

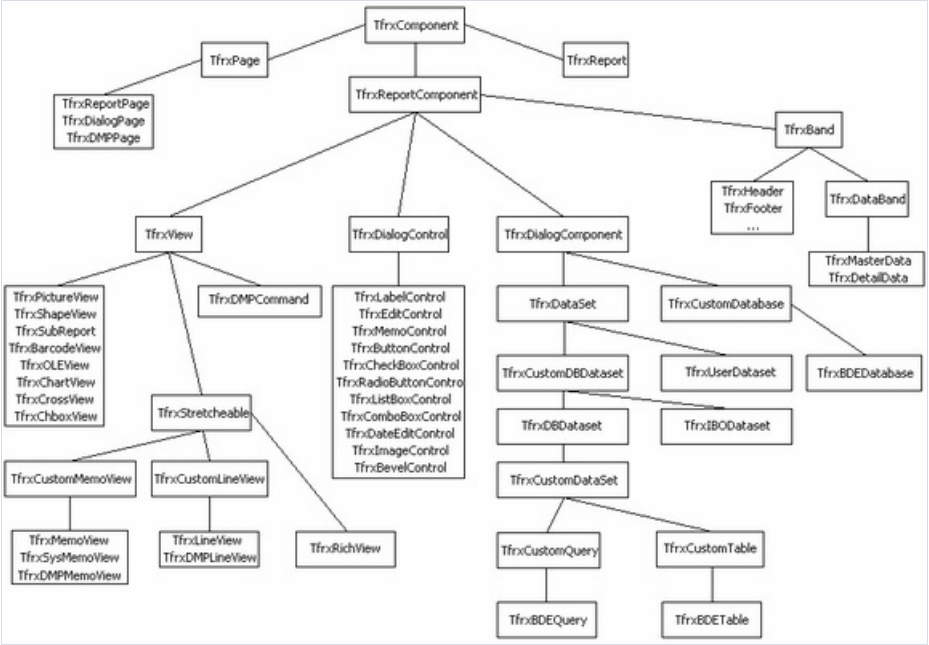
Быстрые отчеты

Руководство разработчика FastReport FMX

Версия 2026.2

© 2008-2026 ООО Быстрые отчеты

Иерархия классов FastReport



Базовым классом для всех компонентов FastReport является класс `TfrxComponent`. Объекты этого типа имеют координаты, размеры, шрифт, признак видимости, а также список подчиненных объектов. Класс также содержит методы для сохранения/восстановления состояния объекта в поток.

```
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); 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` - очищает содержимое объекта и удаляет все его дочерние объекты.
- `CreateUniqueName` - создает уникальное имя для объекта, который помещен в отчет.
- `LoadFromStream` - считывает содержимое объекта и всех его дочерних объектов из потока.
- `SaveToStream` - сохраняет объект в поток.
- `SetBounds` - устанавливает координаты и размеры объекта.
- `FindObject` - ищет объект с заданным именем среди дочерних объектов.
- `GetDescription` - возвращает описание объекта.

Следующие методы вызываются при изменении соответствующих свойств. Вы можете перекрыть их, если вам нужна дополнительная обработка:

- `SetParent`
- `SetLeft`

- `SetTop`
- `SetWidth`
- `SetHeight`
- `SetFont`
- `SetParentFont`
- `SetVisible`
- `FontChanged`

В классе `TfrxComponent` определены следующие свойства:

- `Objects` - список дочерних объектов;
- `AllObjects` - список всех подчиненных объектов;
- `Parent` - ссылка на родительский объект;
- `Page` - ссылка на страницу отчета, которой принадлежит объект;
- `Report` - ссылка на отчет, которому принадлежит объект;
- `IsDesigning` - истина, если запущен дизайнер;
- `IsLoading` - истина, если объект в процессе загрузки из потока;
- `IsPrinting` - истина, если объект печатается на принтере;
- `BaseName` - базовое имя объекта. Это значение используется в методе `CreateUniqueName`, который добавляет к базовому имени первую свободную цифру;
- `Left` - координата X объекта (относительно родителя);
- `Top` - координата Y объекта (относительно родителя);
- `Width` - ширина объекта;
- `Height` - высота объекта;
- `AbsLeft` - абсолютная координата X объекта;
- `AbsTop` - абсолютная координата Y объекта;
- `Font` - шрифт объекта;
- `ParentFont` - если истина, то использовать установки шрифта родительского объекта;
- `Restrictions` - набор флагов, запрещающих те или иные действия над объектом;
- `Visible` - видимость объекта.

Следующий основной класс - `TfrxReportComponent`. Объекты этого типа могут быть помещены в отчет. Класс содержит метод `Draw` для отрисовки объекта, а также методы `BeforePrint` / `GetData` / `AfterPrint`, которые вызываются при запуске отчета.

```
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;
```

- `Draw` - вызывается при отрисовке объекта. Параметры: Canvas - холст; Scale - масштаб по осям X,Y; Offset - смещение относительно начала холста;
- `BeforePrint` - вызывается перед обработкой объекта (в процессе построения отчета). Этот метод должен сохранить состояние объекта;
- `GetData` - метод должен загрузить данные в объект;
- `AfterPrint` - вызывается после того, как объект обработан. Метод должен восстановить состояние объекта.

Класс `TfrxDialogComponent` является базовым для написания невизуальных компонентов, которые могут быть помещены на диалоговую форму в отчете.

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

Класс `TfrxDialogControl` является базовым для написания элементов управления, которые могут быть помещены на диалоговую форму в отчете. Класс содержит большое количество свойств и событий, общих для большинства элементов управления.

```
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;
```

При написании своего элемента управления вам надо наследоваться от этого класса, перенести нужные свойства в раздел published и добавить новые свойства, специфичные для вашего элемента управления. Написание собственных элементов управления будет рассмотрено более подробно в соответствующей главе.

Класс `TfrxView` является базовым для большинства компонентов, которые могут быть размещены на странице отчета. Объект этого типа имеет рамку и заливку, может подключаться к источнику данных. Практически все стандартные объекты FastReport наследуются от данного класса.

