When docking a new `TAppForm` instance on the `TappDockForm`, essentially the following code is executed:

```
function TAppDockForm.DockAppForm(AMaster, Form: TWisaForm) : TControl;
```

```
begin
  result := TAppTabsheet.Create (Self);
  with TAppTabSheet(result) do
    begin
    Pagecontrol := WindowPager;
    TabForm := Form
    end;
  Form.ManualDock(TAppTabSheet(result),Nil,alnone);
```

The `TAppTabSheet` descendent of `TTabSheet` is created specially to be able to introduce the `TabForm` property, which contains a link to the form that is docked on the tabsheet. The `WindowPager` is the `TPageControl` on which all forms are docked.

If the `TAppTabSheet` was not used, then the following code could be used:

```
  WindowPager.DockControl(Form);
```

A `TPageControl` automatically creates a new tab if a control is docked onto it.

After a control is docked, some resizing is done (all forms are designed to be resizable) and the tabsheet where the new form is located is made the active tabsheet, so the new form becomes visible.

Establishing a master-detail relationship is done using the `MasterDS` argument. If it is not `Nil`, then the new `TAppForm` is examined; A property `MainDataSource` (of type `TDataSource`) is used, and this one is used to establish a master-detail relation with code similar to the following:

```
If (MasterDS<>Nil) and
    (NewForm.MainDataSource<>Nil) then
  NewForm.MainDataSource.DataSet.DataSource:=MasterDS;
```

The programmer sets the `MainDataSource` property at design time to one of the datasources in the form. If the associated dataset needs any parameters, it will obtain them from `MasterDS`, as it happens with e.g. the `TQuery` object or the `TIBQuery` objects in Delphi. Only one type of dataset is used in the application, one which has the `DataSource` property, so it is guaranteed that the above assignment is valid.

If the user now scrolls in the `MasterDS`'s dataset, the parameters in the dataset on the detail form will be updated.

In the case the user does not want the detail form to be synchronised with the master form, then any eventual parameters in its `MainDataSource` are simply filled up with any field values found in the `MasterDS`

```
Var
  I : Integer;
  N : String;
  F : TField;

begin
If (MasterDS<>Nil) and
    (NewForm.MainDataSource<>Nil) then
  With NewForm.MainDataSource.DataSet do
    For I:=0 to Params.Count-1 do
     If Not Params[i].Bound Then
       begin
```

```
        N:=Params[i].Name;
        F:=FindField(N);
        If F<>Nil then
          Params[i].AssignField(f);
        end;
```

If the user now scrolls in the `MasterDS`'s dataset, the detail form will not change, as the parameter values are filled in once, and not dynamically updated from the master datasource.

When designing queries and forms, one must take this into consideration: When designing a detail form, the query must have a parameter with a name of a field which can be found in the master datasource for this query. Sometimes, it may be necessary to add this field to the master dataset, just for the benefit of showing details.

In the design of some forms, it should be anticipated that the form may be created to show details of some other form, but that the form also appears as en entry in the menu: The list of addresses of a customer is a form which was designed like this: It can be called when the overview of the customers is on screen, but the 'list of addresses' may be chosen from a menu entry to provide faster access.

Since the form is designed to function as a detail screen, the query on this form expects a parameter which holds the customers ID.

However, if the list of addresses is chosen from a menu entry, then there is no dataset available to provide this ID. So, an overview of customers must be created and placed 'before' the form with addresses. The 'main' dataset on this overview will then be used to provide the parameters.

To be able to know which overview should be created when a form is opened, a `DefaultBrowserForm` property is introduced in `TAppForm`. This string property contains the class name of a form which must be created when the (detail) form needs a (master) form with a dataset which can function as a master datasource for the main datasource. We call this form the 'master browser form'.

