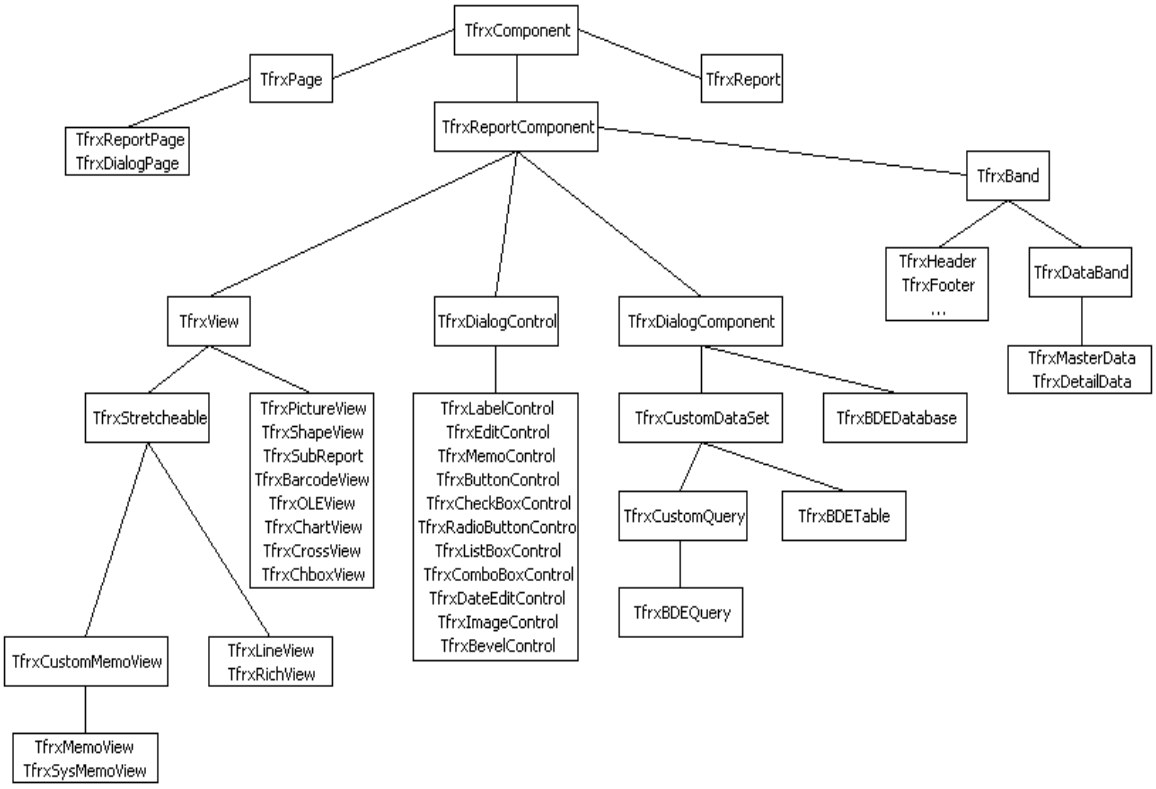
FastReport 3.0

Developer's manual

Edition 1.01

Table of Contents

Hierarchy of FastReport classes



TfrxComponent is the basic class for all FastReport components. Objects of this type have such parameters as "coordinates," "size," "font," "visibility," and lists of subordinate objects. The class also contains methods, which allow to save/restore object's state to/from the stream.

```
  TfrxComponent = class(TComponent)
  protected
    procedure SetParent(AParent: TfrxComponent); virtual;
    procedure SetLeft(Value: Extended); virtual;
    procedure SetTop(Value: Extended); virtual;
    procedure SetWidth(Value: Extended); virtual;
    procedure SetHeight(Value: Extended); virtual;
    procedure SetFont(Value: TFont); virtual;
    procedure SetParentFont(Value: Boolean); virtual;
    procedure SetVisible(Value: Boolean); virtual;
    procedure FontChanged(Sender: TObject); virtual;
  public
    constructor Create(AOwner: TComponent); override;
    procedure Assign(Source: TPersistent); override;
    procedure Clear; virtual;
    procedure CreateUniqueName;
    procedure LoadFromStream(Stream: TStream); virtual;
    procedure SaveToStream(Stream: TStream; SaveChildren: Boolean =
True); virtual;
    procedure SetBounds(ALeft, ATop, AWidth, AHeight: Extended);
    function FindObject(const AName: String): TfrxComponent;
```

```
    class function GetDescription: String; virtual;

    property Objects: TList readonly;
    property AllObjects: TList readonly;
    property Parent: TfrxComponent;
    property Page: TfrxPage readonly;
    property Report: TfrxReport readonly;
    property IsDesigning: Boolean;
    property IsLoading: Boolean;
    property IsPrinting: Boolean;
    property BaseName: String;

    property Left: Extended;
    property Top: Extended;
    property Width: Extended;
    property Height: Extended;
    property AbsLeft: Extended readonly;
    property AbsTop: Extended readonly;

    property Font: TFont;
    property ParentFont: Boolean;
    property Restrictions: TfrxRestrictions;
    property Visible: Boolean;
  end;
```

- Clear – clears object's contents and deletes all its child objects.
- CreateUniqueName – creates a unique name for an object placed into the report.
- LoadFromStream – loads object contents and all its child objects from the stream.
- SaveToStream – saves an object to the stream. The SaveChildren parameter defines, whether state of all child objects should also be saved.
- SetBounds – sets coordinates and size of an object.
- FindObject – searches for an object with specified name among the child objects.
- GetDescription – returns to the object's description.

The following methods are called when modifying the corresponding properties. If additional handling is needed, you can override them:

```
SetParent
SetLeft
SetTop
SetWidth
SetHeight
SetFont
SetParentFont
SetVisible
FontChanged
```

The following properties are defined in the "TfrxComponent" class:

- Objects – the list of child objects;
- AllObjects – the list of all subordinate objects;
- Parent – link to the parent object;

- Page – link to the report page, which the object belongs to;
- Report – link to the report, which the object belongs to;
- IsDesigning – "True," if the designer is running;
- IsLoading – "True," if an object is being loaded from the stream;
- IsPrinting - true, if an object is being printed out;
- BaseName - basic name of an object. This value is used in the "CreateUniqueName" method;
- Left – object's X coordinate (relatively to a parent);
- Top - object's Y coordinate (relatively to a parent);
- Width – object's width;
- Height – object's height;
- AbsLeft - object's X absolute coordinate;
- AbsTop - object's Y absolute coordinate;
- Font – object's font;
- ParentFont – if "True," then uses the parent object font settings;
- Restrictions – set of flags, which restrict some object operations;
- Visible – object's visibility.

The next basic class is TfrxReportComponent. Objects of this type can be placed into a report. The class contains the "Draw" method for object's painting, as well as "BeforePrint/GetData/AfterPrint" methods, which are called as soon as a report runs.

```
  TfrxReportComponent = class(TfrxComponent)
  public
    procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY:
Extended); virtual; abstract;
    procedure BeforePrint; virtual;
    procedure GetData; virtual;
    procedure AfterPrint; virtual;
    function GetComponentText: String; virtual;
    property OnAfterPrint: TfrxNotifyEvent;
    property OnBeforePrint: TfrxNotifyEvent;
  end;
```

The Draw method is called when painting an object. The parameters are:
- "Canvas" – canvas;
- "Scale" – zoom by X-axis and Y-axis;
- Offset – offset relatively to the edges of the canvas.

The BeforePrint method is called right before the object is handled (during process of building a report). This method saves object's state.

The GetData method is called to load data into an object.

The AfterPrint is called after the object is handled. The method restores the object's state.

The TfrxDialogComponent class is the basic one for writing non-visual

components, which can be placed to a dialogue form in a report.

```
TfrxDialogComponent = class(TfrxReportComponent)
public
  property Bitmap: TBitmap;
  property Component: TComponent;
published
  property Left;
  property Top;
end;
```

The "TfrxDialogControl" class is the basic one for writing common control, which can be placed to a dialogue form in a report. The class contains a large number of general properties and events shared with most of the common controls.

```
TfrxDialogControl = class(TfrxReportComponent)
protected
  procedure InitControl(AControl: TControl);
public
  property Caption: String;
  property Color: TColor;
  property Control: TControl;
  property OnClick: TfrxNotifyEvent;
  property OnDblClick: TfrxNotifyEvent;
  property OnEnter: TfrxNotifyEvent;
  property OnExit: TfrxNotifyEvent;
  property OnKeyDown: TfrxKeyEvent;
  property OnKeyPress: TfrxKeyPressEvent;
  property OnKeyUp: TfrxKeyEvent;
  property OnMouseDown: TfrxMouseEvent;
  property OnMouseMove: TfrxMouseMoveEvent;
  property OnMouseUp: TfrxMouseEvent;
published
  property Left;
  property Top;
  property Width;
  property Height;
  property Font;
  property ParentFont;
  property Enabled: Boolean;
  property Visible;
end;
```

When writing your own element, you should inherit from this class, transfer required properties into the "published" section, and then specify new properties for your common control. Refer to the corresponding chapter to know more about writing user common controls.