```
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;
```

В классе определены следующие методы:

- `BeginDraw` - метод вызывается из метода `Draw` и вычисляет целочисленные координаты и размеры области отрисовки. Вычисленные значения помещаются в переменные FX, FY, FX1, FY1, FDX, FDY. Также вычисляется толщина рамки (помещается в FFrameWidth);
- `DrawBackground` - отрисовка фона объекта;
- `DrawFrame` - отрисовка рамки объекта;
- `DrawLine` - вспомогательный метод, рисующий линию с заданными координатами и толщиной;
- `IsDataField` - возвращает True, если св-ва `DataSet`, `DataField` содержат непустые значения.

К следующим свойствам можно обращаться после вызова метода `BeginDraw`:

- `FX`, `FY`, `FX1`, `FY1`, `FDX`, `FDY`, `FFrameWidth` - координаты, размеры и толщина рамки объекта, вычисленные с учетом масштабирования и смещения;
- `FScaleX`, `FScaleY` - масштаб, копия параметров ScaleX, ScaleY из метода `Draw`;
- `FCanvas` - холст, копия параметра Canvas из метода `Draw`.

В классе определены следующие свойства, общие для большинства объектов отчета:

- `BrushStyle` - стиль заливки объекта;
- `Color` - цвет заливки объекта;
- `DataField` - имя поля данных, к которому подключен объект;

- `DataSet` - источник данных;
- `Frame` - рамка объекта;
- `Align` - выравнивание объекта относительно его родителя;
- `Printable` - определяет, надо ли печатать данный объект на принтере;
- `ShiftMode` - режим смещения объекта в случае, когда над данным объектом расположен растягиваемый объект;
- `TagStr` - вспомогательное поле для хранения различной информации.

Класс `TfrxStretcheable` является базовым для написания компонентов, которые могут менять свою высоту в зависимости от находящихся в них данных.

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

Объекты данного класса могут не только растягиваться. Они также могут быть "разорваны" на части в случае, когда объект не помещается на страницу целиком. При этом объект выводится по частям до тех пор, пока все его данные не будут выведены.

В классе определены следующие методы:

- `CalcHeight` - должен вычислить и вернуть высоту объекта с учетом хранящихся в нем данных;
- `InitPart` - вызывается перед началом разбиения объекта;
- `DrawPart` - должен отрисовать очередную порцию данных, которые помещаются в объекте. Возвращаемое значение - это величина неиспользованного пространства, на котором не удалось вывести данные.

Написание собственных компонентов отчета

FastReport имеет большое количество компонентов, которые можно поместить на страницу отчета. Это текст, рисунок, линия, геометрическая фигура, OLE, Rich, штрихкод, диаграмма и пр. Вы также можете написать собственный компонент и подключить его к FastReport.

В FastReport определено несколько классов, от которых наследуются компоненты. Подробнее об этих классах можно прочитать в главе "Иерархия классов". Для нас наибольший интерес представляет класс `TfrxView`, т.к. именно от него наследуется большинство компонентов отчета.

Нам придется реализовать, как минимум, метод `Draw`, определенный в базовом классе `TfrxReportComponent`.

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

Этот метод вызывается при любой отрисовке компонента, в дизайнере, в окне предварительного просмотра или на принтере. `TfrxView` перекрывает этот метод для отрисовки фона и рамки объекта. Метод должен нарисовать содержимое объекта на поверхности рисования Canvas. Координаты и размеры объекта хранятся в свойствах `AbsLeft`, `AbsTop`, `Width`, `Height` соответственно.

Параметры ScaleX, ScaleY определяют масштабирование объекта по осям X и Y соответственно. Эти параметры равны 1 при 100% масштабе и могут меняться, если пользователь меняет масштаб изображения в дизайнере или окне предварительного просмотра. Параметры OffsetX и OffsetY указывают смещение координат по осям X и Y. Таким образом, координата левого верхнего угла с учетом этих параметров будет равна

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

Чтобы упростить работу с координатами, в классе `TfrxView` определен метод `BeginDraw` с такими же параметрами, как и метод `Draw`.

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

Его следует вызывать первой строкой в методе `Draw`. Этот метод выполняет преобразование координат в целочисленные значения `FX`, `FY`, `FX1`, `FY1`, `FDX`, `FDY`, `FFrameWidth`, которые затем можно использовать в методах TCanvas. Также этот метод копирует значения Canvas, ScaleX, ScaleY в переменные `FCanvas`, `FScaleX`, `FScaleY`, к которым можно будет обращаться из любого метода класса.

Также в классе `TfrxView` определены два метода для отрисовки фона и рамки объекта.

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

Перед вызовом этих методов надо вызвать метод `BeginDraw`.

Рассмотрим создание компонента, который отображает стрелку.

```
type
  TfrxArrowView = class(TfrxView)
  public
    { нам нужно перекрыть только два метода }
    procedure Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended); override;
    class function GetDescription: String; override;
  published
    { выносим нужные свойства в секцию published }
    property BrushStyle;
    property Color;
    property Frame;
  end;

class function TfrxArrowView.GetDescription: String;
begin
  { описание компонента будет выводиться рядом с его иконкой на панели инструментов }
  Result := 'Arrow object';
end;

procedure TfrxArrowView.Draw(Canvas: TCanvas; ScaleX, ScaleY, OffsetX, OffsetY: Extended);
begin
  { вызываем этот метод, чтобы выполнить преобразование координат }
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);
  with Canvas do
  begin
    { настраиваем цвета }
    Brush.Color := Color;
    Brush.Style := BrushStyle;
    Pen.Width := FFrameWidth;
    Pen.Color := Frame.Color;
    { рисуем стрелку }
    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;

{ регистрация }

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(hInstance, 'frxArrowView');
  frxObjects.RegisterObject(TfrxArrowView, Bmp);

finalization
  { удаляем компонент из списка доступных }
  frxObjects.Unregister(TfrxArrowView);
  Bmp.Free;

end.
```

Для создания компонента, который отображает какие-либо данные из БД, надо вынести свойства `DataSet`, `DataField` в раздел published и перекрыть метод `GetData`. Рассмотрим это на примере стандартного компонента `TfrxCheckBoxView`.

Компонент может подключаться к полю БД с помощью свойств `DataSet`, `DataField`, которые присутствуют

в базовом классе `TfrxView`. Кроме того, у компонента есть свойство `Expression`, в которое можно поместить выражение. Оно будет вычислено и результат будет помещен в свойство `Checked`. Компонент отображает крестик, если свойство `Checked` равно True. Ниже приведен исходный текст этого компонента (только те части, которые нас интересуют).

```
  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;
```

Написание собственных элементов управления

FastReport содержит набор элементов управления, которые могут быть помещены на диалоговую форму внутри отчета. Это следующие элементы:

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

Назначение этих элементов управления соответствуют стандартным контролам из палитры компонент Delphi. Если вас не устраивает стандартная функциональность, вы можете написать свой элемент управления и использовать его в своих отчетах.

Базовым классом для всех элементов управления является класс `TfrxDialogControl`, описанный в файле frxClass:

```
  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;
```

Для создания своего элемента управления необходимо наследоваться от этого класса и перекрыть, как минимум, конструктор и метод `GetDescription`. В конструкторе необходимо создать элемент управления и инициализировать его с помощью метода `InitControl`. Метод `GetDescription` должен возвращать описание элемента управления. Как видите из описания класса `TfrxDialogControl`, он уже содержит большое количество свойств и методов в разделе public. Вам необходимо перенести нужные описания в раздел published своего элемента управления, а также создать новые свойства, характерные для вашего элемента.

Регистрация и удаление элемента управления выполняется с помощью методов глобального объекта frxObjects, объявленного в файле frxDsgnIntf:

```
frxObjects.RegisterObject(ClassRef: TfrxComponentClass; ButtonBmp: TBitmap);
frxObjects.Unregister(ClassRef: TfrxComponentClass);
```

При регистрации указывается имя класса контрола и его картинку. Размеры ButtonBmp должны быть 16x16 точек.

Рассмотрим пример элемента управления, который реализует упрощенную функциональность стандартного делфийского контрола `TBitBtn`.

