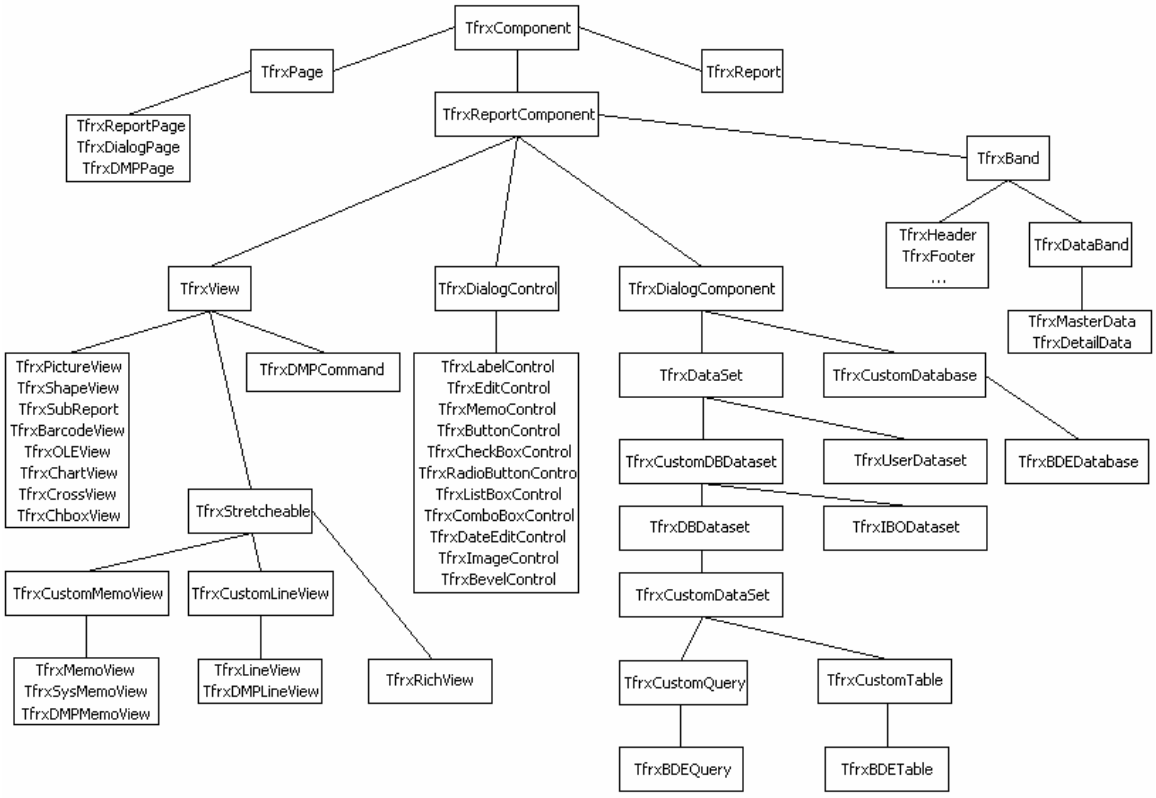
FastReport 4
Developer's Manual

Manual v 1.2.0

目次

Fast Report クラスの階層



"**TfrxComponent**" は、すべての FastReport コンポーネントの基本クラスです。この型のオブジェクトは、"座標"、"サイズ"、"フォント"、"可視性" などの属性と、下位オブジェクトの一覧を持ちます。また、このクラスは、オブジェクトの状態をストリームに保存したり、ストリームから復元したりすることができるメソッドも含んでいます。

```
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　　オブジェクトの内容を消去し、その子オブジェクトをすべて削除します
- CreateUniqueName　　レポートに配置されたオブジェクトの一意の名前を作成します
- LoadFromStream　　ストリームからオブジェクトとその子オブジェクトをすべて読み込みます
- SaveToStream　　ストリームにオブジェクトを保存します
- SetBounds　　オブジェクトの座標とサイズを設定します
- FindObject　　子オブジェクトの中で、指定された名前を持つオブジェクトを検索します
- GetDescription　　オブジェクトの説明を返します

以下のメソッドは、それに対応するプロパティを変更するときに呼び出されます。追加の処理が必要な場合は、そのメソッドをオーバーライドできます。

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

以下のプロパティが、TfrxComponent クラスで定義されています。

- Objects　　子オブジェクトの一覧を取得します
- AllObjects　　すべての下位オブジェクトの一覧
- Parent　　親オブジェクトへのリンク
- Page　　オブジェクトが属しているレポートページへのリンク
- Report　　オブジェクトが現れるレポートへのリンク
- IsDesigning　　デザイナーが起動している場合は"True"
- IsLoading　　オブジェクトがストリームから読み込まれている場合は"True"
- IsPrinting　　オブジェクトが出力されている場合は"True"
- BaseName　　オブジェクトのベース名。この値はCreateUniqueName メソッドで使用されます
- Left　　オブジェクトのＸ座標（親との相対）
- Top　　オブジェクトのＹ座標（親との相対）
- Width　　オブジェクトの幅
- Height　　オブジェクトの高さ
- AbsLeft　　オブジェクトのＸ絶対座標
- AbsTop　　オブジェクトのＹ絶対座標
- Font　　オブジェクトのフォント
- ParentFont　　"True" の場合は、親オブジェクトのフォント設定を使用します

- Restrictions 1 つ以上のオブジェクトの操作を制限するフラグのセット
- Visible オブジェクトの可視性

次の基本クラスは、**TfrxReportComponent** です。この型のオブジェクトは、レポートデザインに配置することができます。このクラスには、オブジェクトを描画するための Draw メソッドのほかに、レポートの実行時に呼び出される BeforePrint/GetData/AfterPrint メソッドもあります。

```
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 メソッドは、次のパラメーターを持ち、オブジェクトを描画するために呼び出されます。