The TfrxView class is the basic one for most components, which can be placed to a report page. An object of this type has such parameters as "Frame" and "Filling," and also can be connected to the data source. Practically all FastReport standard objects are inherited from this class.

```
  TfrxView = class(TfrxReportComponent)
  protected
    FX, FY, FX1, FY1, FDX, FDY, FFrameWidth: Integer;
    FScaleX, FScaleY: Extended;
    FCanvas: TCanvas;
    procedure BeginDraw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX,
OffsetY: Extended); virtual;
    procedure DrawBackground;
    procedure DrawFrame;
    procedure DrawLine(x, y, x1, y1, w: Integer);
  public
    function IsDataField: Boolean;
    property BrushStyle: TBrushStyle;
    property Color: TColor;
    property DataField: String;
    property DataSet: TfrxDataSet;
    property Frame: TfrxFrame;
  published
    property Align: TfrxAlign;
    property Printable: Boolean;
    property ShiftMode: TfrxShiftMode;
    property TagStr: String;
    property Left;
    property Top;
    property Width;
    property Height;
    property Restrictions;
    property Visible;
    property OnAfterPrint;
    property OnBeforePrint;
  end;
```

The following methods are defined in the class:

- BeginDraw - the method is called from the "Draw" method and calculates integer-valued coordinates and sizes of the object area. The calculated values are presented as "FX," "FY," "FX1," "FY1," "FDX," and "FDY" variables. The frame width (it is placed in the FFrameWidth) is also calculated;
- DrawBackground - draws background for an object;
- DrawFrame - draws the object's frame;
- DrawLine(x, y, x1, y1, w: Integer) - draws a line with specified coordinates and width;
- IsDataField returns "True," if properties of DataSet and DataField contain nonempty values.

One can refer to the following properties after calling the "BeginDraw" method:

- FX, FY, FX1, FY1, FDX, FDY, FFrameWidth are the coordinates, sizes and width of the object's frame, calculated according to the "Scale" and "Offset" parameters;
- "FScaleX" and "FScaleY" are the scales, which are copies of the "ScaleX" and "ScaleY" parameters from the "Draw" method;
- "FCanvas" is a canvas, which is a copy of the "Canvas" parameter from the "Draw" method.

The following properties, being general for most of the report's objects, are defined in the class:

- BrushStyle – object's filling style;
- Color – object's filling color;
- DataField - data field's name, which the object is connected to;
- DataSet - data source;
- Frame - object's frame;
- Align - object's aligning relatively to its parent;
- Printable – defines whether the given object should be printed out;
- ShiftMode is the mode of object's shifting in cases when a stretchable object is placed over the given one;
- TagStr - the field for storing different information.

The TfrxStretcheable class is the basic one for writing components, which modify their height depending on the data placed in it.

```
TfrxStretcheable = class(TfrxView)
public
  function CalcHeight: Extended; virtual;
  function DrawPart: Extended; virtual;
  procedure InitPart; virtual;
published
  property StretchMode: TfrxStretchMode;
end;
```

Objects of the given class can be not only stretched, but they can also be "broken" into pieces in cases when an object does not find room on the page. At the same time, the object is displayed piecemeal until all its data is displayed.

The following methods are defined in the class:

- CalcHeight is to calculate and return the object's height according to the data placed in it;
- InitPart is called before object's breaking;
- DrawPart redraws the next data chunk, which is placed in the object. A "Return value" is a value of the unused space where it was impossible to display the data.

Writing custom report components

FastReport has a great number of components, which can be placed into a report page. They are: text, picture, line, geometrical figure, OLE, Rich, bar code, diagram etc. You can also write your own component, and then attach it to FastReport.

In FastReport, there are several defined classes, from which the components are

inherited. For more details, see the "Hierarchy of classes" chapter. The TfrxView class is of main interest to us, since most report components are inherited from it.

One should realize at least the "Draw" method defined in the TfrxReportComponent basic class.

```
    procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY:
Extended); virtual;
```

This method is called when a component is painted, in the designer, in the preview window, and during printing. The TfrxView overrides this method for drawing object's frame and background. The method should draw the component's contents on the "Canvas" drawing surface. Object's coordinates and sizes are stored in the "AbsLeft, AbsTop," "Width," and "Height" properties respectively.

The "ScaleX" and "ScaleY" parameters define scaling of an object in X-axis and Y-axis respectively. These parameters are equal 1 at 100% zoom and can vary, if a user modifies zooming either in the designer or in the preview window. The OffsetX and OffsetY parameters point shifting of the coordinates by the X-axis and Y-axis. Thus, taking into account these parameters, a coordinate of the upper left corner will be as following:

```
  X := Round(AbsLeft * ScaleX + OffsetX);
```

To simplify operations with coordinates, the "BeginDraw" method (which has parameters similar to the "Draw" method) is defined in the "TfrxView" class

```
  procedure BeginDraw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY:
Extended); virtual;
```

It should be called in the first line of the "Draw" method. This method performs transformation of the coordinates into FX, FY, FX1, FY1, FDX, FDY, FFrameWidth integer values, which can be later used in the TCanvas methods. This method also copies the Canvas, ScaleX, and ScaleY values into the FCanvas, FScaleX, FScaleY variables to which one can refer from any method of the class.

There are also two methods for drawing backgrounds for and frames of objects in the TfrxView class.

```
    procedure DrawBackground;
    procedure DrawFrame;
```

The BeginDraw method should be called before calling these methods.

Let us examine creation process of the component, which displays an arrow.

```
type
  TfrxArrowView = class(TfrxView)
  public
```

```
    { we should override only two methods }
    procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY:
Extended); override;
    class function GetDescription: String; override;
  published
    { carry out the required properties into the published section }
    property BrushStyle;
    property Color;
    property Frame;
  end;

class function TfrxArrowView.GetDescription: String;
begin
  { component's description will be displayed next to its icon in the
toolbar }
  Result := 'Arrow object';
end;

procedure TfrxArrowView.Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX,
OffsetY: Extended);
begin
  { call this method to perform the coordinates' transformation }
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);
  with Canvas do
  begin
    { set colors }
    Brush.Color := Color;
    Brush.Style := BrushStyle;
    Pen.Width := FFrameWidth;
    Pen.Color := Frame.Color;
    { draw an arrow }
    Polygon(
      [Point(FX, FY + FDY div 4),
      Point(FX + FDX * 38 div 60, FY + FDY div 4),
      Point(FX + FDX * 38 div 60, FY),
      Point(FX1, FY + FDY div 2),
      Point(FX + FDX * 38 div 60, FY1),
      Point(FX + FDX * 38 div 60, FY + FDY * 3 div 4),
      Point(FX, FY + FDY * 3 div 4)]);
  end;
end;

{ registration }
var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(hInstance, 'frxArrowView');
  { place our component to the "Other" standard category }
  frxObjects.RegisterObject(TfrxArrowView, Bmp, 'Other');

finalization
  { delete the component from the list of available ones }
  frxObjects.Unregister(TfrxArrowView);
  Bmp.Free;

end.
```

To create a component, which displays any data from DB, one should transfer the DataSet, DataField properties into the "published" section, and then override the "GetData" method. Let us examine it on the example of the TfrxCheckBoxView standard component.

The component can be connected to DB field via the "DataSet" and "DataField" properties, which are presented in the TfrxView basic class. In addition, this component has the "Expression" property, into which an expression can be placed. As soon as it is calculated, the result will be placed into the "Checked" property. The component displays a check, if the "Checked" property equals "True." Below is the initial text (the most important parts of it) of this component.

```
  TfrxCheckBoxView = class(TfrxView)
  private
    FChecked: Boolean;
    FExpression: String;
    procedure DrawCheck(ARect: TRect);
  public
    procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY:
Extended); override;
    procedure GetData; override;
  published
    property Checked: Boolean read FChecked write FChecked default True;
    property DataField;
    property DataSet;
    property Expression: String read FExpression write FExpression;
  end;

procedure TfrxCheckBoxView.Draw(Canvas: TCanvas; ScaleX, ScaleY,
OffsetX, OffsetY: Extended);
begin
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);

  DrawBackground;
  DrawCheck(Rect(FX, FY, FX1, FY1));
  DrawFrame;
end;

procedure TfrxCheckBoxView.GetData;
begin
  inherited;
  if IsDataField then
    FChecked := DataSet.Value[DataField]
  else if FExpression <> '' then
    FChecked := Report.Calc(FExpression);
end;
```

Writing custom common controls

FastReport contains a set of common controls, which can be placed on the dialogue form inside a report. They are the following elements:

```
TfrxLabelControl
TfrxEditControl
TfrxMemoControl
TfrxButtonControl
TfrxCheckBoxControl
TfrxRadioButtonControl
TfrxListBoxControl
TfrxComboBoxControl
TfrxDateEditControl
TfrxImageControl
TfrxBevelControl
TfrxPanelControl
TfrxGroupBoxControl
TfrxBitBtnControl
TfrxSpeedButtonControl
TfrxMaskEditControl
TfrxCheckListBoxControl
```

These elements correspond to the standard controls of the Delphi component palette. If standard functionality does not satisfy you, you can create your own common control and use it in your reports.