```
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
    { добавляем новые свойства }
    property Kind: TBitBtnKind read GetKind write SetKind default bkCustom;
    { эти свойства уже объявлены в родительском классе }
    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
  { конструктор по умолчанию }
  inherited;
  { создаем нужный элемент управления }
  FButton := TBitBtn.Create(nil);
  FButton.Caption := 'BitBtn';
  { инициализируем его }
  InitControl(FButton);
  { такие размеры он будет иметь по умолчанию }
  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;
  { загружаем картинку из ресурса - естественно, вы ее должны предварительно туда поместить }
  Bmp.LoadFromResourceName(hInstance, 'frxBitBtnControl');
  frxObjects.RegisterObject(TfrxBitBtnControl, Bmp);

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

end.
```

Описание обработчиков событий

Как быть, если необходимо определить новый обработчик события, которого нет в базовом классе? Рассмотрим это на примере элемента управления `TfrxEditControl`:

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

constructor TfrxEditControl.Create(AOwner: TComponent);
begin
  ...
  { подключаем наш обработчик }
  FEdit.OnChange := DoOnChange;
  InitControl(FEdit);
  ...
end;

procedure TfrxEditControl.DoOnChange(Sender: TObject);
begin
  { вызываем обработчик события }
  if Report <> nil then
    Report.DoNotifyEvent(Sender, FOnChange);
end;
```

Здесь необходимо отметить следующий момент. Обработчик события в FastReport - это процедура, объявленная в скрипте отчета. Ссылкой на такой обработчик будет строка, содержащая его имя. Поэтому, например, в отличие от делфийского типа `TNotifyEvent`, который является адресом метода, тип обработчика в FastReport - строковый (тип `TfrxNotifyEvent` объявлен как String[63]).

Регистрация компонента в скриптовой системе

Чтобы к нашему компоненту можно было обращаться из скрипта, необходимо зарегистрировать класс, его свойства и методы в скриптовой системе. Код регистрации по соглашению, принятому в FastReport, можно разместить в файле, имеющем такое же имя, как и файл с кодом самого компонента плюс суффикс RTTI (например, в нашем случае frxBitBtnRTTI.pas). Более подробно о регистрации классов, их методов и свойств можно прочитать в документации по скриптовой библиотеке FastScript.

```
uses fs_iinterpreter, frxBitBtn, frxClassRTTI;

type
  TFunctions = class(TfsRTTIModule)
  public
    constructor Create(AScript: TfsScript); override;
  end;

constructor TFunctions.Create(AScript: TfsScript);
begin
  inherited Create(AScript);
  with AScript do
  begin
    { регистрируем класс и указываем, кто является его родителем }
    AddClass(TfrxBitBtnControl, 'TfrxDialogControl');
    { если в вашем модуле несколько элементов управления, их можно зарегистрировать тут же }
    { напр. AddClass(TfrxAnotherControl, 'TfrxDialogControl'); }
  end;
end;

initialization
  fsRTTIModules.Add(TFunctions);

end.
```

Написание редактора компонента

По умолчанию редактор любого элемента управления (его можно вызвать из контекстного меню элемента или по двойному щелчку мыши) создает пустой обработчик события `OnClick`. Это поведение можно переопределить, написав свой редактор. Кроме того, редактор позволяет позволяет добавить свои пункты в контекстное меню компонента.

Базовый класс для всех редакторов описан в файле frxDsgnIntf:

```
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;
```

Если ваш редактор не создает собственных пунктов в контекстном меню, то вам необходимо перекрыть два метода - `Edit` и `HasEditor`. Первый метод должен выполнять необходимые действия (например, показывать диалоговое окно) и возвращать True, если состояние компонента было изменено. Метод `HasEditor` должен просто возвращать True. Если он вернет False, или вы не будете перекрывать этот метод - редактор вызываться не будет. Это нужно, если ваш компонент не имеет редактора и вы хотите просто добавить пункты в контекстное меню компонента.

Если редактор добавляет пункты в контекстное меню, то вам необходимо перекрывать методы `GetMenuItems` (в этом методе с помощью вызовов функции `AddItem` вы создаете меню) и `Execute` (этот метод вызывается, когда вы выбрали один из своих пунктов в меню компонента, и здесь надо описать реакцию на выбранный пункт меню).

Регистрация редактора выполняется с помощью процедуры, описанной в файле frxDsgnIntf:

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

Первый параметр - это имя класса, для которого нужно создать редактор. Второй параметр - имя класса самого редактора.

Рассмотрим простой редактор для нашего элемента управления, который будет выводить окошко с именем нашего элемента и добавлять два пункта "Enabled" и "Visible" в контекстное меню элемента (при выборе пунктов будут меняться свойства `Enabled` и `Visible` элемента). Код редактора по соглашению, принятому в FastReport, можно разместить в файле, имеющем такое же имя, как и файл с кодом самого компонента плюс суффикс Editor (например, в нашем случае frxBitBtnEditor.pas).

```
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;
  { св-во Component - это редактируемый компонент. В нашем случае это TfrxBitBtnControl }
  c := TfrxBitBtnControl(Component);
  ShowMessage('Это ' + 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);
  { параметры метода AddItem: имя пункта меню, его тэг и состояние Checked/Unchecked }
  AddItem('Enabled', 1, c.Enabled);
  AddItem('Visible', 2, c.Visible);
end;

initialization
  frxComponentEditors.Register(TfrxBitBtnControl, TfrxBitBtnEditor);

end.
```

Написание редактора свойства

Когда вы выделяете компонент в дизайнере, его свойства отображаются в инспекторе объектов. Вы можете создать свой редактор для любого свойства любого компонента. Примером этого может служить стандартный редактор свойства `Font`: если выделить это свойство, то в правой части строки редактирования появляется кнопка ..., нажав на которую, вы вызываете стандартное диалоговое окно "Свойства шрифта". Еще один пример - редактор свойства `Color`. Он показывает в выпадающем списке названия стандартных цветов и рядом - образец цвета.