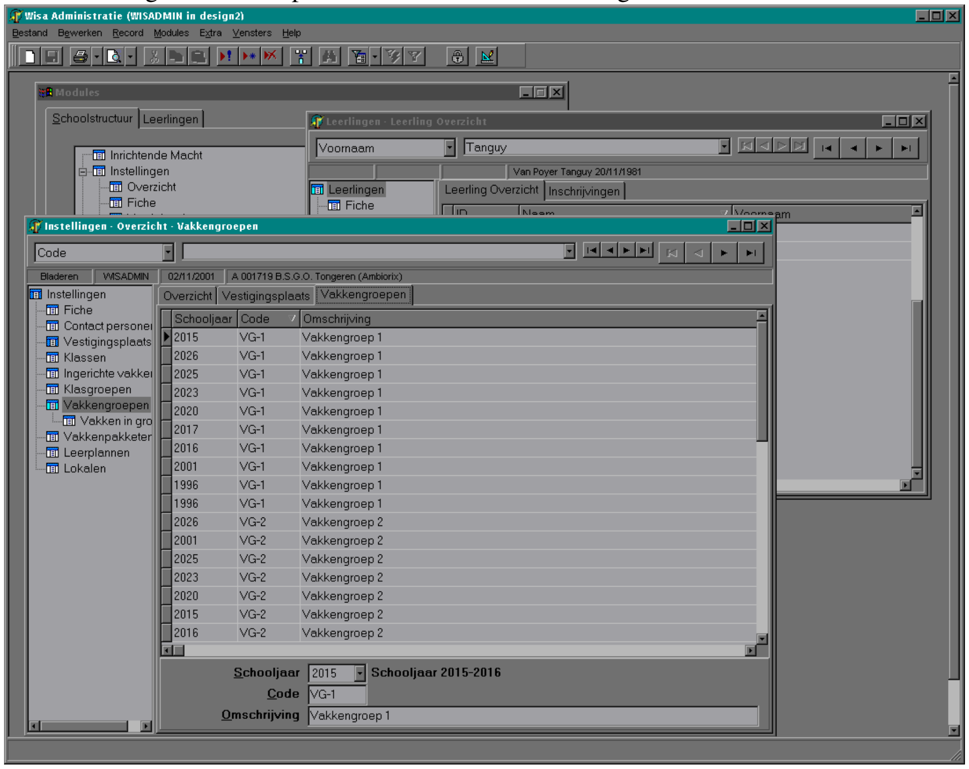
It is the dockmanager which checks whether a 'master browser form' should be created:

```
With NewForm do
  If (DefaultBrowserForm <> '') and
     (Not Assigned(MasterDS)) then
    begin
    // recursively add form.
    AppBrowser :=AddAppForm(DockForm,DefaultBrowserForm,Nil,nil);
    MasterDS:= AppBrowser.MainDataSource;
    AppBrowser.Show;
    end
```

From this code it is obvious that this process can be repeated recursively; This is not so uncommon as one may think. For instance: A form to edit a student's personal data (call it `TStudentForm`) requires a list of students (to obtain the ID of the student whose data needs to be edited). The list of students is the list of students in a certain group (call it `TGroupMembersForm`). So an overview of available groups (classes) is needed (`TGroupOverView`). But the overview of available groups needs a school ID (multiple schools can be managed in the application), so an overview of schools is needed (TSchoolOverviewForm).

So when the user clicks on the menu entry 'Student personal data', the following chain of events occurs:

Figuur 2: Example of an MDI window showing some docked forms



Schooljaar	Code	Omschrijving
2015	VG-1	Vakkengroep 1
2026	VG-1	Vakkengroep 1
2025	VG-1	Vakkengroep 1
2023	VG-1	Vakkengroep 1
2020	VG-1	Vakkengroep 1
2017	VG-1	Vakkengroep 1
2016	VG-1	Vakkengroep 1
2001	VG-1	Vakkengroep 1
1996	VG-1	Vakkengroep 1
1996	VG-1	Vakkengroep 1
2026	VG-2	Vakkengroep 2
2001	VG-2	Vakkengroep 2
2025	VG-2	Vakkengroep 2
2023	VG-2	Vakkengroep 2
2020	VG-2	Vakkengroep 2
2015	VG-2	Vakkengroep 2
2016	VG-2	Vakkengroep 2

Schooljaar 2015 Schooljaar 2015-2016
Code VG-1
Omschrijving Vakkengroep 1

1. The `TStudent` is created. Its `DefaultBrowserForm` property equals `TGroupMembers`.

2. A `TGroupMembers` form is created, and a link is established with the `TStudent` form. Its `DefaultBrowserForm` is `TGroupOverView`

3. The `TGroupOverView` form is created, and a link is established with the `TGroupMembers`. The `DefaultBrowserForm` property of the `TGroupOverView` equals `TSchoolOverview`