The basic class for all common controls is the TfrxDialogControl class described in the frxClass file:

```
TfrxDialogControl = class(TfrxReportComponent)
protected
  procedure InitControl(AControl: TControl);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  class function GetDescription: String; virtual;
  property Caption: String;
  property Color: TColor;
  property Control: TControl;
  property OnClick: TfrxNotifyEvent;
  property OnDblClick: TfrxNotifyEvent;
  property OnEnter: TfrxNotifyEvent;
  property OnExit: TfrxNotifyEvent;
  property OnKeyDown: TfrxKeyEvent;
  property OnKeyPress: TfrxKeyPressEvent;
  property OnKeyUp: TfrxKeyEvent;
  property OnMouseDown: TfrxMouseEvent;
  property OnMouseMove: TfrxMouseMoveEvent;
  property OnMouseUp: TfrxMouseEvent;
published
  property Left;
  property Top;
  property Width;
  property Height;
  property Font;
  property ParentFont;
  property Enabled: Boolean;
  property Visible;
end;
```

To create your own element, you should inherit from this class and override at least the constructor and the "GetDescription" methods. It will be necessary to create a common control and initialize it via the "InitControl" method in the constructor. The GetDescription method is to return the description of the common control. As you can see from the TfrxDialogControl class's description, it already contains a huge number of properties and methods in the public section. You need to transfer the necessary properties/events into the "published" section of your common control, and also to create new properties, which are typical for your element.

Registration and deleting of the common control is performed with the help of the frxObjects global object's methods declared in the frxDsgnIntf file:

```
frxObjects.RegisterObject(ClassRef: TfrxComponentClass; ButtonBmp:
TBitmap; const CategoryName: String = '');
frxObjects.Unregister(ClassRef: TfrxComponentClass);
```

During registration, you should specify control class' name, its picture, and the name of a category, to which it should be placed. If the name of a category is not specified, the control is placed in the basic component palette. The ButtonBmp size should be 22x22 pixels.

Categories allow grouping the objects' pictures in the toolbar, according to their functions. The categories are also used for saving space in the toolbar, since its size is limited. Nevertheless, it can contain a large number of objects.

To register a category, use the following method of the "frxObjects" object described in the "frxDsgnIntf" file:

```
frxObjects.RegisterCategory(const CategoryName: String; ButtonBmp:
TBitmap; const ButtonHint: String; ImageIndex: Integer = -1);
```

The first parameter is the name of a category. This name is used for identification of the category only and does not appear anywhere else. It should also be used when calling the "frxObjects.RegisterObject" procedure, if you want to place the registered object inside the given category. The second parameter is a picture; its size is 22x22 pixels. The third parameter is the category's description; it is displayed as a hint, if the button with a category is selected by the cursor.

Let us examine an example of the common control, which realizes simplified functionality of the standard Delphi TBitBtn control.

```
uses frxClass, frxDsgnIntf, Buttons;

type
  TfrxBitBtnControl = class(TfrxDialogControl)
  private
    FButton: TBitBtn;
    procedure SetKind(const Value: TBitBtnKind);
    function GetKind: TBitBtnKind;
```

```
  public
    constructor Create(AOwner: TComponent); override;
    class function GetDescription: String; override;
    property Button: TBitBtn read FButton;
  published
    { add new properties }
    property Kind: TBitBtnKind read GetKind write SetKind default
bkCustom;
    { these properties are already declared in the parent class }
    property Caption;
    property OnClick;
    property OnEnter;
    property OnExit;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;


constructor TfrxBitBtnControl.Create(AOwner: TComponent);
begin
  { default constructor }
  inherited;
  { create the required common control }
  FButton := TBitBtn.Create(nil);
  FButton.Caption := 'BitBtn';
  { initialize it }
  InitControl(FButton);

  { it will have such size by default }
  Width := 75;
  Height := 25;
end;

class function TfrxBitBtnControl.GetDescription: String;
begin
  Result := 'BitBtn control';
end;

procedure TfrxBitBtnControl.SetKind(const Value: TBitBtnKind);
begin
  FButton.Kind := Value;
end;

function TfrxBitBtnControl.GetKind: TBitBtnKind;
begin
  Result := FButton.Kind;
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  {Load a picture from a resource. Of course, you should beforehand
place it there.}
```

```
  Bmp.LoadFromResourceName(hInstance, 'frxBitBtnControl');
  frxObjects.RegisterObject(TfrxBitBtnControl, Bmp);

finalization
  frxObjects.Unregister(TfrxBitBtnControl);
  Bmp.Free;

end.
```

Creating an event handler

What should be done, if it is necessary to define a new event handler, which does not belong to the basic class? Let us examine it on the example of the "TfrxEditControl" common control:

```
  TfrxEditControl = class(TfrxDialogControl)
  private
    FEdit: TEdit;
    { new event }
    FOnChange: TfrxNotifyEvent;
    procedure DoOnChange(Sender: TObject);
    ...
  public
    constructor Create(AOwner: TComponent); override;
    ...
  published
    { new event }
    property OnChange: TfrxNotifyEvent read FOnChange write FOnChange;
    ...
  end;

constructor TfrxEditControl.Create(AOwner: TComponent);
begin
  ...
  { connect our handler }
  FEdit.OnChange := DoOnChange;
  InitControl(FEdit);
  ...
end;

procedure TfrxEditControl.DoOnChange(Sender: TObject);
begin
  { call event's handler }
  if Report <> nil then
    Report.DoNotifyEvent(Sender, FOnChange);
end;
```

It is important to notice that events in FastReport are a procedure declared in the report's script. A string containing its name will be the link to such handler. That is why, for example, unlike the Delphi TNotifyEvent type, which is the method's address, the handler's type in FastReport is a string (the TfrxNotifyEvent type is declared as String [63]).

Registering a component in the script system

To refer to our component from the script, it is necessary to register its class, its properties, and methods in the script system. The register code, according to the agreement accepted in FastReport, may be placed in a file with a name similar to the name of the file with the component's code, adding the RTTI suffix (for example, frxBitBtnRTTI.pas in our case). See more about registration of classes, their methods and properties in the FastScript script library's documentation.

```
uses fs_iinterpreter, frxBitBtn, frxClassRTTI;

type
  TFunctions = class(TObject)
  public
    constructor Create;
    destructor Destroy; override;
  end;

var
  Functions: TFunctions;

constructor TFunctions.Create;
begin
  { fsGlobalUnit is a variable defined in the fs_iinterpreter unit. It
contains the description }
  { of all classes, methods, types, variables etc. are registered in the
script system }
  with fsGlobalUnit do
  begin
    AddedBy := Self;

    { register a class, and then define its parent }
    AddClass(TfrxBitBtnControl, 'TfrxDialogControl');

    { if there are several common controls in your unit, they can be
registered right here }
    { for example, AddClass(TfrxAnotherControl, 'TfrxDialogControl'); }

    AddedBy := nil;
  end;
end;

destructor TFunctions.Destroy;
begin
  if fsGlobalUnit <> nil then
    fsGlobalUnit.RemoveItems(Self);
  inherited;
end;


initialization
  Functions := TFunctions.Create;

finalization
  Functions.Free;

end.
```

Writing a component editor

Any common control's editor (it can be called from the element's context menu or by double-clicking) creates an OnClick blank event's handler by default. This behavior can be replaced with writing a custom editor. In addition, the editor allows adding own items to the component's context menu.

The basic class for all editors is described in the frxDsgnIntf file:

```
TfrxComponentEditor = class(TObject)
protected
  function AddItem(Caption: String; Tag: Integer;
    Checked: Boolean = False): TMenuItem;
public
  function Edit: Boolean; virtual;
  function HasEditor: Boolean; virtual;
  function Execute(Tag: Integer; Checked: Boolean): Boolean; virtual;
  procedure GetMenuItems; virtual;
  property Component: TfrxComponent readonly;
  property Designer: TfrxCustomDesigner readonly;
end;
```

If your editor does not create own items in the contextual menu, you will need to override two methods, i.e. "Edit" and "HasEditor." The first method performs necessary actions (for example, displays a dialogue box) and returns "True," if the component's state was modified. The "HasEditor" method should return "True" if your component has an editor. If it either returns "False" or you do not override this method, the editor will not be called. This becomes necessary, if your component does not have an editor and you wish to add items into the component's context menu.