Базовый класс для всех редакторов свойств описан в файле frxDsgnIntf:

```
  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 GetName: String; 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;
```

Так же вы можете наследоваться от любого из нижеперечисленных классов, которые уже реализуют некоторую базовую функциональность для работы со свойствами соответствующих типов:

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

В классе определено несколько свойств:

- `Component` - ссылка на родительский компонент (не на само свойство!), которому принадлежит данное свойство;
- `frComponent` - то же, но приведенное к типу `TfrxComponent` (для удобства использования в некоторых случаях);
- `Designer` - ссылка на дизайнер отчета;
- `ItemHeight` - высота ячейки, в которой отображается свойство. Может быть полезно в методах OnDrawXXX;
- `PropInfo` - ссылка на структуру PPropInfo, содержащую информацию о редактируемом свойстве;
- `Value` - значение свойства в строковом виде;
- `Values` - список значений. Это свойство должно быть заполнено в методе `GetValue`, если определен атрибут paValueList (см. ниже).

Следующие методы являются служебными и позволяют получить или установить значение редактируемого свойства.

```
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);
```

Вы должны использовать те методы, которые соответствуют типу свойства. Так, если свойство типа Integer, то используйте методы `GetOrdValue` и `SetOrdValue`. Эти же методы используются для работы со свойством типа `TObject`, поскольку такое свойство содержит 32-х битный адрес объекта. В этом случае необходимо делать приведение типа, например:

```
MyFont := TFont(GetOrdValue);
```

Для создания своего редактора необходимо наследоваться от базового класса и перекрыть один или несколько методов, объявленных в разделе public (это зависит от типа свойства и функциональности, которую вы хотите реализовать). Одним из методов, который вам наверняка придется перекрыть, является метод `GetAttributes`. Этот метод должен вернуть набор атрибутов свойства. Атрибуты определены следующим образом:

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

Назначение атрибутов следующее:

- paValueList - свойство представляет собой выпадающий список значений. Пример - свойство `Color`. Если этот атрибут присутствует, необходимо перекрыть метод `GetValues`;
- paSortList - сортирует элементы списка, применяется совместно с paValueList;

- paDialog - свойство имеет редактор. Если этот атрибут присутствует, в правой части строки редактирования выводится кнопка ... . При ее нажатии вызывается метод Edit;
- paMultiSelect - разрешить редактирование данного свойства у нескольких одновременно выбранных объектов. Некоторые свойства, например, `Name`, не имеют этого атрибута;
- paSubProperties - свойство является объектом типа `TPersistent` и имеет свои свойства, которые также надо отобразить. Пример - свойство `Font`;
- paReadOnly - нельзя изменять значение в строке редактирования. Некоторые свойства, являющиеся типами Class, Set имеют этот атрибут;
- paOwnerDraw - отрисовка значения свойства выполняется с помощью метода `OnDrawItem`. Если определен атрибут paValueList, то отрисовка выпадающего списка выполняется с помощью метода `OnDrawLBItem`.

Метод `Edit` вызывается в двух случаях: если выделить свойство и дважды щелкнуть на его значении; либо, если свойство имеет атрибут paDialog - при нажатии на кнопку ... . Метод должен вернуть True, если состояние свойства было изменено.

Метод `GetName` в большинстве случаев перекрывать не надо. Он возвращает имя редактируемого свойства.

Метод `GetValue` должен вернуть значение свойства в виде строки (она будет показана в инспекторе объектов). Если вы наследуетесь от базового класса `TfrxPropertyEditor`, то перекрывать метод надо обязательно.

Метод `SetValue` должен установить значение свойства, переданное в виде строки. Если вы наследуетесь от базового класса `TfrxPropertyEditor`, то перекрывать метод надо обязательно.

Метод `GetValues` надо перекрывать в случае, если вы определили атрибут paValueList. Данный метод должен заполнить значениями свойство `Values`.

Следующие три метода позволяют выполнить ручную отрисовку значения свойства (подобным образом работает редактор свойства Color). Эти методы вызываются, если вы определили атрибут paOwnerDraw.

Метод `OnDrawItem` вызывается при отрисовке значения свойства в инспекторе объектов (когда свойство не является выбранным - иначе его значение просто отображается в строке редактирования). Например, редактор свойства Color выводит слева от значения свойства квадратик, закрашенный цветом, соответствующим значению.

Метод `GetExtraLBSize` вызывается в случае, если вы определили атрибут paValueList. Метод возвращает число пикселей, на которое надо увеличить ширину выпадающего списка, чтобы уместить выводимое изображение. По умолчанию этот метод возвращает значение, соответствующее высоте ячейки для отрисовки свойства. Если вам надо выводить картинку, ширина которой больше, чем высота, то данный метод надо перекрыть.

Метод `OnDrawLBItem` вызывается при отрисовке строки в выпадающем списке, если вы определили атрибут paValueList. По сути, этот метод является обработчиком события TListBox.OnDrawItem и имеет тот же набор параметров.

Регистрация редактора свойства выполняется с помощью процедуры, описанной в файле frxDsgnIntf:

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

- PropertyType - информация о типе свойства, передается с помощью системной функции `TypeInfo`,

например TypeInfo(String);

- ComponentClass - имя компонента, свойство которого вы хотите редактировать (может быть nil);
- PropertyName - имя свойства, которое вы хотите редактировать (может быть пустой строкой);
- EditorClass - имя редактора свойства.

Обязательно указывать лишь параметр PropertyType. Параметры ComponentClass и/или PropertyName могут быть пустыми. Это позволяет регистрировать редактор на любое свойство типа PropertyType, либо на любое свойство конкретного компонента ComponentClass и его наследников, либо на конкретное свойство PropertyName конкретного компонента (или любого компонента, если параметр ComponentClass пустой).