- Canvas キャンバス
- Scale X 軸および Y 軸方向のスケーリング（拡大縮小）
- Offset キャンバスの端からの相対的なオフセット

BeforePrint メソッドは、オブジェクトを処理する直前（レポートの作成処理中）に呼び出されます。このメソッドは、オブジェクトの状態を保存します。

GetData メソッドは、オブジェクトにデータを読み込むために呼び出されます。

AfterPrint は、オブジェクト処理後に呼び出されます。このメソッドは、オブジェクトの状態を復元します。

TfrxDialogComponent は、非ビジュアル コンポーネントを記述するための基本クラスです。レポートのダイアログ フォームで使用することができます。

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

TfrxDialogControl は、コモン コントロールを記述するための基本クラスです。レポートのダイアログ フォームで使用することができます。このクラスは、ほとんどのコモン コントロールで共有されている、一般的なプロパティおよびイベントを多数含んでいます。

```
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;
```

独自のカスタム コントロールを書く場合は、このクラスから継承し、必要なプロパティを"published" セクションに移動させてから、独自のコモン コントロール用の新しいプロパティを追加する必要があります。カスタム コントロールの記述については、次の章で詳しく説明します。

TfrxView は、ほとんどのコンポーネントの基本クラスです。レポート デザイン ページに配置することができます。このクラスのオブジェクトは、Frame や Filling などの属性を持っており、データ ソースに接続することもできます。FastReport の標準オブジェクトのほとんどは、このクラスから継承されます。

```
  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;
```

以下のメソッドが、このクラスで定義されています。

- BeginDraw	Draw メソッドから呼び出され、整数値の座標および描画領域サイズを計算します 計算された値は、FX、FY、FX1、FY1、FDX、および FDY 変数として公開されます また、枠線の幅（FFrameWidth として公開）も計算されます
- DrawBackground	オブジェクトの背景を描画します
- DrawFrame	オブジェクトの枠を描画します
- DrawLine	指定された座標と幅（太さ）を使用して、線を描画します
- IsDataField	DataSet および DataField プロパティに空でない値が含まれている場合は、"True" を返します

BeginDraw メソッドを呼び出した後、次のプロパティを参照することができます。

- FX, FY, FX1, FY1, FDX, FDY, FFrameWidth — Scale および Offset パラメーターを使って計算された、オブジェクトの枠線の座標、サイズ、および幅
- FscaleX, FScaleY — X および Y 方向のスケーリング。これらは、Draw メソッドの ScaleX および ScaleY パラメーターのコピーです
- FCanvas — キャンバス。これは Draw メソッドの Canvas パラメーターのコピーです