If the editor adds items into the context menu, you should override the "GetMenuItems" (in this method, you can create a menu with the help of calling in the AddItem function) and "Execute" (this method is called, when you select one of your items in the component's menu; response to the selected menu item should be described here) methods.

Registration of the editor is performed via the procedure described in the "frxDsgnIntf" file:

```
frxComponentEditors.Register(ComponentClass: TfrxComponentClass;
ComponentEditor: TfrxComponentEditorClass);
```

The first parameter is the class' name, for which the editor is to be created. The second parameter is the name of the editor's class.

Let us examine a simple editor for our common control, which will display a window with the name of our element and add the "Enabled" and "Visible" items to the

element's context menu (when items are selected, the element's "Enabled" and "Visible" properties will change). The editor code, according to the agreement accepted in FastReport, can be placed in a file having the same name as the file with the very component's code, adding the Editor suffix (for example, frxBitBtnEditor.pas in our case).

```
uses frxClass, frxDsgnIntf, frxBitBtn;

type
  TfrxBitBtnEditor = class(TfrxComponentEditor)
  public
    function Edit: Boolean; override;
    function HasEditor: Boolean; override;
    function Execute(Tag: Integer; Checked: Boolean): Boolean; override;
    procedure GetMenuItems; override;
  end;

function TfrxBitBtnEditor.Edit: Boolean;
var
  c: TfrxBitBtnControl;
begin
  Result := False;
  { the Component property is the edited component. In this case, it is
the TfrxBitBtnControl }
  c := TfrxBitBtnControl(Component);
  ShowMessage('This is ' + c.Name);
end;

function TfrxBitBtnEditor.HasEditor: Boolean;
begin
  Result := True;
end;

function TfrxBitBtnEditor.Execute(Tag: Integer; Checked: Boolean):
Boolean;
var
  c: TfrxBitBtnControl;
begin
  Result := True;
  c := TfrxBitBtnControl(Component);
  if Tag = 1 then
    c.Enabled := Checked
  else if Tag = 2 then
    c.Visible := Checked;
end;

procedure TfrxBitBtnEditor.GetMenuItems;
var
  c: TfrxBitBtnControl;
begin
  c := TfrxBitBtnControl(Component);
  { the AddItem method's parameters: name of the menu item, its tag and
Checked/Unchecked condition }
  AddItem('Enabled', 1, c.Enabled);
  AddItem('Visible', 2, c.Visible);
end;
```

```
initialization
  frxComponentEditors.Register(TfrxBitBtnControl, TfrxBitBtnEditor);

end.
```

Writing a property editor

When you select a component in the designer, its properties are displayed in the object inspector. You can create your own editor for any property. The "Font" property's standard editor can exemplify that: if this property is selected, the ... button appears in the right part of the line; call the standard "font properties" dialogue box by clicking the button. One more example is the "Color" property's editor. It shows names of standard colors and color specimens in the drop-down list.

The base class for all property editors is described in the "frxDsgnIntf" unit:

```
  TfrxPropertyEditor = class(TObject)
  protected
    procedure GetStrProc(const s: String);
    function GetFloatValue: Extended;
    function GetOrdValue: Integer;
    function GetStrValue: String;
    function GetVarValue: Variant;
    procedure SetFloatValue(Value: Extended);
    procedure SetOrdValue(Value: Integer);
    procedure SetStrValue(const Value: String);
    procedure SetVarValue(Value: Variant);
  public
    constructor Create(Designer: TfrxCustomDesigner); virtual;
    destructor Destroy; override;
    function Edit: Boolean; virtual;
    function GetAttributes: TfrxPropertyAttributes; virtual;
    function GetExtraLBSize: Integer; virtual;
    function GetValue: String; virtual;
    procedure GetValues; virtual;
    procedure SetValue(const Value: String); virtual;
    procedure OnDrawLBItem(Control: TWinControl; Index: Integer; ARect:
TRect; State: TOwnerDrawState); virtual;
    procedure OnDrawItem(Canvas: TCanvas; ARect: TRect); virtual;
    property Component: TPersistent readonly;
    property frComponent: TfrxComponent readonly;
    property Designer: TfrxCustomDesigner readonly;
    property ItemHeight: Integer;
    property PropInfo: PPropInfo readonly;
    property Value: String;
    property Values: TStrings readonly;
  end;
```

You also can inherit from any of the following classes which themselves realize some basic functionality for working with properties of corresponding types:

```
  TfrxIntegerProperty = class(TfrxPropertyEditor)
  TfrxFloatProperty = class(TfrxPropertyEditor)
```

```
  TfrxCharProperty = class(TfrxPropertyEditor)
  TfrxStringProperty = class(TfrxPropertyEditor)
  TfrxEnumProperty = class(TfrxPropertyEditor)
  TfrxClassProperty = class(TfrxPropertyEditor)
  TfrxComponentProperty = class(TfrxPropertyEditor)
```

Several properties are defined in the class:

- Component - link to the parent component (not to the property itself!), to which the given property belongs;
- frComponent - the same, but casted to the TfrxComponent type (for convenience of use in some cases);
- Designer – the link to the report's designer;
- ItemHeight - height of the item, in which the property is displayed. It can be useful in the OnDrawXXX;
- PropInfo - link to the PPropInfo structure, which contains information about the edited property;
- Value - property's value displayed as a string;
- Values - the list of values. This property is to be filled in the "GetValue" method, if the "paValueList" attribute is defined (see below).

The following methods are service ones. They can be used to get or set a value of the edited property.

```
    function GetFloatValue: Extended;
    function GetOrdValue: Integer;
    function GetStrValue: String;
    function GetVarValue: Variant;
    procedure SetFloatValue(Value: Extended);
    procedure SetOrdValue(Value: Integer);
    procedure SetStrValue(const Value: String);
    procedure SetVarValue(Value: Variant);
```

You should use the methods, which correspond to the property's type. Thus, use the "GetOrdValue" and the "SetOrdValue" methods, if a property is of the "Integer" type. These methods are also used for working with a property of the "TObject" type, since such property contains the 32-bit object's address. In this case, it is sufficient to do a cast of the following type, for example: MyFont := TFont(GetOrdValue).

To create your own editor, it is necessary to inherit from the basic class and override one or several methods defined in the public section (this depends on the property type and functionality you wish to realize). One of the methods you surely have to override is the "GetAttributes" method. The "GetAttributes" method is to return a set of the property's attributes. The attributes are defined in the following way:

```
  TfrxPropertyAttribute = (paValueList, paSortList, paDialog,
paMultiSelect, paSubProperties, paReadOnly, paOwnerDraw);
  TfrxPropertyAttributes = set of TfrxPropertyAttribute;
```

Assignment of the attribute is realized as following:

- paValueList - the property represents the dropping down list of values. (This function is exemplified in the "Color" property). If this attribute is presented, the "GetValues" method should be overridden;
- paSortList - sorts the list's elements. It is used together with paValueList;
- paDialog - the property has an editor. If this attribute is presented, the ... button is displayed in the right part of the editing line. The Edit method is called on by clicking on it;
- paMultiSelect - allow editing of the given property in some objects selected at the same time. Some properties (such as "Name", etc) do not have this attribute;
- paSubProperties - the property is an object of the TPersistent type and has its own properties, which are also should be displayed. (This function is exemplified in the "Font" property);
- paReadOnly - it is impossible to modify a value in the editor line. Some properties, being the "Class" or "Set" types, possess this attribute;
- paOwnerDraw - drawing of the property's value is performed via the "OnDrawItem" method. If the "paValueList" attribute is defined, then drawing of the drop-down list is performed via the OnDrawLBItem method.

The Edit method is called in two cases: either by selecting a property, by double-clicking its value, or (if a property has the paDialog attribute) by clicking the ... button. The method should return "True," if the property's value was modified.

The "GetValue" method should return the property's value as a string (it will be displayed in the object inspector). If you inherit from the TfrxPropertyEditor basic class, it is necessary to override the method.

The "SetValue" method is to set the property's value transferred as a string. If you inherit from the TfrxPropertyEditor basic class, it is necessary to override the method.

The "GetValues" method should be overridden in case you defined the "paValueList" attribute. This method should fill the "Values" property with values.

The following three methods allow performing manual drawing of the property's value (the Color property's editor works in the same way). These methods are called, if you define the "paOwnerDraw" attribute.

The "OnDrawItem" method is called when drawing the property's value in the object inspector (when the property is not selected; otherwise its value is simply displayed in the editing line). For example, the Color property's editor draws a rectangle, filled with the color according to the value, at the left of the property's value.

The "GetExtraLBSize" method is called in case you defined the "paValueList" attribute. The method returns the number of pixels, by which the width of the "Drop-Down List" should be adjusted in order to find room for the displayed picture. By default, this method returns the value corresponding to the cell's height for property's enveloping.

If you need to deduce a picture, width of which is larger than its height, the given method should be overridden.

The "OnDrawLBItem" method is called when drawing a string in the drop-down list, if you defined the paValueList attribute. In fact, this method is the TListBox.OnDrawItem event's handler and has the same set of parameters.