Рассмотрим три примера редакторов свойств. Код редактора по соглашению, принятому в FastReport, можно разместить в файле, имеющем такое же имя, как и файл с кодом самого компонента плюс суффикс Editor.

```
{ редактор свойства TFont. Показывает кнопку редактора (...) }
{ наследуемся от ClassProperty }

type
  TfrxFontProperty = class(TfrxClassProperty)
  public
    function Edit: Boolean; override;
    function GetAttributes: TfrxPropertyAttributes; override;
  end;

function TfrxFontProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { свойство имеет вложенные свойства и редактор. Его нельзя редактировать вручную }
  Result := [paMultiSelect, paDialog, paSubProperties, paReadOnly];
end;

function TfrxFontProperty.Edit: Boolean;
var
  FontDialog: TFontDialog;
begin
  { создаем стандартный диалог }
  FontDialog := TFontDialog.Create(Application);
  try
    { берем значение свойства }
    FontDialog.Font := TFont(GetOrdValue);
    FontDialog.Options := FontDialog.Options + [fdForceFontExist];
    { показываем диалог }
    Result := FontDialog.Execute;
    { присваиваем новое значение }
    if Result then
      SetOrdValue(Integer(FontDialog.Font));
  finally
    FontDialog.Free;
  end;
end;

{ регистрация }

frxPropertyEditors.Register(TypeInfo(TFont), nil, '', TfrxFontProperty);
```

```
{ редактор свойства TFont.Name. Показывает выпадающий список доступных шрифтов }
{ наследуемся от StringProperty, т.к. свойство строкового типа }

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;

{ регистрация }

frxPropertyEditors.Register(TypeInfo(String), TFont, 'Name', TfrxFontNameProperty);
```

```
{ редактор свойства TPen.Style. Показывает картинку - образец выбранного стиля }

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;

{ метод рисует толстую горизонтальную линию с выбранным стилем }

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;

{ отрисовка выпадающего списка }

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;

{ отрисовка значения свойства }

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;

{ возвращаем ширину картинки }

function TfrxPenStyleProperty.GetExtraLBSize: Integer;
begin
  Result := 36;
end;

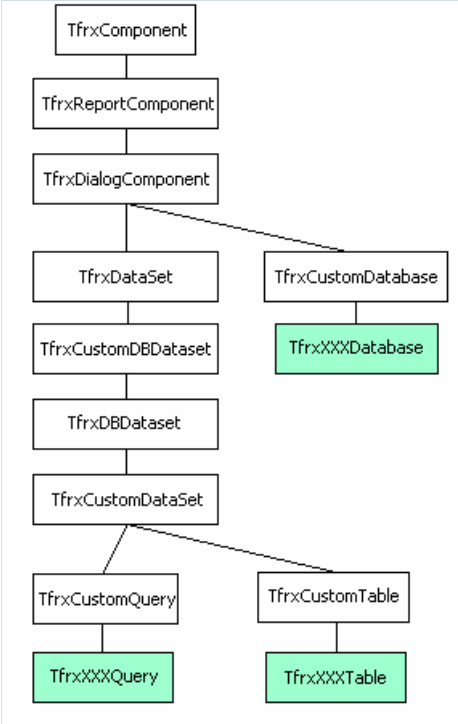
{ регистрация }

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

Написание собственных движков баз данных

FastReport позволяет строить отчеты не только на основе данных, определенных в приложении. Вы также можете определить свои собственные источники данных (подключения к БД, таблицы, запросы) прямо в отчете. В комплекте с FastReport идут движки для ADO, BDE, IBX, DBX, FIB. Вы можете создать собственный движок и подключить его к FastReport.

На рисунке показана иерархия классов, предназначенных для создания движков баз данных. Зеленым цветом выделены компоненты нового движка.



Как видно, стандартный набор компонентов движка БД включает в себя Database, Table и Query. Вы можете реализовать все эти компоненты, или только некоторые из них (например, многие БД не имеют компонента типа Table). Также вы можете реализовать компоненты, не входящие в стандартный набор (например, аналог StoredProc).

Рассмотрим подробнее базовые классы.

`TfrxDialogComponent` - базовый класс для всех невизуальных компонентов, которые могут быть помещены в отчет FastReport. В нем не определено каких-либо важных свойств и методов.

Класс `TfrxCustomDatabase` является базовым для написания компонент типа `TDatabase`.

```
TfrxCustomDatabase = class(TfrxDialogComponent)
protected
  procedure SetConnected(Value: Boolean); virtual;
  procedure SetDatabaseName(const Value: String); virtual;
  procedure SetLoginPrompt(Value: Boolean); virtual;
  procedure SetParams(Value: TStrings); virtual;
  function GetConnected: Boolean; virtual;
  function GetDatabaseName: String; virtual;
  function GetLoginPrompt: Boolean; virtual;
  function GetParams: TStrings; virtual;
public
  procedure SetLogin(const Login, Password: String); virtual;
  property Connected: Boolean read GetConnected write SetConnected default False;
  property DatabaseName: String read GetDatabaseName write SetDatabaseName;
  property LoginPrompt: Boolean read GetLoginPrompt write SetLoginPrompt default True;
  property Params: TStrings read GetParams write SetParams;
end;
```

В классе определены следующие свойства:

- `Connected` – является ли подключение к БД активным;
- `DatabaseName` – имя БД;
- `LoginPrompt` – надо ли запрашивать пароль при поключении к БД;
- `Params` – параметры подключения.

От данного класса наследуется компонент типа TDatabase. Для его реализации необходимо перекрыть все виртуальные методы и вынести нужные свойства в секцию published. Также необходимо добавить свойства, специфичные для вашего компонента.

Классы `TfrxDataset` , `TfrxCustomDBDataset` , `TfrxDBDataset` обеспечивают функции доступа к данным. Ядро FastReport использует эти компоненты для навигации и обращения к полям набора данных. В данном случае они являются частью общей иерархии и не представляют интереса.

`TfrxCustomDataSet` - базовый класс для компонентов БД, производных от `TDataSet` . От этого класса наследуются компоненты - аналоги Query, Table, StoredProc. По сути, класс представляет собой обертку над `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;
  property DBConnected: Boolean;
published
  property Filter: String;
  property Filtered: Boolean;
  property Master: TfrxDBDataSet;
end;
```

В классе определены следующие свойства:

- `DataSet` - ссылка на внутренний объект типа TDataSet;

- `Fields` - ссылка на DataSet.Fields;
- `Active` - активен ли набор данных;
- `DBConnected` – подключен ли набор данных к компоненту TfrxXXXDatabase;
- `Filter` - выражение для фильтрации;
- `Filtered` - активна ли фильтрация;
- `Master` – ссылка на источник данных, являющийся основным. Применяется для связей типа master-detail.
- `MasterFields` – список пар полей вида field1=field2. Применяется для связей типа master-detail.

`TfrxCustomTable` - базовый класс для компонентов БД типа Table. Класс является оберткой над компонентом типа Table.

```
TfrxCustomTable = class(TfrxCustomDataset)
protected
  function GetIndexFieldNames: String; virtual;
  function GetIndexName: String; virtual;
  function GetTableName: String; virtual;
  procedure SetIndexFieldNames(const Value: String); virtual;
  procedure SetIndexName(const Value: String); virtual;
  procedure SetTableName(const Value: String); virtual;
published
  property MasterFields;
  property TableName: String read GetTableName write SetTableName;
  property IndexName: String read GetIndexName write SetIndexName;
  property IndexFieldNames: String read GetIndexFieldNames write SetIndexFieldNames;
end;
```

В классе определены следующие свойства:

- `TableName` – имя таблицы;
- `IndexName` – имя индекса;
- `IndexFieldNames` – имена индексных полей.

От данного класса наследуется компонент типа Table. Для его реализации необходимо определить недостающие свойства, как правило, `Database`. Также необходимо перекрыть виртуальные методы из классов `TfrxCustomDataset`, `TfrxCustomTable`.

`TfrxCustomQuery` - базовый класс для компонентов БД типа Query. Класс является оберткой над компонентом типа Query.

```
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;
```

В классе определены свойства, общие для всех компонентов Query - `SQL` и `Params`. Т.к. разные компоненты Query имеют разную реализацию параметров (например, `TParams`, `TParameters`), свойство `Params` имеет тип `TfrxParams` и является оберткой над конкретным типом параметров.