このクラスには、ほとんどのレポートオブジェクトで見つけられる以下のプロパティが定義されています。

- BrushStyle — オブジェクトの塗りつぶしスタイル
- Color — オブジェクトの塗りつぶしの色
- DataField — オブジェクトが接続されているデータフィールドの名前
- DataSet — データソース
- Frame — オブジェクトの枠線
- Align — オブジェクトとその親との相対的な配置
- Printable — 指定されたオブジェクトを印刷できるかどうかを定義します
- ShiftMode — オブジェクトの上に伸縮可能なオブジェクトが配置された場合の、オブジェクトのシフトモード
- TagStr — ユーザー情報を保管するためのフィールド

TfrxStretcheable は、収容されるデータに応じて高さを変えるコンポーネントを記述するための基本クラスです。

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

このクラスのオブジェクトは、縦方向に引き伸ばすことができるほか、出力ページに十分な幅がない場合には、より小さな断片に分けることもできます。オブジェクトは、そのデータがすべて出力可能なスペースに収まるまで分けられます。

以下のメソッドが、このクラスで定義されています。

- CalcHeight — 収容されるデータを基にオブジェクトの高さを計算し、その値を返します
- InitPart — オブジェクトを分割する前に呼び出されます
- DrawPart — オブジェクトに収容されている次のデータチャンクを描画します
戻り値は、データを表示することができなかった未使用スペースの量です

カスタム レポートコンポーネントの記述

FastReport は、レポートデザインページに配置することができる、多数のコンポーネントを備えています。それらには、テキスト、図、線、幾何学的図形、OLE、リッチ テキスト、バーコード、グラフなどがあります。また、独自のカスタム コンポーネントを記述して、それをFastReport に加えることもできます。

FastReport にはいくつかのクラスがあり、そこからコンポーネントを継承できます。詳細については、「FastReport クラスの階層」を参照してください。TfrxView クラスは、ほとんどのレポート コンポーネントがこれから継承されるため、最も重要なものです。

少なくとも、TfrxReportComponent 基本クラスの Draw メソッドが定義されている必要があります。

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

このメソッドは、コンポーネントがデザイナーやプレビュー ウィンドウ、あるいは出力の印刷で描画されるときに呼び出されます。TfrxView は、オブジェクトの枠線と背景を描画するために、このメソッドをオーバーライドします。このメソッドは、"Canvas" 描画面上にコンポーネントの内容を描画する必要があります。オブジェクトの座標とサイズは、それぞれ AbsLeft および AbsTop と、Width および Height プロパティに格納されます。

ScaleX および ScaleY パラメーターは、それぞれ X 軸および Y 軸におけるオブジェクトのスケーリングを定義します。これらのパラメーターは、100% ズームで 1 となり、ユーザーがデザイナーやプレビュー ウィンドウでズームを変更すると、値が変わります。OffsetX および OffsetY パラメーターは、オブジェクトを X 軸および Y 軸方向に移動させます。したがって、これらすべてのパラメーターを考慮に入れると、左上隅の座標は次のようになります。

```
X := Round(AbsLeft * ScaleX + OffsetX);
Y := Round(AbsTop * ScaleY + OffsetY);
```

座標の操作を簡略化するために、TfrxView クラスには BeginDraw メソッド（Draw と同様のパラメーターを持ちます）が定義されています。

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

このメソッドは、Draw メソッドの最初の行で呼び出す必要があります。これは、座標を整数値の FX、FY、FX1、FY1、FDX、FDY、FFrameWidth に変換します。これらは後から、TCanvas メソッドで使用することができます。また、Canvas、ScaleX、および ScaleY の値を、任意のクラスのメソッドから参照できる FCanvas、FScaleX、および FScaleY 変数へコピーします。