Registration of the property's editor is performed via the procedure described in the frxDsgnIntf file:

```
procedure frxPropertyEditors.Register(PropertyType: PTypeInfo;
ComponentClass: TClass; const PropertyName: String; EditorClass:
TfrxPropertyEditorClass);
```

- PropertyType - information about the property's type, transferred via the "TypeInfo" system function, for example TypeInfo(String);
- ComponentClass - name of the component, the property of which you want to edit (may be nil);
- PropertyName - name of the property you want to edit (may be a blank string);
- EditorClass - name of the property's editor.

It is necessary to specify the "PropertyType" parameter only. The "ComponentClass" and/or "PropertyName" parameters may be blank. This allows to register the editor either to any property of the PropertyType type, to any property of the concrete ComponentClass components and its successors, or to the PropertyName concrete property of the concrete component (or any component, if the ComponentClass parameter is blank).

Let us examine three examples of the properties' editors. The editor's code, according to the agreement accepted in FastReport, can be placed in a file possessing the same name as the file with the code of the very component, adding the Editor suffix.

```
-------------------------------------------------

{ the TFont property's editor displays the editor's button(...) }
{ inherit from the ClassProperty }
type
  TfrxFontProperty = class(TfrxClassProperty)
  public
    function Edit: Boolean; override;
    function GetAttributes: TfrxPropertyAttributes; override;
  end;

function TfrxFontProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { the property has nested properties and the editor. It cannot be
edited manually }
  Result := [paMultiSelect, paDialog, paSubProperties, paReadOnly];
end;
```

```
function TfrxFontProperty.Edit: Boolean;
var
  FontDialog: TFontDialog;
begin
  { create a standard dialogue }
  FontDialog := TFontDialog.Create(Application);
  try
    { take the property's value }
    FontDialog.Font := TFont(GetOrdValue);
    FontDialog.Options := FontDialog.Options + [fdForceFontExist];
    { display a dialogue }
    Result := FontDialog.Execute;
    { bind a new value }
    if Result then
      SetOrdValue(Integer(FontDialog.Font));
  finally
    FontDialog.Free;
  end;
end;

{ registration }
frxPropertyEditors.Register(TypeInfo(TFont), nil, '', TfrxFontProperty);

------------------------------------------------

{ the TFont.Name property's editor displays the drop-down list of
available fonts }
{ inherit from the StringProperty, as the property is of the string type
}
type
  TfrxFontNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;

function TfrxFontNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxFontNameProperty.GetValues;
begin
  Values.Assign(Screen.Fonts);
end;

{ registration }
frxPropertyEditors.Register(TypeInfo(String), TFont, 'Name',
TfrxFontNameProperty);


------------------------------------------------

{ the TPen.Style. property's editor displays a picture, which is a
pattern of the selected style }
type
  TfrxPenStyleProperty = class(TfrxEnumProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
```

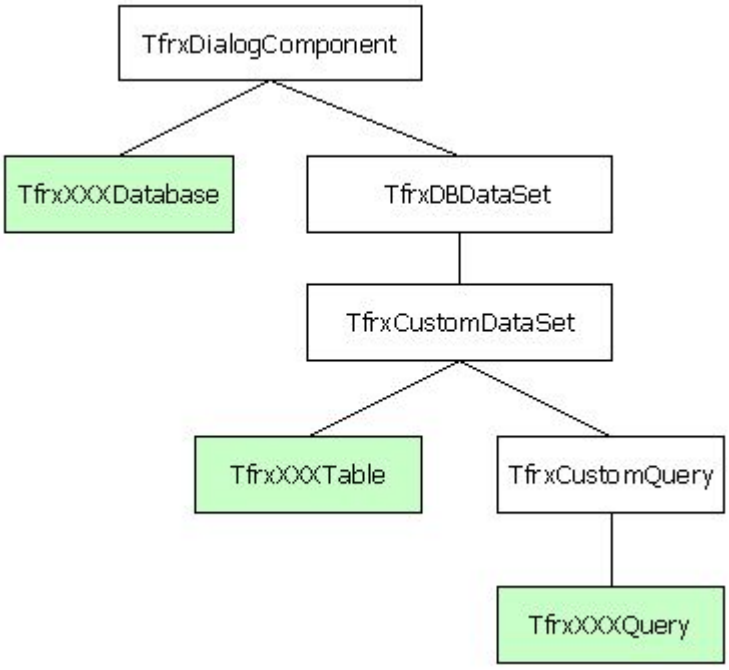
```
    function GetExtraLBSize: Integer; override;
    procedure OnDrawLBItem(Control: TWinControl; Index: Integer;
      ARect: TRect; State: TOwnerDrawState); override;
    procedure OnDrawItem(Canvas: TCanvas; ARect: TRect); override;
  end;

function TfrxPenStyleProperty.GetAttributes: TfrxPropertyAttributes;
begin
  Result := [paMultiSelect, paValueList, paOwnerDraw];
end;

{ the method draws a thick horizontal line with the selected style }
procedure HLine(Canvas: TCanvas; X, Y, DX: Integer);
var
  i: Integer;
begin
  with Canvas do
  begin
    Pen.Color := clBlack;
    for i := 0 to 1 do
    begin
      MoveTo(X, Y - 1 + i);
      LineTo(X + DX, Y - 1 + i);
    end;
  end;
end;

{ drawing the drop-down list }
procedure TfrxPenStyleProperty.OnDrawLBItem(Control: TWinControl; Index:
Integer; ARect: TRect; State: TOwnerDrawState);
begin
  with TListBox(Control), TListBox(Control).Canvas do
  begin
    FillRect(ARect);
    TextOut(ARect.Left + 40, ARect.Top + 1, TListBox(Control).Items
[Index]);

    Pen.Color := clGray;
    Brush.Color := clWhite;
    Rectangle(ARect.Left + 2, ARect.Top + 2, ARect.Left + 36,
ARect.Bottom - 2);

    Pen.Style := TPenStyle(Index);
    HLine(TListBox(Control).Canvas, ARect.Left + 3, ARect.Top +
(ARect.Bottom - ARect.Top) div 2, 32);
    Pen.Style := psSolid;
  end;
end;

{ drawing the property value }
procedure TfrxPenStyleProperty.OnDrawItem(Canvas: TCanvas; ARect:
TRect);
begin
  with Canvas do
  begin
    TextOut(ARect.Left + 38, ARect.Top, Value);

    Pen.Color := clGray;
    Brush.Color := clWhite;
```

```
    Rectangle(ARect.Left, ARect.Top + 1, ARect.Left + 34, ARect.Bottom -
4);

    Pen.Color := clBlack;
    Pen.Style := TPenStyle(GetOrdValue);
    HLine(Canvas, ARect.Left + 1, ARect.Top + (ARect.Bottom - ARect.Top)
div 2 - 1, 32);
    Pen.Style := psSolid;
  end;
end;

{ return the picture's width }
function TfrxPenStyleProperty.GetExtraLBSize: Integer;
begin
  Result := 36;
end;

{ registration }
frxPropertyEditors.Register(TypeInfo(TPenStyle), TPen, 'Style',
TfrxPenStyleProperty);
```

Writing custom DB engines

FastReport allows building reports not only on the basis of data defined in the application. You can define your own data sources (connections to DB, queries) right in the report as well. FastReport is supplied with engines for ADO, BDE, IBX. You can create your own engine, and then connect it to FastReport.

The picture below shows the hierarchy of classes intended for creating DB engines. The components of a new engine are highlighted with green color.



As you can see, a standard set of the DB engine's components includes Database, Table and Query. You can realize all these components or some of them (for example, many DB have no component of the Table type). You can also realize components, which are not included into the standard set (for example, the StoredProc analogue).

Let us examine basic classes in detail.

TfrxDialogComponent is a basic class for all non-visual components, which can be placed into the FastReport dialogue form. There are no any important properties or methods defined in it.

TfrxCustomDataSet is a basic class of DB components derived from TDataSet. The components inherited from this class are clones of "Query," "Table," and "StoredProc." As a matter of fact, the class represents a cover over TDataSet.

```
TfrxCustomDataset = class(TfrxDBDataSet)
protected
  procedure SetMaster(const Value: TDataSource); virtual;
  procedure SetMasterFields(const Value: String); virtual;
public
  property DataSet: TDataSet;
  property Fields: TFields readonly;
  property MasterFields: String;
  property Active: Boolean;
published
  property Filter: String;
  property Filtered: Boolean;
  property Master: TfrxDBDataSet;
end;
```

The following properties are defined in the class:

- DataSet is a link to the buried object of the "TdataSet" type;
- Fields is a link to the DataSet.Fields;
- Active - whether a data set is active;
- Filter - expression for filtering;
- Filtered – whether filtering is active;
- Master is a link to master dataset in a master-detail relationship.
- MasterFields is a list of fields like field1=field2. Used for master-detail relations.