В данном классе определены следующие методы:

- `SetSQL` - должен установить свойство `SQL` компонента типа Query;
- `GetSQL` - должен вернуть свойство `SQL` компонента типа Query;
- `UpdateParams` - должен скопировать значения параметров в компонент типа Query. Если компонент Query имеет параметры типа `TParams`, то копирование осуществляется стандартной процедурой `frxParamsToTParams`.

Рассмотрим создание движка БД на примере IBX. Полный исходный текст движка можно найти в каталоге SOURCE\IBX. Здесь мы будем приводить выдержки из исходного текста с комментариями и некоторыми поправками.

Компоненты IBX, вокруг которых мы будем строить обертку - `TIBDatabase`, `TIBTable`, `TIBQuery`. Соответственно, наши компоненты будут называться `TfrxIBXDatabase`, `TfrxIBXTable`, `TfrxIBXQuery`.

Компонент для палитры Delphi

Первый компонент, который мы должны создать - `TfrxIBXComponents`, он будет помещен в палитру компонент FastReport при регистрации движка в среде Delphi. При помещении этого компонента в проект Delphi автоматически добавит ссылку на модуль нашего движка в список uses. На этот компонент удобно возложить еще одну задачу - определить у него свойство `DefaultDatabase`, которое ссылается на уже имеющееся в проекте подключение к БД. По умолчанию все компоненты `TfrxIBXTable` и `TfrxIBXQuery` будут ссылаться на это подключение.

Компонент необходимо наследовать от класса `TfrxDBComponents`:

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

Единственная функция должна возвращать описание, например 'IBX Components'. Реализация компонента `TfrxIBXComponents` следующая:

```
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;
```

Мы определяем глобальную переменную IBXComponents, которая будет ссылаться на экземпляр компонента

`TfrxIBXComponents`. На случай, если вы несколько раз поместили компонент в проект (хотя это и не имеет смысла), предусмотрено сохранение ссылки на предыдущий компонент и восстановление ее после удаления компонента.

В свойство `DefaultDatabase` можно поместить ссылку на уже имеющееся в проекте подключение к БД. Мы напишем компоненты `TfrxIBXTable`, `TfrxIBXQuery` таким образом, чтобы они могли использовать это подключение по умолчанию (собственно, ради этого нам нужна глобальная переменная IBXComponents).

Компонент Database

Следующий компонент - `TfrxIBXDatabase`. Он представляет собой обертку над `TIBDatabase`.

```
  TfrxIBXDatabase = class(TfrxCustomDatabase)
  private
    FDatabase: TIBDatabase;
    FTransaction: TIBTransaction;
    function GetSQLDialect: Integer;
    procedure SetSQLDialect(const Value: Integer);
  protected
    procedure SetConnected(Value: Boolean); override;
    procedure SetDatabaseName(const Value: String); override;
    procedure SetLoginPrompt(Value: Boolean); override;
    procedure SetParams(Value: TStrings); override;
    function GetConnected: Boolean; override;
    function GetDatabaseName: String; override;
    function GetLoginPrompt: Boolean; override;
    function GetParams: TStrings; override;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    class function GetDescription: String; override;
    procedure SetLogin(const Login, Password: String); override;
    property Database: TIBDatabase read FDatabase;
  published
    { определяем свойства, имеющиеся у TIBDatabase. Обратите внимание, что многие св-ва уже есть в
родительском классе. }
    property DatabaseName;
    property LoginPrompt;
    property Params;
    property SQLDialect: Integer read GetSQLDialect write SetSQLDialect;
    { свойство Connected надо располагать последним! }
    property Connected;
  end;

constructor TfrxIBXDatabase.Create(AOwner: TComponent);
begin
  inherited;
  { создаем компонент - подключение }
  FDatabase := TIBDatabase.Create(nil);
  { создаем компонент - транзакцию (специфика IBX) }
  FTransaction := TIBTransaction.Create(nil);
  FDatabase.DefaultTransaction := FTransaction;
  { не забудьте эту строку! }
  Component := FDatabase;
end;

destructor TfrxIBXDatabase.Destroy;
begin
  { удаляем транзакцию }
  FTransaction.Free;
  { а подключение удалится автоматически в родительском классе }
  inherited;
end;

{ описание компонента - оно будет показано рядом с иконкой в панели объектов }
class function TfrxIBXDatabase.GetDescription: String;
begin
  Result := 'IBX Database';
end;

{ перенаправляем свойства компонента на свойства обертки и наоборот }
```

```
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;

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

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

{ этот метод нужен для работы с мастером подключения к БД }
procedure TfrxIBXDatabase.SetLogin(const Login, Password: String);
begin
  Params.Text := 'user_name=' + Login + #13#10 + 'password=' + Password;
end;
```

Как видим, ничего сложного. Мы создаем объект FDatabase: TIBDatabase, и определяем свойства, которые мы хотели бы видеть в дизайнере. Для каждого свойства пишутся методы Get и Set. Аналогично поступаем с остальными классами, которые также являются обертками над соответствующими компонентами БД.

Компонент Table

Следующий класс - `TfrxIBXTable`. Он, как говорилось выше, наследуется от стандартного класса `TfrxCustomDataSet`. Вся базовая функциональность (работа со списком полей, базовыми свойствами) уже реализована в базовом классе. Нам необходимо определить только те свойства, которые являются специфичными для данного компонента. Также необходимо перекрыть методы `SetMaster`, `SetMasterFields` для работы механизма master-detail.

```
  TfrxIBXTable = class(TfrxCustomTable)
  private
    FDatabase: TfrxIBXDatabase;
    FTable: TIBTable;
    procedure SetDatabase(const Value: TfrxIBXDatabase);
  protected
    procedure Notification(AComponent: TComponent; Operation: TOperation); override;
    procedure SetMaster(const Value: TDataSource); override;
    procedure SetMasterFields(const Value: String); override;
    procedure SetIndexFieldNames(const Value: String); override;
    procedure SetIndexName(const Value: String); override;
    procedure SetTableName(const Value: String); override;
    function GetIndexFieldNames: String; override;
    function GetIndexName: String; override;
    function GetTableName: String; override;
  public
    constructor Create(AOwner: TComponent); override;
    constructor DesignCreate(AOwner: TComponent; Flags: Word); override;
    class function GetDescription: String; override;
    procedure BeforeStartReport; override;
    property Table: TIBTable read FTable;
  published
    property Database: TfrxIBXDatabase read FDatabase write SetDatabase;
  end;

constructor TfrxIBXTable.Create(AOwner: TComponent);
begin
  { создаем компонент - таблицу }
  FTable := TIBTable.Create(nil);
  { присваиваем ссылку на него свойству DataSet из базового класса - не забудьте эту строку! }
  DataSet := FTable;
  { присваиваем ссылку на подключение к БД по умолчанию }
  SetDatabase(nil);
  { после этого можно вызывать базовый конструктор }
  inherited;
end;

{ этот конструктор вызывается в момент, когда вы добавляете компонент в отчет. Он автоматически
подключает таблицу к компоненту TfrxIBXDatabase, если он уже есть. }
constructor TfrxIBXTable.DesignCreate(AOwner: TComponent; Flags: Word);
var
  i: Integer;
  l: TList;
begin
  inherited;
  l := Report.AllObjects;
  for i := 0 to l.Count - 1 do
    if TObject(l[i]) is TfrxIBXDatabase then
    begin
      SetDatabase(TfrxIBXDatabase(l[i]));
      break;
    end;
end;
```