TfrxView クラスには、オブジェクトの背景と枠線を描画するための 2 つのメソッドがあります。

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

これら2 つのメソッドを呼び出す前に、BeginDraw メソッドを呼び出す必要があります。

矢印を表示するコンポーネントの作成方法について検討してみましょう。

```
type
  TfrxArrowView = class(TfrxView)
  public
    { 2 つのメソッドのみオーバーライドする必要がある }
    procedure Draw(Canvas: TCanvas;
              ScaleX, ScaleY, OffsetX, OffsetY: Extended); override;
    class function GetDescription: String; override;
  published
    { 必要なプロパティを published セクションに置く }
    property BrushStyle;
    property Color;
    property Frame;
  end;

class function TfrxArrowView.GetDescription: String;
```

```
begin
  { ツール バー内のコンポーネントのアイコンの隣に、コンポーネントの説明が表示される}
  Result := 'Arrow object';
end;

procedure TfrxArrowView.Draw(Canvas: TCanvas;
                             ScaleX, ScaleY, OffsetX, OffsetY: Extended);
begin
  { このメソッドを呼び出して、座標を変換する}
  BeginDraw(Canvas, ScaleX, ScaleY, OffsetX, OffsetY);
  with Canvas do
  begin
    { 色を設定する}
    Brush.Color := Color;
    Brush.Style := BrushStyle;
    Pen.Width := FFrameWidth;
    Pen.Color := Frame.Color;
    { 矢印を描画する}
    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;

{ 登録 }
var
  Bmp: TBitmap;

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

finalization
  { 使用可能なコンポーネントの一覧から削除する}
  frxObjects.Unregister(TfrxArrowView);
  Bmp.Free;

end.
```

データベースからデータを表示するコンポーネントを作成するには、DataSet および DataField プロパティを published セクションに移動させてから、GetData メソッドをオーバーライドします。TfrxCheckBoxView 標準コンポーネントを例にとって、これを見てみましょう。

TfrxCheckBoxView コンポーネントは、TfrxView 基本クラスで宣言されている DataSet および DataField プロパティを使用して、データベース フィールドに接続することができます。また、このコンポーネントには、式を入れることができる Expression プロパティがあります。式が計算されるとすぐに、結果が Checked プロパティに入れられます。Checked プロパティが "True" である場合、コンポーネントはバツ印を表示します。コンポーネントの定義の最も重要な部分は以下のとおりです。

```
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;
```

カスタム コモン コントロールの記述

FastReport には、レポート内のダイアログ フォームに配置することができる、一連のコモン コントロールが含まれています。それは次のとおりです。

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

これらのコントロールは、Delphi のコンポーネント パレットの標準 コントロールに対応しています。標準的な機能では十分でない場合は、自身のレポートで使用するための、独自のコモン コントロールを作成することができます。

すべてのコモン コントロールの基本クラスは、frxClass ファイルで宣言されている**TfrxDialogControl** クラスです。

```
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;
```

独自のコントロールを作成するには、このクラスから継承して、少なくともコンストラクターとGetDescription メソッドをオーバーライドする必要があります。コモン コントロールを作成し、コンストラクターの InitControl メソッドを使ってそれを初期化する必要があります。GetDescription メソッドは、コモン コントロールの説明を返すためのものです。ご覧のとおり、TfrxDialogControl クラスには

既に、public セクションに多数のプロパティとメソッドが含まれています。必要に応じて、プロパティやイベントをコモン コントロールの published セクションに移動させ、さらに、独自のコントロールに特有の新しいプロパティを作成します。

コモン コントロールの登録および削除は、frxDsgnIntf ファイルで宣言されている frxObjects グローバル オブジェクトのメソッドを使用して実行されます。

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

登録時には、コントロールのクラス名とその画像を指定する必要があります。ButtonBmp のサイズは、16x16 ピクセルでなければなりません。

Delphi の標準コントロール TBitBtn の機能を簡略化する、コモン コントロールの例を見てみましょう。

```
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
    { 新しいプロパティを追加する}
    property Kind: TBitBtnKind read GetKind
                          write SetKind default bkCustom;
    { 以下のプロパティは、親クラスに既に宣言されている}
    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
  { 既定のコンストラクター}
  inherited;
  { 必要なコモン コントロールを作成する}
  FButton := TBitBtn.Create(nil);
  FButton.Caption := 'BitBtn';
  { 初期化する}
  InitControl(FButton);

  { 既定のサイズを設定する}
  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;
  { リソースから図を読み込む。
    当然ながら、あらかじめそこに配置しておく必要があります }
  Bmp.LoadFromResourceName(hInstance, 'frxBitBtnControl');
  frxObjects.RegisterObject(TfrxBitBtnControl, Bmp);

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

end.
```