A component of the Table type inherits from the given class. For its realization, it is necessary to define lacking properties; as a rule, they are: "Database," "IndexName," and "TableName." Also you should override SetMaster, SetMasterFields methods to allow master-detail relations.

TfrxCustomQuery is a basic class for DB components of the "Query" type. The class is a cover for a Query type component.

```
TfrxCustomQuery = class(TfrxCustomDataset)
protected
  procedure SetSQL(Value: TStrings); virtual; abstract;
  function GetSQL: TStrings; virtual; abstract;
public
  procedure UpdateParams; virtual; abstract;
published
  property Params: TfrxParams;
  property SQL: TStrings;
end;
```

The "SQL" and "Params" properties (which are general for all Query components) are defined in the class. Since different Query components have different realization of parameters (for example, TParams and TParameters), the "Params" property has the "TfrxParams" type and is a cover for the concrete parameters' type.

The following methods are defined in this class:

- SetSQL is to set the "SQL" property of the component of the "Query" type;
- GetSQL is to get the "SQL" property of the component of the "Query" type;
- UpdateParams is to copy parameters' values into the component of the Query type. If a Query component's parameters are of the TParams type, copying is performed via the frxParamsToTParams standard procedure.

Let us illustrate creation process of the DB engine by the IBX example. The full original text of the engine can be found in the SOURCE\IBX directory. Below are some quotations from the source text with our comments.

The IBX components around which we will build a cover are: TIBDatabase, TIBTable, and TIBQuery. Accordingly, our components will be named "TfrxIBXDatabase," "TfrxIBXTable," and "TfrxIBXQuery."

"TfrxIBXComponents" is another component we should create; it will be placed into the FastReport component palette when registering the engine (in Delphi environment). As soon as this component is placed into a project, Delphi automatically adds a link to the unit of our engine into the "Uses" list. It is convenient to assign one more task in this component, i.e. to define the "DefaultDatabase" property in it, which refers to the existing connection to DB. By default, all the TfrxIBXTable and TfrxIBXQuery components will refer to this connection. It is necessary to inherit the component from the TfrxDBComponents class:

```
  TfrxDBComponents = class(TComponent)
  public
    function GetDescription: String; virtual; abstract;
  end;
```

The description should be returned by one function only, for example "IBX Components." We plan to add new methods to the list in the future in order to provide support of the visual query builders. Realization of the "TfrxIBXComponents" component is as following:

```
type
  TfrxIBXComponents = class(TfrxDBComponents)
  private
    FDefaultDatabase: TIBDatabase;
    FOldComponents: TfrxIBXComponents;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    function GetDescription: String; override;
  published
    property DefaultDatabase: TIBDatabase read FDefaultDatabase write
FDefaultDatabase;
  end;

var
  IBXComponents: TfrxIBXComponents;

constructor TfrxIBXComponents.Create(AOwner: TComponent);
begin
  inherited;
  FOldComponents := IBXComponents;
  IBXComponents := Self;
end;

destructor TfrxIBXComponents.Destroy;
begin
  if IBXComponents = Self then
    IBXComponents := FOldComponents;
  inherited;
end;
```

```
function TfrxIBXComponents.GetDescription: String;
begin
  Result := 'IBX';
end;
```

We define the IBXComponents global variable, which will refer to the TfrxIBXComponents component's copy. If you place the component into the project several times (though it is senseless), you will nevertheless be able to save the link to the previous component and restore it after deleting the component.

A link to the connection to DB, which already exists in the project, can be placed into the "DefaultDatabase" property. The way we will write the TfrxIBXTable, TfrxIBXQuery components allows them using this connection by default (actually, this is what we need the IBXComponents global variable for).

The following component is the TfrxIBXDatabase one. It represents a cover over the TIBDatabase.

```
  TfrxIBXDatabase = class(TfrxDialogComponent)
  private
    FDatabase: TIBDatabase;
    FTransaction: TIBTransaction;
    procedure SetConnected(Value: Boolean);
    procedure SetDatabaseName(const Value: String);
    procedure SetLoginPrompt(Value: Boolean);
    procedure SetParams(Value: TStrings);
    function GetConnected: Boolean;
    function GetDatabaseName: String;
    function GetLoginPrompt: Boolean;
    function GetParams: TStrings;
    function GetSQLDialect: Integer;
    procedure SetSQLDialect(const Value: Integer);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    class function GetDescription: String; override;
    property Database: TIBDatabase read FDatabase;
  published
    { list TIBDatabase properties }
    property DatabaseName: String read GetDatabaseName write
SetDatabaseName;
    property LoginPrompt: Boolean read GetLoginPrompt write
SetLoginPrompt default True;
    property Params: TStrings read GetParams write SetParams;
    property Connected: Boolean read GetConnected write SetConnected
default False;
    property SQLDialect: Integer read GetSQLDialect write SetSQLDialect;
  end;

constructor TfrxIBXDatabase.Create(AOwner: TComponent);
begin
  inherited;
  { create a component - connection }
```

```
  FDatabase := TIBDatabase.Create(nil);
  { create a component - transaction (specificity of the IBX) }
  FTransaction := TIBTransaction.Create(nil);
  FDatabase.DefaultTransaction := FTransaction;
  { do not forget this string! }
  Component := FDatabase;
  { component's icon - take it from the standard set }
  FImageIndex := 37;
end;

destructor TfrxIBXDatabase.Destroy;
begin
  { delete the transaction }
  FTransaction.Free;
  { the connection will be deleted automatically in the parent class }
  inherited;
end;

{ component's description will be displayed next to the icon in the
objects toolbar }
class function TfrxIBXDatabase.GetDescription: String;
begin
  Result := 'IBX Database';
end;

{ redirect component's properties to the cover's properties and vice
versa }
function TfrxIBXDatabase.GetConnected: Boolean;
begin
  Result := FDatabase.Connected;
end;

function TfrxIBXDatabase.GetDatabaseName: String;
begin
  Result := FDatabase.DatabaseName;
end;

function TfrxIBXDatabase.GetLoginPrompt: Boolean;
begin
  Result := FDatabase.LoginPrompt;
end;

function TfrxIBXDatabase.GetParams: TStrings;
begin
  Result := FDatabase.Params;
end;

procedure TfrxIBXDatabase.SetConnected(Value: Boolean);
begin
  FDatabase.Connected := Value;
  FTransaction.Active := Value;
end;

procedure TfrxIBXDatabase.SetDatabaseName(const Value: String);
begin
  FDatabase.DatabaseName := Value;
end;

procedure TfrxIBXDatabase.SetLoginPrompt(Value: Boolean);
```

```
begin
  FDatabase.LoginPrompt := Value;
end;

procedure TfrxIBXDatabase.SetParams(Value: TStrings);
begin
  FDatabase.Params := Value;
end;

function TfrxIBXDatabase.GetSQLDialect: Integer;
begin
  Result := FDatabase.SQLDialect;
end;

procedure TfrxIBXDatabase.SetSQLDialect(const Value: Integer);
begin
  FDatabase.SQLDialect := Value;
end;
```

As you can see, this is not that complicated. We create the FDatabase: "TIBDatabase" object, and then define properties we want the designer to possess. The "Get" and "Set" methods are written for each property.

The next class is TfrxIBXTable. It inherits, as it was mentioned above, from the TfrxCustomDataSet standard class. All basic functionality (operating with the list of fields, master-detail, basic properties) is already realized in the basic class. We only need to define properties, which are specific for the given component.

```
  TfrxIBXTable = class(TfrxCustomDataset)
  private
    FDatabase: TfrxIBXDatabase;
    FTable: TIBTable;
    procedure SetIndexName(const Value: String);
    function GetIndexName: String;
    function GetTableName: String;
    procedure SetTableName(const Value: String);
    procedure SetDatabase(const Value: TfrxIBXDatabase);
  protected
    procedure SetMaster(const Value: TDataSource); override;
    procedure SetMasterFields(const Value: String); override;
  public
    constructor Create(AOwner: TComponent); override;
    class function GetDescription: String; override;
    property Table: TIBTable read FTable;
  published
    property Database: TfrxIBXDatabase read FDatabase write SetDatabase;
    property IndexName: String read GetIndexName write SetIndexName;
    property TableName: String read GetTableName write SetTableName;
  end;

constructor TfrxIBXTable.Create(AOwner: TComponent);
begin
  { create a component - a table }
  FTable := TIBTable.Create(nil);
```

```
  { assign a link to the DataSet property from the basic class - do not
forget this string! }
  DataSet := FTable;
  { assign a link to connection to DB by default }
  SetDatabase(nil);
  { after that the basic constructor may be called in}
  inherited;
  { component's icon; we take it from the standard set }
  FImageIndex := 38;
end;

class function TfrxIBXTable.GetDescription: String;
begin
  Result := 'IBX Table';
end;

procedure TfrxIBXTable.SetDatabase(const Value: TfrxIBXDatabase);
begin
  { the Database property of the TfrxIBXDatabase type, and not of the
TIBDatabase one! }
  FDatabase := Value;
  { if a value <> nil, connect a table to the selected component }
  if Value <> nil then
    FTable.Database := Value.Database
  { otherwise, try to connect to DB by default, defined in the
TfrxIBXComponents component }
  else if IBXComponents <> nil then
    FTable.Database := IBXComponents.DefaultDatabase
  { if there were no TfrxIBXComponents for some reason, reset to nil }
  else
    FTable.Database := nil;
end;

function TfrxIBXTable.GetIndexName: String;
begin
  Result := FTable.IndexName;
end;

function TfrxIBXTable.GetTableName: String;
begin
  Result := FTable.TableName;
end;

procedure TfrxIBXTable.SetIndexName(const Value: String);
begin
  FTable.IndexName := Value;
end;

procedure TfrxIBXTable.SetTableName(const Value: String);
begin
  FTable.TableName := Value;
end;

procedure TfrxIBXTable.SetMaster(const Value: TDataSource);
begin
  FTable.MasterSource := Value;
end;

procedure TfrxIBXTable.SetMasterFields(const Value: String);
```

```
begin
  FTable.MasterFields := Value;
end;
```