4. The `TSchoolOverview` form is created and a link is established with the `TGroupOverView`.

5. The `TSchoolOverview` form is docked and shown.

6. The `TGroupOverView` form is docked and shown.

7. The `TGroupMembers` form is docked and shown.

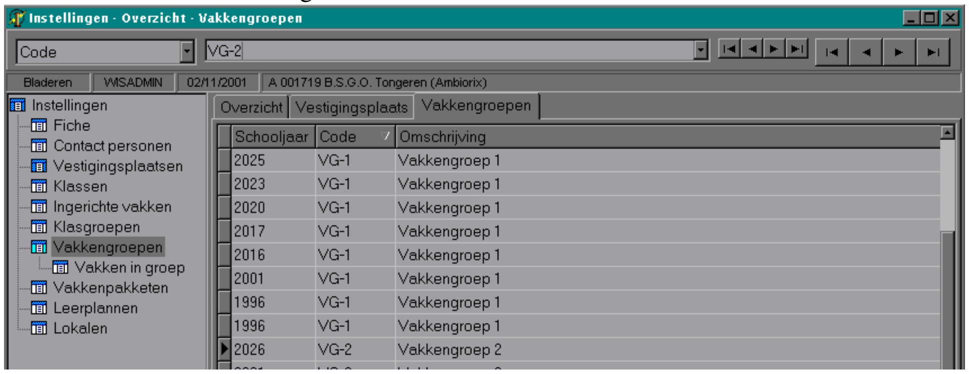
8. Finally, the `TStudent` form is docked and shown.

An example of an application with some MDI forms that have such a series of forms docked on them is shown in figuur 2 op pagina 9.

The MDI window on which all these forms are docked contains some extra functionality. This is treated in the next section.

4 The DockForm

The MDI form on which all other forms are docked have some extra functionality to manage the forms docked on it:



Figuur 3: A detail view of a dockform.

Instellingen - Overzicht - Vakkengroepen

Code | VG-2

Bladeren | WISADMIN | 02/11/2001 | A 001719 B.S.G.O. Tongeren (Ambiorix)

- Instellingen
  - Fiche
  - Contact personen
  - Vestigingsplaatsen
  - Klassen
  - Ingerichte vakken
  - Klasgroepen
  - Vakkengroepen
    - Vakken in groep
  - Vakkenpakketen
  - Leerplannen
  - Lokalen

Overzicht | Vestigingsplaats | Vakkengroepen

Schooljaar	Code	Omschrijving
2025	VG-1	Vakkengroep 1
2023	VG-1	Vakkengroep 1
2020	VG-1	Vakkengroep 1
2017	VG-1	Vakkengroep 1
2016	VG-1	Vakkengroep 1
2001	VG-1	Vakkengroep 1
1996	VG-1	Vakkengroep 1
1996	VG-1	Vakkengroep 1
2026	VG-2	Vakkengroep 2

1. A pagecontrol, which allows to select one of the currently docked forms.
2. A Treeview, which shows the currently docked forms in a hierarchical manner; it also shows for each form which related forms can be opened.
3. Further there is functionality to close any docked form by means of a pop-up menu. Forms that depend on the chosen form by means of a master/detail relationship, are closed first.

If data-aware forms are docked on the dockform, the following additional functionality is present:

1. A navigator which can be used to browse through the dataset on the currently visible form.
2. A navigator which can be used to browse through the master dataset of the currently visible form. For example, if the currently visible form contains the list of addresses of a client, and the master datasource is showing the list of clients, then this navigator will browse through the list of clients.
3. A statusbar which shows some useful information: The state of the currently active dataset: Browse, Edit, Insert etc. A short description of the current record in the master browser dataset, and some 'audit' information: Who changed the record last, and when. The display of this information is controlled by some new properties of the datasets used in the application.
4. A lookup-edit control: When something is typed in this control, it is used as an argument to a `Locate` on the currently active dataset. This can be used to perform an incremental search. The field on which the lookup is performed is selectable with a combobox which shows the available fields.

By implementing all this functionality in the dockform, the functionality is present in all forms that are docked on this form, and does not need to be implemented in each form separately. A detail of a dockform with three forms docked on it in a master-detail relationship, is shown in figuur 3 op pagina 10. Note that the statusbar is placed at the top of the window. This is done for clarity: All status information and navigation controls are concentrated at the top of the window.