イベントハンドラーの記述

新しいイベントハンドラーがまた基本クラスに属していない場合、そのイベントハンドラーをどのように定義すればよいでしょうか？TfrxEditControl コモン コントロールを例にとって、これを見てみましょう。

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

constructor TfrxEditControl.Create(AOwner: TComponent);
begin
  ...
  { ハンドラーに接続する}
  FEdit.OnChange := DoOnChange;
  InitControl(FEdit);
  ...
end;

procedure TfrxEditControl.DoOnChange(Sender: TObject);
begin
  { イベントハンドラーを呼び出す}
  if Report <> nil then
    Report.DoNotifyEvent(Sender, FOnChange);
end;
```

重要なのは、FastReport のイベントハンドラーは、レポートスクリプトで定義されたプロシージャであることに注目することです。TfrxNotifyEvent 型は String[63] として宣言されています。FastReport では、ハンドラーへのリンクはハンドラーの名前を含む文字列です。これは、Delphi の TNotifyEvent 型とは違い、メソッドのアドレスです。

スクリプトシステムへのコンポーネントの登録

スクリプトからコンポーネントを参照するには、クラスと、そのプロパティおよびメソッドを最初にスクリプトシステムに登録しておく必要があります。登録コードは、コンポーネントコードのファイルと同じ名前の、末尾に'RTTI' を付け加えた名前のファイル（次の場合におけるfrxBitBtnRTTI.pas）に置くことができます。FastScript におけるクラス、登録、メソッド、およびプロパティの詳細については、スクリプトライブラリのドキュメントを参照してください。

```
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
    { クラスを登録してから、その親を定義する}
    AddClass(TfrxBitBtnControl, 'TfrxDialogControl');

    { 作成したユニットに複数のコモン コントロールがある場合は、
      ここでそれらを登録できる
      例 :AddClass(TfrxAnotherControl, 'TfrxDialogControl'); }
  end;
end;


initialization
  fsRTTIModules.Add(TFunctions);

end.
```

コンポーネントエディターの記述

コモン コントロール エディター (コントロールのコントロール メニューから、またはダブルクリックによって開かれます) はすべて、既定で空の OnClick イベント ハンドラーを作成します。この動作は、カスタム エディターを記述することによって変えることができます。また、カスタム エディターでは、コンポーネントのコンテキスト メニューに項目を追加することができます。

すべてのエディターの基本クラスは、frxDsgnIntf ファイルで宣言されている TfrxComponentEditor です。

```
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;
```

エディターがコンテキスト メニューに独自の項目を作成しない場合は、Edit および HasEditor の 2 つのメソッドをオーバーライドする必要があります。1 つ目のメソッドは、必須の操作を実行する (たとえば、ダイアログ ボックスを表示する) もので、コンポーネントの内容が変更された場合は、"True" を返します。HasEditor メソッドは、コンポーネントにエディターがある場合には、"True" を返す必要があります。"False" が返されるか、このメソッドをオーバーライドしなかった場合、エディターは開きません。コンポーネントにエディターが備わっていないときに、コンポーネントのコンテキスト メニューに項目を追加したい場合は、"False" を返す必要があります。

エディターがコンテキスト メニューに項目を追加する場合は、GetMenuItems と Execute をオーバーライドする必要があります。GetMenuItems では、AddItem 関数を使用してメニューを作成できます。Execute では、コンポーネント メニュー内の項目を選択することによって開始される操作を作成できます。

エディターの登録は、frxDsgnIntf ファイルに定義されているプロシージャを使用して実行されます。

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

最初のパラメーターは、エディターの作成対象となるコンポーネントのクラス名です。2 番目のパラメーターは、エディターのクラス名です。

コモン コントロール用のシンプルなエディターを見てみましょう。このエディターは、要素名を含んでいるウィンドウを表示し、コントロールのコンテキスト メニューに"Enabled" と"Visible" 項目を追加します (これは、Enabled プロパティと Visible プロパティの変更を行います)。FastReport では、エディター コードは、コンポーネントのコードを含んでいるファイルと同じ名前の、末尾に'Editor' を付け加えた名前のファイル (次の場合における frxBitBtnEditor.pas) に置かれていることを必要とします。