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

{ отслеживаем удаление компонента TfrxIBXDatabase, на который мы ссылаемся в св-ве FDatabase. Иначе можем
получить ошибку. }
procedure TfrxIBXTable.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (Operation = opRemove) and (AComponent = FDatabase) then
    SetDatabase(nil);
end;

procedure TfrxIBXTable.SetDatabase(const Value: TfrxIBXDatabase);
begin
  { обратите внимание - свойство Database типа TfrxIBXDatabase, а не TIBDatabase! }
  FDatabase := Value;
  { если значение <> nil, подключаем таблицу к выбранному компоненту }
  if Value <> nil then
    FTable.Database := Value.Database
  { иначе пробуем подключить к БД по умолчанию, определенной в компоненте TfrxIBXComponents }
  else if IBXComponents <> nil then
    FTable.Database := IBXComponents.DefaultDatabase
  { если по каким-то причинам TfrxIBXComponents не существует, сбрасываем в nil }
  else
    FTable.Database := nil;
  { если подключились успешно, надо установить флаг DBConnected }
  DBConnected := FTable.Database <> nil;
end;

function TfrxIBXTable.GetIndexFieldNames: String;
begin
  Result := FTable.IndexFieldNames;
end;

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

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

procedure TfrxIBXTable.SetIndexFieldNames(const Value: String);
begin
  FTable.IndexFieldNames := Value;
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;
```

```
  FTable.IndexFieldNames := Value;
end;

{ этот метод необходим в некоторых случаях }
procedure TfrxIBXTable.BeforeStartReport;
begin
  SetDatabase(FDatabase);
end;
```

Компонент Query

Наконец, последний компонент - `TfrxIBXQuery`. Он наследуется от базового класса `TfrxCustomQuery`, в котором уже определены все необходимые свойства. Нам остается только определить свойство `Database` и перекрыть метод `SetMaster` (`SetMasterFields` в случае с Query перекрывать не надо). Реализация остальных методов аналогична компоненту `TfrxIBXTable`.

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

constructor TfrxIBXQuery.Create(AOwner: TComponent);
begin
  { создаем компонент - запрос }
  FQuery := TIBQuery.Create(nil);
  { присваиваем ссылку на него свойству DataSet из базового класса - не забудьте эту строку! }
  Dataset := FQuery;
  { присваиваем ссылку на подключение к БД по умолчанию }
  SetDatabase(nil);
  { после этого можно вызывать базовый конструктор }
  inherited;
end;

constructor TfrxIBXQuery.DesignCreate(AOwner: TComponent; Flags: Word);
var
  i: Integer;
  l: TList;
begin
  inherited;
  l := Report.AllObjects;
  for i := 0 to l.Count - 1 do
    if TObject(l[i]) is TfrxIBXDatabase then
    begin
      SetDatabase(TfrxIBXDatabase(l[i]));
      break;
    end;
end;

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

procedure TfrxIBXQuery.Notification(AComponent: TComponent; Operation: TOperation);
begin
```

```
  inherited;
  if (Operation = opRemove) and (AComponent = FDatabase) then
    SetDatabase(nil);
end;

procedure TfrxIBXQuery.SetDatabase(const Value: TfrxIBXDatabase);
begin
  FDatabase := Value;
  if Value <> nil then
    FQuery.Database := Value.Database
  else if IBXComponents <> nil then
    FQuery.Database := IBXComponents.DefaultDatabase
  else
    FQuery.Database := nil;
  DBConnected := 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
  { в этом методе необходимо присвоить значения из Params в FQuery.Params }
  { это делается стандартной процедурой }
  frxParamsToTParams(Self, FQuery.Params);
end;

procedure TfrxIBXQuery.BeforeStartReport;
begin
  SetDatabase(FDatabase);
end;
```

Регистрация компонентов

Регистрация всех компонентов движка выполняется в секции initialization.

```
initialization
  { вместо картинок используем индексы стандартных картинок 37,38,39 }
  frxObjects.RegisterObject1(TfrxIBXDataBase, nil, '', '', 0, 37);
  frxObjects.RegisterObject1(TfrxIBXTable, nil, '', '', 0, 38);
  frxObjects.RegisterObject1(TfrxIBXQuery, nil, '', '', 0, 39);

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

end.
```

Этого достаточно для того, чтобы использовать движок в отчетах. Осталось две вещи: зарегистрировать классы движка в скриптовой системе для того, чтобы к ним можно было обращаться из скрипта, и зарегистрировать редакторы некоторых свойств (например, TfrxIBXTable.TableName), чтобы с компонентом было удобно работать.

Код регистрации движка в скриптовой системе лучше вынести в отдельный файл с суффиксом RTTI. Подробнее о регистрации классов в скриптовой системе можно прочитать в соответствующей главе. Вот пример такого файла:

```
unit frxIBXRTTI;

interface

{$I frx.inc}

implementation

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

type
  TFunctions = class(TfsRTTIModule)
  public
    constructor Create(AScript: TfsScript); override;
  end;

{ TFunctions }

constructor TFunctions.Create;
begin
  inherited Create(AScript);
  with AScript do
  begin
    AddClass(TfrxIBXDatabase, 'TfrxComponent');
    AddClass(TfrxIBXTable, 'TfrxCustomDataset');
    AddClass(TfrxIBXQuery, 'TfrxCustomQuery');
  end;
end;

initialization
  fsRTTIModules.Add(TFunctions);

end.
```