Besides the above information, there is also a TreeView which shows a graphical representation of the forms docked on the MDI form: Each node in the TreeView represents a docked form. The hierarchy is determined by the master-detail relations of the docked forms: a form with details of a master forms appears as a child node of the node that represents the master form. Clicking on a node will raise the form, as if the user had clicked on the tab of the pagecontrol on which the form is docked. The currently active form is shown with an alternate colour. This TreeView has been called the 'navigation TreeView'.

What is more, the TreeView also contains nodes that represent possible detail forms: under each node the possible detail forms are shown as child nodes. When such a child node is clicked, the corresponding detail form is opened and hooked up to the form.

The nodes that represent possible detail forms differ from the nodes that represent already opened forms by their image colour. Nodes that the user cannot open can appear grayed: e.g. When there are no clients in the overview of clients, the detail form with the addresses of clients is not accessible. In the case the user has not sufficient rights to open the form, the node could be removed altogether from the TreeView.

The MDI docking window contains the logic to keep all these controls synchronised; Whenever the user changes the currently visible form, the following actions are performed:

1. The `DataSource` property of both navigators is re-determined from the newly active form.
2. The `DataSource` used to display information in the statusbar is determined.
3. The `DataSource` used to do a lookup is set.
4. The active control on the currently visible form is restored. Docking a form on another form has as a consequence that the 'ActiveControl' property is lost. Only the MDI Window's ActiveControl is meaningful. So some logic is needed to preserve and restore the Active control of the currently active form.
5. In the TreeView, the Node for the active form gets highlighted.

Special events are introduced in the forms to prevent the user from switching to another form. These events are then fired when the user attempts to switch the current form: In the 'OnChanging' event of the Pagecontrol on which all forms are docked, one can check the result of these events to decide whether the user is allowed to switched to another form.

In our programs we have opted no to allow the user to switch from one form to another if the active form contains unsaved data; This avoids possible problems in the Master-Detail relations, e.g. A user entering Detail data when the Master data have not yet been posted. One of the reasons for this is that, by means of the navigation treeview, the user can open not only detail forms of the current form, but of all forms that have been docked on the MDI window, and sometimes this can cause strange effects if the data in the current form is not saved.

When closing the MDI form, all docked forms are closed in an orderly manner; The list of docked forms is closed starting with the last opened form, working back to the fist opened form. Any form first closes any detail forms that may be attached to it, and only when all child windows are closed, then it closes itself. This algorithm ensures that all data is saved correctly and no conflicts arise.

There is a tricky point that should be taken in consideration when implementing such a loop. When closing a form using the close method as follows:

```
  DetailForm.Close;
```

What happens behind the scenes is that a `WM_CLOSE` message is sent to the form. In order to be sure that the form is closed and destroyed, the application must allow Windows to send the message, and the application should have processed the message before the next detail form is destroyed or before the master form is destroyed.

So the loop to close the detail forms looks something like this:

```
For I:=0 to FormLinks.Count-1 do
  Begin
  FormLinks[I].FormInstance.Close;
  ProcessMessages;
  If Not (FormLinks[I].FormInstance=Nil) then
    Break;
  end;
```

The `FormLinks` property is a collection which represent the possible detail forms of a form (it will be discussed later). The `FormInstance` property of each of the items in the collection contains a pointer to the instance of the detail form that was created.

The `ProcessMessages` call allows Windows to send the `WM_CLOSE` message (sent by the `Close` method), and lets the application process it. The detail form will close (if it is allowed to close) and free itself. The destructor of the form will notify the master form that the detail form is gone; It does this by emptying the `FormLinks[I].FormInstance` pointer. This allows the master form to check whether the detail form was actually closed.

There are cases when a detail form cannot be closed (The `OnCloseQuery` handler returned `CanClose:=False`) in that case the parent form also cannot be closed, and the process of closing forms is aborted.

In what follows, 2 ways of opening a new screen are discussed.

5 A FormLinks property

In the previous section, it was shown that the MDI dock window has a TreeView which displays a hierarchical view of all opened forms, but that it also shows for each form the possible detail forms for that form.

This information is obtained from a new property of the form: `FormLinks`. Instead of dropping a number of buttons on the form (one for each type of detail which can be opened), the programmer edits the `FormLinks` property. The `FormLinks` property is a descendent of `TCollection` called `TFormLinks`, and the items in the collection are of type `TFormLink`. Using a `TCollection` has the benefit that the standard Delphi collection editor can be used to edit this new property.