Finally, lets examine the last component, "TfrxIBXQuery". It inherits from the TfrxCustomQuery basic class, in which the necessary properties are already defined. We only need to define the Database property and override the SetMaster method.

```
  TfrxIBXQuery = class(TfrxCustomQuery)
  private
    FDatabase: TfrxIBXDatabase;
    FQuery: TIBQuery;
    procedure SetDatabase(const Value: TfrxIBXDatabase);
  protected
    procedure SetMaster(const Value: TDataSource); override;
    procedure SetSQL(Value: TStrings); override;
    function GetSQL: TStrings; override;
  public
    constructor Create(AOwner: TComponent); override;
    class function GetDescription: String; override;
    procedure UpdateParams; override;
    property Query: TIBQuery read FQuery;
  published
    property Database: TfrxIBXDatabase read FDatabase write SetDatabase;
  end;

constructor TfrxIBXQuery.Create(AOwner: TComponent);
begin
  { create a component - query }
  FQuery := TIBQuery.Create(nil);
  { assign a link to it to the DataSet property from the basic class -
do not forget this line! }
  Dataset := FQuery;
  { assign a link to the connection to DB by default }
  SetDatabase(nil);
  { after that a basic constructor may be called in }
  inherited;
  { component's icon - take it from the standard set }
  FImageIndex := 39;
end;

class function TfrxIBXQuery.GetDescription: String;
begin
  Result := 'IBX Query';
end;

procedure TfrxIBXQuery.SetDatabase(const Value: TfrxIBXDatabase);
begin
  { realization is analogical to TfrxIBXTable.SetDatabase }
  FDatabase := Value;
  if Value <> nil then
    FQuery.Database := Value.Database
  else if IBXComponents <> nil then
    FQuery.Database := IBXComponents.DefaultDatabase
  else
    FQuery.Database := nil;
```

```
end;

procedure TfrxIBXQuery.SetMaster(const Value: TDataSource);
begin
  FQuery.DataSource := Value;
end;

function TfrxIBXQuery.GetSQL: TStrings;
begin
  Result := FQuery.SQL;
end;

procedure TfrxIBXQuery.SetSQL(Value: TStrings);
begin
  FQuery.SQL := Value;
end;

procedure TfrxIBXQuery.UpdateParams;
begin
  { in this method it is sufficient to assign values from Params into
FQuery.Params }
  { this is performed via the standard procedure }
  frxParamsToTParams(Self, FQuery.Params);
end;
```

Registration of all engine's components is performed in the "Initialization" section. The category, where all the components are placed, is registered in the first place.

```
var
  CatBmp: TBitmap;

initialization
  CatBmp := TBitmap.Create;
  CatBmp.LoadFromResourceName(hInstance, 'frxIBX');
  frxObjects.RegisterCategory('IBX', CatBmp, 'IBX Components');
  { use indexes of standard pictures 37,38,39 instead of pictures}
  frxObjects.RegisterObject1(TfrxIBXDataBase, nil, '', 'IBX', 0, 37);
  frxObjects.RegisterObject1(TfrxIBXTable, nil, '', 'IBX', 0, 38);
  frxObjects.RegisterObject1(TfrxIBXQuery, nil, '', 'IBX', 0, 39);

finalization
  CatBmp.Free;
  frxObjects.Unregister(TfrxIBXDataBase);
  frxObjects.Unregister(TfrxIBXTable);
  frxObjects.Unregister(TfrxIBXQuery);

end.
```

It is quite enough for using the engine in reports. There are two more things left at this stage: to register engine's classes in the script system in order to make them referable from the script, and to register editors of several properties (for example, TfrxIBXTable.TableName) to make the work with the component more convenient.

It is better to store the engine's registration code in a separate file with the RTTI

suffix. See more about registration of classes in the script system in the corresponding chapter. Here is an example of such file:

```
unit frxIBXRTTI;

interface

{$I frx.inc}

implementation

uses
  Windows, Classes, fs_iinterpreter, frxIBXComponents
{$IFDEF Delphi6}
, Variants
{$ENDIF};

type
  TFunctions = class(TObject)
  public
    constructor Create;
    destructor Destroy; override;
  end;

var
  Functions: TFunctions;


{ TFunctions }

constructor TFunctions.Create;
begin
  with fsGlobalUnit do
  begin
    AddedBy := Self;
    AddClass(TfrxIBXDatabase, 'TfrxComponent');
    AddClass(TfrxIBXTable, 'TfrxCustomDataset');
    AddClass(TfrxIBXQuery, 'TfrxCustomQuery');
    AddedBy := nil;
  end;
end;

destructor TFunctions.Destroy;
begin
  if fsGlobalUnit <> nil then
    fsGlobalUnit.RemoveItems(Self);
  inherited;
end;

initialization
  Functions := TFunctions.Create;

finalization
  Functions.Free;

end.
```

It is recommended to place the code of properties' editors to a separate file with the Editor suffix as well. In our case, it is necessary to write editors to the TfrxIBXDatabase.DatabaseName, TfrxIBXTable.IndexName, TfrxIBXTable.TableName properties. See more about writing properties' editors in the corresponding chapter. Below is an example of such file:

```
unit frxIBXEditor;

interface

{$I frx.inc}

implementation

uses
  Windows, Classes, SysUtils, Forms, Dialogs, frxIBXComponents,
frxCustomDB,
  frxDsgnIntf, frxRes, IBDatabase, IBTable
{$IFDEF Delphi6}
, Variants
{$ENDIF};

type
  TfrxDatabaseNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    function Edit: Boolean; override;
  end;

  TfrxTableNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;

  TfrxIndexNameProperty = class(TfrxStringProperty)
  public
    function GetAttributes: TfrxPropertyAttributes; override;
    procedure GetValues; override;
  end;


{ TfrxDatabaseNameProperty }

function TfrxDatabaseNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { this property possesses the editor }
  Result := [paDialog];
end;

function TfrxDatabaseNameProperty.Edit: Boolean;
var
  SaveConnected: Bool;
  db: TIBDatabase;
begin
  { get a link to the TfrxIBXDatabase.Database }
  db := TfrxIBXDatabase(Component).Database;
```

```
  { create a standard OpenDialog }
  with TOpenDialog.Create(nil) do
  begin
    InitialDir := GetCurrentDir;
    { we are interested in *.gdb files }
    Filter := frxResources.Get('ftDB') + ' (*.gdb)|*.gdb|' +
frxResources.Get('ftAllFiles') + ' (*.*)|*.*';
    Result := Execute;
    if Result then
    begin
      SaveConnected := db.Connected;
      db.Connected := False;
      { if a dialogue is completed successfully, assign a new DB name }
      db.DatabaseName := FileName;
      db.Connected := SaveConnected;
    end;
    Free;
  end;
end;


{ TfrxTableNameProperty }

function TfrxTableNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { the property represents the list of values }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxTableNameProperty.GetValues;
var
  t: TIBTable;
begin
  inherited;
  { get a link to the TIBTable  component }
  t := TfrxIBXTable(Component).Table;
  { fill the list of tables available }
  if t.Database <> nil then
    t.DataBase.GetTableNames(Values, False);
end;


{ TfrxIndexProperty }

function TfrxIndexNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { the property represents the list of values }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxIndexNameProperty.GetValues;
var
  i: Integer;
begin
  inherited;
  try
    { get a link to the TIBTable component }
    with TfrxIBXTable(Component).Table do
      if (TableName <> '') and (IndexDefs <> nil) then
```

```
      begin
        { update indexes }
        IndexDefs.Update;
        { fill the list of indexes available }
        for i := 0 to IndexDefs.Count - 1 do
          if IndexDefs[i].Name <> '' then
            Values.Add(IndexDefs[i].Name);
      end;
  except
  end;
end;


initialization
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXDataBase,
'DatabaseName', TfrxDataBaseNameProperty);
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXTable,
'TableName', TfrxTableNameProperty);
  frxPropertyEditors.Register(TypeInfo(String), TfrxIBXTable,
'IndexName', TfrxIndexNameProperty);

end.
```

Connecting custom functions to the report

FastReport has a quite large number of standard functions, which can be used in a report. There also is a possibility to connect your own functions. Connection of functions is performed via the "FastScript" script library's interface, which is included in FastReport (to know more about FastScript, refer to the manual of this library).