```
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;
```

```
  { コンポーネントプロパティは編集されるコンポーネントです。
    この場合は、TfrxBitBtnControl です }
  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);
  { AddItem メソッドのパラメーター：メニュー項目の名前、
    タグ、チェック/未チェックの状態 }
  AddItem('Enabled', 1, c.Enabled);
  AddItem('Visible', 2, c.Visible);
end;

initialization
  frxComponentEditors.Register(TfrxBitBtnControl, TfrxBitBtnEditor);

end.
```

プロパティエディターの記述

デザイナーでコンポーネントを選択すると、そのプロパティがオブジェクトインスペクターに表示されます。任意のプロパティに対する独自のエディターを作成することができます。たとえば、Font プロパティにはエディターがあります。このプロパティが選択されると、[..] ボタンが行の右側に現れます。このボタンをクリックすることにより、標準の「フォントのプロパティ」ダイアログボックスが開きます。もう1 つの例は、Color プロパティです。これは、ドロップダウンリストに標準の色と色見本の名前を公開します。

TfrxPropertyEditor は、すべてのプロパティエディターの基本クラスであり、frxDsgnIntf ユニットで宣言されています。

```
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;
```

また、次のクラスのいずれかから継承することもできます。これらのクラスは、対応する型のプロパティを操作するためのいくつかの基本的な機能を提供します。

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

いくつかのプロパティが、TfrxPropertyEditor クラスで定義されています。

- Component	指定されたプロパティが属する、親コンポーネントへのリンク（プロパティ自体へのリンクではありません）
- frComponent	上と同じですが、TfrxComponent 型にキャストされます 場合によっては便利）
- Designer	レポートデザイナーへのリンク
- ItemHeight	プロパティが表示される項目の高さ。OnDrawXXX で有用です
- PropInfo	編集されたプロパティに関する情報を格納する、PPropInfo 構造体へのリンク

- Value　文字列として表示されるプロパティ値
- Values　値の一覧。このプロパティは、paValueList 属性が定義されている場合に、GetValue メソッドで埋めるためのものです（下記参照）

以下は、システム メソッドです。編集されたプロパティ値の取得と設定に使用できます。