Редакторы свойств

Код редакторов свойств лучше поместить в отдельный файл с суффиксом Editor. В нашем случае надо написать редакторы для свойств TfrxIBXDatabase.DatabaseName, TfrxIBXTable.IndexName, TfrxIBXTable.TableName. Подробнее о написании редакторов свойств можно прочитать в соответствующей главе. Вот пример такого файла:

```
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
  { это свойство имеет редактор }
  Result := [paDialog];
end;

function TfrxDatabaseNameProperty.Edit: Boolean;
var
  SaveConnected: Bool;
  db: TIBDatabase;
begin
  { получаем ссылку на TfrxIBXDatabase.Database }
  db := TfrxIBXDatabase(Component).Database;
  { создаем стандартный OpenDialog }
  with TOpenDialog.Create(nil) do
  begin
    InitialDir := GetCurrentDir;
    { нас интересуют файлы *.gdb }
    Filter := frxResources.Get('ftDB') + ' (*.gdb)|*.gdb|' + frxResources.Get('ftAllFiles') + '
```

```
(*.*)|*.*';
    Result := Execute;
    if Result then
    begin
      SaveConnected := db.Connected;
      db.Connected := False;
      { если диалог завершен успешно, присваиваем новое имя БД }
      db.DatabaseName := FileName;
      db.Connected := SaveConnected;
    end;
    Free;
  end;
end;

{ TfrxTableNameProperty }

function TfrxTableNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { свойство представляет собой список значений }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxTableNameProperty.GetValues;
var
  t: TIBTable;
begin
  inherited;
  { получаем ссылку на компонент TIBTable }
  t := TfrxIBXTable(Component).Table;
  { заполняем список доступных таблиц }
  if t.Database <> nil then
    t.DataBase.GetTableNames(Values, False);
end;

{ TfrxIndexProperty }

function TfrxIndexNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { свойство представляет собой список значений }
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxIndexNameProperty.GetValues;
var
  i: Integer;
begin
  inherited;
  try
    { получаем ссылку на компонент TIBTable }
    with TfrxIBXTable(Component).Table do
      if (TableName <> '') and (IndexDefs <> nil) then
      begin
        { обновляем индексы }
        IndexDefs.Update;
        { заполняем список доступных индексов }
        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.
```

Подключение собственных функций к отчету

FastReport имеет довольно большое количество стандартных функций, которые можно использовать в отчете. Вы также можете подключать свои собственные функции. Подключение функций производится через интерфейс скриптовой библиотеки FastScript, входящей в состав FastReport (подробнее о FastScript вы можете прочитать в руководстве по этой библиотеке).

Рассмотрим пример подключения процедуры и функции. Количество и тип параметров подключаемой функции может быть любым. Нельзя передавать параметры типа Set и Record, т.к. они не поддерживаются в FastScript. Такие параметры необходимо передавать в виде более простых типов, например, тип `TRect` передавать как X0, Y0, X1, Y1: Integer. Подробнее о добавлении функций со всевозможными параметрами вы можете прочитать в документации по FastScript.

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

procedure TForm1.MyProc(s: String);
begin
// нужная логика
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)');
```

Подключенную функцию можно использовать в скрипте отчета, а также обращаться к ней из объектов типа `TfrxMemoView`. Функция также отображается в окне "Дерево данных". В этом окне функции разбиты по категориям, и при выборе каждой функции внизу окна отображается подсказка по выбранной функции.

Изменим код примера, чтобы зарегистрировать функции в отдельной категории и отобразить описание функции:

```
frxReport1.AddFunction('function MyFunc(s: String; i: Integer): Boolean', 'Мои функции', 'Функция MyFunc
всегда возвращает True');
frxReport1.AddFunction('procedure MyProc(s: String)', 'Мои функции', 'Процедура MyProc не делает
ничего');
```

Если вы хотите зарегистрировать функции в одной из стандартных категорий, используйте следующие имена категорий:

'ctString' - работа со строками;

'ctDate' - работа с датой/временем;

'ctConv' - функции преобразования;

'ctFormat' - форматирование;

'ctMath' - математические функции;

'ctOther' - прочие функции.

Если указать пустое имя категории, функция будет помещена в корень дерева функций.

Если вы собираетесь подключать большое количество функций, либо использовать функции во всех отчетах, имеет смысл вынести всю логику в отдельный модуль. Вот пример такого модуля:

```
unit myfunctions;

interface

implementation

uses SysUtils, Classes, fs_iinterpreter;

type
  TFunctions = class(TfsRTTIModule)
  private
    function CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String; var Params:
Variant): Variant;
  public
    constructor Create(AScript: TfsScript); override;
  end;

function MyFunc(s: String; i: Integer): Boolean;
begin
// нужная логика
end;

procedure MyProc(s: String);
begin
// нужная логика
end;

{ TFunctions }

constructor TFunctions.Create;
begin
  inherited Create(AScript);
  with AScript do
  begin
    AddMethod('function MyFunc(s: String; i: Integer): Boolean', CallMethod, 'Мои функции', 'Функция
MyFunc всегда возвращает True');
    AddMethod('procedure MyProc(s: String)', CallMethod, 'Мои функции', 'Процедура MyProc не делает
ничего');
  end;
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
  fsRTTIModules.Add(TFunctions);

end.
```

Написание собственных мастеров

Вы можете расширить функциональность FastReport с помощью так называемых мастеров (wizards). FastReport, к примеру, содержит стандартный мастер "Мастер отчета", который вызывается из меню "Файл|Новый...".

В FastReport поддерживается 2 типа мастеров. Первый - это уже упомянутые мастера, вызываемые из меню "Файл|Новый...". Второй - это мастера, которые можно вызвать с панели инструментов "Мастера".

Базовый класс для любого мастера - это `TfrxCustomWizard`, определенный в файле frxClass.

```
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;
```

Для того, чтобы написать свой мастер, необходимо наследоваться от этого класса и перекрыть, как минимум, методы `GetDescription` и `Execute`. Первый метод возвращает название мастера; второй метод вызывается при запуске мастера и должен вернуть True, если мастер отработал успешно и внес какие-либо изменения в отчет. Во время работы мастера можно обращаться к методам и свойствам дизайнера и самого отчета через свойства `Designer` и `Report`.

Регистрация и удаление мастера выполняется с помощью процедур, описанных в файле frxDsgnIntf:

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

При регистрации указывается имя класса мастера, его картинку и то, является ли мастер размещаемым на панели инструментов "Мастера". Если мастер надо поместить на панель инструментов, размеры ButtonBmp должны быть 16x16 точек, иначе - 32x32 точки.

Рассмотрим примитивный мастер, который регистрируется в меню "Файл|Новый..." и добавляет в отчет новую страницу.

```
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
  { запрещаем любые отрисовки в дизайнере }
  Designer.Lock;
  { создаем новую страницу в отчете }
  Page := TfrxReportPage.Create(Report);
  { создаем уникальное имя для страницы }
  Page.CreateUniqueName;
  { устанавливаем размеры и ориентацию бумаги по умолчанию }
  Page.SetDefaults;
  { обновляем страницы отчета и переключаем фокус на последнюю (добавленную) страницу }
  Designer.ReloadPages(Report.PagesCount - 1);
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  { загружаем картинку из ресурса - естественно, вы ее должны предварительно туда поместить }
  Bmp.LoadFromResourceName(hInstance, 'frxMyWizard');
  frxWizards.Register(TfrxMyWizard, Bmp);

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

end.
```