Each `TFormLink` item in the collection has the following published properties:

DisplayName The name of the detail form as shown in the Navigation TreeView.

DetailForm The class name of the form to be created when the user clicks on the node for this form. This name will be given to the dock manager and form manager to create the detail form.

MasterDS Is the `Datasource` on this form which will be used to establish the master-detail relationship.

FollowMode This is an enumerated type:

```
TFollowMode = (fmSynchronized,fmSnapshot)
```

The `fmSynchronized` means that a real master-detail relationship will be established between `MasterDS` and the main datasource of the detail form (This is the default). `fmSnapShot` means that parameters which appear in the detail form's main datasource and for which a corresponding field exists in the `MasterDS` datasource, will be copied just once.

OpenMode This is an enumerated type which determines how the detail form will be opened:

```
TOpenMode = (omModal,omTabbed,omNonModal);
```

The `omTabbed` value means that the detail form will be opened on a new tab of the MDI Window on which the current form is docked (this is the default). The `omNonModal` value means that the detail form will be opened in a new MDI window. The `omModal` value is used to show the detail form on a modal window.

OverrideFollowMode A boolean property that specifies whether the user can change the `FollowMode` property in a pop-up menu, at run-time.

OverrideOpenMode A boolean property that specifies whether the user can change the `OpenMode` property in a pop-up menu, at run-time.

ReadOnly A boolean property which indicates whether the detail form should be opened read-only: In that case the user is not allowed to edit data in the detail form.

NeedRecords A boolean which indicates whether the `MasterDS` dataset should contain records; If it does not, the node for this detail form will be disabled.

Visible Indicates whether the node (in the navigation TreeView) for this detail form should be visible or not.

Some combinations of the `OpenMode` or `FollowMode` are of course not useful; it makes little sense to open a detail form synchronized and modal - if it is modal, the user has no opportunity to scroll in the master dataset.

The following events are also defined:

BeforeCreateForm an event of type `TBeforeCreateFormEvent`:

```
TBeforeCreateFormEvent = Procedure (Var FormName : String) Of Object;
```

It gets passed the classname of the detail form that will be created; This name can be changed to something else or made empty. The latter can be used to signal that a form should not be created at this time. The former can be used to show a different detail form based on e.g. what record is currently selected in the form. For example, in a list of journals, the form showing the details of a journal depends on the type of the journal. when the details form is opened, the type of the journal is first checked to see which form should be opened.

AfterCreateForm This event of type `TAfterCreateFormEvent` occurs after the form was created:

```
TAfterCreateFormEvent = Procedure (TheForm : TWisaForm) Of Object;
```

This event can be used to set some additional properties on the newly created form (A pointer to the newly created instance is passed in `TheForm`).

Besides all these published properties which can be set by the programmer at design time, there is also a run-time only property `FormInstance`. If the user opened the detail form in `fmSynchronize` mode then this will contain a pointer to the detail form.

The `TFormLink` class also has an `Execute` method, which creates the detail form according to the properties set for this detail form. After creating the form it will establish the master-detail link. This is the method that is called when the user clicks a node in the Navigation TreeView.

To be able to fill in the `DetailForm`, a special property editor was implemented which presents a list of available forms. Not all forms are presented, only the forms in the current project; Delphi currently provides no means to retrieve a list of all forms of all projects in the project group. This could be remedied by keeping a list of forms somewhere in a file, and reading from that list. The main reason for wanting to present a list is to avoid typing mistakes; it is still possible to type a class name of a form that does not (yet) exist.

6 Developing a ScreenButton

Besides the `FormLinks`, a `TDetailButton` can also be implemented; it has the same properties and events as the `TFormLink`; When it is clicked, it executes the same code as the `TFormLink`'s `Execute` method. It keeps - just as the formlink - a pointer to the opened form if the used decides to open a form.

A form must be made aware of the `TScreenButton`: When the form is closed (or CloseQuery is called), the form should check all screenbuttons on it, and check whether these buttons have a form opened. If so, this form should be closed first. Failing to do so, will result in master-detail links being broken.

7 Conslusion

In this article, some ideas on managing forms and navigating between forms in a uniform way were presented. In doing so, some extra properties of a specialised `TForm` descendent have been encountered. In a next article, more ways to extend Delphi's `TForm` component will be presented, and several ways to integrate this in Delphi's IDE will be discussed.