Let us examine an example of how a procedure and/or a function can be connected. There are two basic ways to perform it: either by using "FastScript" interface, or with help of the "TfrxReport" component's methods. Quantity and type of the connected function's parameters can be different. One cannot transfer parameters of the "Set" and "Record" type, as they are not supported in FastScript. It is required to transfer such parameters as simpler types, for example, to transfer the TRect as X0, Y0, X1, Y1: Integer. See more about process of adding functions with different parameters in the FastScript documentation.

Way 1:

```
uses fs_iinterpreter;

function TForm1.MyFunc(s: String; i: Integer): Boolean;
begin
// necessary logic
end;

procedure TForm1.MyProc(s: String);
begin
// necessary logic
end;
```

```
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const
MethodName: String; var Params: Variant): Variant;
begin
  if MethodName = 'MYFUNC' then
    Result := MyFunc(Params[0], Params[1])
  else if MethodName = 'MYPROC' then
    MyProc(Params[0]);
end;

  fsGlobalUnit.AddMethod('function MyFunc(s: String; i: Integer):
Boolean', CallMethod);
  fsGlobalUnit.AddMethod('procedure MyProc(s: String)', CallMethod);
```

First of all, you should add functions' descriptions via calling fsGlobalUnit.AddMethod. The first parameter is the syntax's description; the second one is a link to the function's handler. Next step would be creation of a handler of the "TfsCallMethodEvent" type and realization of functions' call in it. The handler is the function of a class:

```
TfsCallMethodEvent = function (Instance: TObject; ClassType: TClass;
const MethodName: String; var Params: Variant): Variant of object;
```

We do not need the "Instance," and "ClassType" parameters yet. "MethodName" is the name of a function in the upper case; Params is the array of parameters.

Way 2:

```
function TForm1.MyFunc(s: String; i: Integer): Boolean;
begin
// necessary logic
end;

procedure TForm1.MyProc(s: String);
begin
// necessary logic
end;

function TForm1.frxReport1UserFunction(const MethodName: String;
var Params: Variant): Variant;
begin
  if MethodName = 'MYFUNC' then
    Result := MyFunc(Params[0], Params[1])
  else if MethodName = 'MYPROC' then
    MyProc(Params[0]);
end;

  frxReport1.AddFunction('function MyFunc(s: String; i: Integer):
Boolean');
  frxReport1.AddFunction('procedure MyProc(s: String)');
```

This way is a little easier. Functions' descriptions are now added via the "TfrxReport.AddFunction" method with a single parameter. Functions' call is realized in the TfrxReport.OnUserFunction event's handler.

Both ways of connection are equivalent. The connected function can be used in the report's script; furthermore, one can refer to it from the objects of the "TfrxMemoView" type. The function is also displayed in the "Data tree" window. In this window functions are split into categories, and thus when you select any function, the hint about this function appears at the bottom of the window.

Let us modify the code of our examples to register functions in a separate category, and display the function's description:

the first way:

```
  fsGlobalUnit.AddMethod('function MyFunc(s: String; i: Integer):
Boolean', CallMethod, 'My functions', 'The MyFunc function always
returns True');
  fsGlobalUnit.AddMethod('procedure MyProc(s: String)', CallMethod, 'My
functions', 'The MyProc procedure does not do anything');
```

the second way:

```
  frxReport1.AddFunction('function MyFunc(s: String; i: Integer):
Boolean', 'My functions', 'The MyFunc function always returns True');
  frxReport1.AddFunction('procedure MyProc(s: String)', 'My functions',
'The MyProc procedure does not do anything');
```

If you want to register functions in one of the standard categories, use the following categories' names:

'ctString' – string function;
'ctDate' - date/time functions;
'ctConv' - conversion functions;
'ctFormat' - formatting;
'ctMath' - mathematical functions;
'ctOther' - other functions.

If a blank category's name is specified, the function is placed to the root of the functions' tree.

If you are going to connect a large number of functions, it is recommended to carry out all the logic into a separate unit. Here is an example of such unit:

```
unit myfunctions;

interface

implementation

uses SysUtils, Classes, fs_iinterpreter;

type
  TFunctions = class(TObject)
  private
    function CallMethod(Instance: TObject; ClassType: TClass; const
```

```
MethodName: String; var Params: Variant): Variant;
  public
    constructor Create;
    destructor Destroy; override;
  end;

var
  Functions: TFunctions;


function MyFunc(s: String; i: Integer): Boolean;
begin
// necessary logic
end;

procedure MyProc(s: String);
begin
// necessary logic
end;


{ TFunctions }

constructor TFunctions.Create;
begin
  with fsGlobalUnit do
  begin
    AddedBy := Self;
    AddMethod('function MyFunc(s: String; i: Integer): Boolean',
CallMethod, 'My functions', 'The MyFunc function always returns True');
    AddMethod('procedure MyProc(s: String)', CallMethod, 'My functions',
'The MyProc procedure does not do anything'');
    AddedBy := nil;
  end;
end;

destructor TFunctions.Destroy;
begin
  if fsGlobalUnit <> nil then
    fsGlobalUnit.RemoveItems(Self);
  inherited;
end;

function TFunctions.CallMethod(Instance: TObject; ClassType: TClass;
const MethodName: String; var Params: Variant): Variant;
begin
  if MethodName = 'MYFUNC' then
    Result := MyFunc(Params[0], Params[1])
  else if MethodName = 'MYPROC' then
    MyProc(Params[0]);
end;

initialization
  Functions := TFunctions.Create;

finalization
  Functions.Free;

end.
```

Writing custom wizards

You can extend functionality of FastReport with the help of so-called wizards. FastReport, for example, contains the standard "Report Wizard," which is called from the "File|New…" menu.

There are two types of wizards supported in FastReport. The first type includes the wizards already mentioned, called from the "File|New…" menu. The second one includes wizards, which can be called from the "Wizards" toolbar.

The basic class for any wizard is "TfrxCustomWizard," defined in the "frxClass" file.

```
  TfrxCustomWizard = class(TComponent)
  public
    constructor Create(AOwner: TComponent); override;
    class function GetDescription: String; virtual; abstract;
    function Execute: Boolean; virtual; abstract;
    property Designer: TfrxCustomDesigner read FDesigner;
    property Report: TfrxReport read FReport;
  end;
```

To write your own wizard, it is necessary to inherit from this class and override at least the "GetDescription" and "Execute" methods. The first one returns the wizard's name; the second one is called when running the wizard; it must return "True," if the wizard finished working successfully and brought any changes to the report. During the wizard's working, you can call methods and properties of the designer and the report properly via the "Designer" and "Report" properties.

Registration and deleting of the wizard is performed via the procedures described in the "frxDsgnIntf" file:

```
frxWizards.Register(ClassRef: TfrxWizardClass; ButtonBmp: TBitmap;
IsToolbarWizard: Boolean = False);
frxWizards.Unregister(ClassRef: TfrxWizardClass);
```

At registration, one enters the name of the wizard's class, its picture, and specifies if the wizard is placed in the "Wizards" toolbar. If the wizard should be placed in the toolbar, the ButtonBmp size must be either 16x16 pixels, or 22x22 pixels otherwise.

Let us examine a primitive wizard, which is being registered in the "File|New..." menu, and then adds a new page to the report.

```
uses frxClass, frxDsgnIntf;

type
  TfrxMyWizard = class(TfrxCustomWizard)
  public
```

```
    class function GetDescription: String; override;
    function Execute: Boolean; override;
  end;

class function TfrxMyWizard.GetDescription: String;
begin
  Result := 'My Wizard';
end;

function TfrxMyWizard.Execute: Boolean;
var
  Page: TfrxReportPage;
begin
  { lock any drawings in the designer }
  Designer.Lock;

  { create a new page in the report }
  Page := TfrxReportPage.Create(Report);
  { create a unique name for the page }
  Page.CreateUniqueName;
  { set sizes and orientation of paper by default }
  Page.SetDefaults;

  { update report's pages and switch the focus to page the last added }
  Designer.ReloadPages(Report.PagesCount - 1);
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  { load a picture from a resource; of course, you should place it there
first }
  Bmp.LoadFromResourceName(hInstance, 'frxMyWizard');
  frxWizards.Register(TfrxMyWizard, Bmp);

finalization
  frxWizards.Unregister(TfrxMyWizard);
  Bmp.Free;

end.
```