```
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);
```

プロパティの型に適合するメソッドを使用する必要があります。したがって、プロパティが Integer 型の場合は、GetOrdValue と SetOrdValue メソッドを使用する、などとなります。このようなプロパティには 32 ビットのオブジェクト アドレスが含まれているため、これらのメソッドは、TObject 型のプロパティを操作する場合にも使用されます。この場合は、次のようなキャストを使用するだけです。MyFont := TFont(GetOrdValue)

独自のエディターを作成するには、基本クラスから継承し、public セクションで宣言されているいくつかのメソッドをオーバーライドします（対象となるメソッドは、実装したいプロパティの型と機能によって決まります）。オーバーライドする必要のあるメソッドの 1 つは、GetAttributes です。これは、プロパティ属性のセットを返します。属性は、次のように定義されています。

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

属性の意味は次のとおりです。

- paValueList　値のドロップダウン リストを表します（たとえば、Color プロパティ）。この属性が存在する場合は、GetValues メソッドをオーバーライドする必要があります
- paSortList　リストの値を並べ替えます。paValueList と一緒に使用されます
- paDialog　エディターを備えています。この属性が存在する場合は、行の右側に [...] ボタンが表示されます。それをクリックすると、Edit メソッドが呼び出されます
- paMultiSelect　同時に 2 つ以上のオブジェクトを選択できます。一部のプロパティ（"Name" など）には、この属性はありません
- paSubProperties　TPersistent 型のオブジェクトで、表示する必要のある独自のプロパティを持っています（たとえば、Font プロパティ）
- paReadOnly　エディターで値を変更することはできません。一部の、Class 型または Set 型のプロパティに、この属性があります
- paOwnerDraw　プロパティ値は、OnDrawItem メソッドによって描画されます。paValueList 属性が存在する場合は、OnDrawLBItem メソッドによってドロップダウン リストが描画されます

Edit メソッドは、次の 2 つの方法で呼び出されます。プロパティを選択し、その値をダブルクリックする、または、（プロパティに paDialog 属性がある場合は）[...] ボタンをクリックする、のいずれかです。プロパティ値が変更された場合、このメソッドは "True" を返す必要があります。

GetValue メソッドは、プロパティ値を文字列として返す必要があります（この値は、オブジェクト インスペクターに表示されます）。TfrxPropertyEditor 基本クラスから継承している場合は、このメソッドをオーバーライドする必要があります。

SetValue メソッドは、文字列として転送されたプロパティ値を設定するものです。TfrxPropertyEditor 基本クラスから継承している場合は、このメソッドをオーバーライドする必要があります。

paValueList 属性が設定されている場合は、GetValues メソッドをオーバーライドする必要があります。このメソッドは、複数の値によって Values プロパティを埋めなければなりません。

以下の 3 つのメソッドは、手動によるプロパティ値の描画を可能にします（Color プロパティエディターが同じように動作します）。これらのメソッドは、paOwnerDraw 属性が設定されている場合に呼び出されます。

OnDrawItem メソッドは、プロパティ値がオブジェクト インスペクターに描画されるときに呼び出されます（プロパティが選択されて

いない場合。選択されている場合は、単に値が編集行に表示されるだけです)。たとえば、Color プロパティエディターは、プロパティ値の左側に、対応する色で塗りつぶされた四角形を描画します。

GetExtraLBSize メソッドは、paValueList 属性が設定されている場合に呼び出されます。このメソッドは、リストを表示するためにドロップダウンリストの幅をどれだけ調整すればよいかを示す、ピクセル数を返します。既定では、セルの高さに相当する値を返します。高さよりも幅の方が大きい図を描画する必要がある場合は、指定されたメソッドをオーバーライドする必要があります。

OnDrawLBItem メソッドは、paValueList 属性が設定されている場合、ドロップダウンリストに文字列を描画するときに呼び出されます。このメソッドは、実質的にはTListBox.OnDrawItem イベントハンドラーで、同じパラメーターのセットを持ちます。

プロパティエディターの登録は、frxDsgnIntf ファイルに定義されているプロシージャによって実行されます。

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

- PropertyType　TypeInfo(String) などの TypeInfo システム関数によって返される、プロパティ型に関する情報
- ComponentClass　編集したいプロパティを含んでいる、コンポーネントの名前 (nil となる場合があります)
- PropertyName　編集したいプロパティの名前 (空文字列となる場合があります)
- EditorClass　プロパティエディターの名前

指定する必要があるのは、PropertyType パラメーターだけです。ComponentClass および PropertyName パラメーターは、空白でもかまいません。これを使用して、次のものにエディターを登録することができます。対象となるのは、PropertyType 型の任意のプロパティ、具象 ComponentClass コンポーネントとその子孫の任意のプロパティ、特定コンポーネントの PropertyName 特定プロパティ (あるいは、ComponentClass パラメーターが空の場合は、任意のコンポーネント) です。

3 つのプロパティエディターの例を見てみましょう。FastReport では、エディター コードは、コンポーネントのコードを含んでいるファイルと同じ名前の、末尾に'Editor' を付け加えた名前のファイル (次の場合における frxBitBtnEditor.pas) に置かれていることを必要とします。

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

{ TFont プロパティエディターは、エディター ボタン ([..]) を表示します }
{ ClassProperty から継承する }
type
  TfrxFontProperty = class(TfrxClassProperty)
  public
    function Edit: Boolean; override;
    function GetAttributes: TfrxPropertyAttributes; override;
  end;

function TfrxFontProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { プロパティには、入れ子になったプロパティとエディターがあります。
    手動で編集することはできません }
  Result := [paMultiSelect, paDialog, paSubProperties, paReadOnly];
end;

function TfrxFontProperty.Edit: Boolean;
var
  FontDialog: TFontDialog;
begin
  { 標準ダイアログを作成する }
  FontDialog := TFontDialog.Create(Application);
  try
    { プロパティ値を取得する }
    FontDialog.Font := TFont(GetOrdValue);
    FontDialog.Options := FontDialog.Options + [fdForceFontExist];
    { ダイアログを表示する }
    Result := FontDialog.Execute;
    { 新しい値を設定する }
```

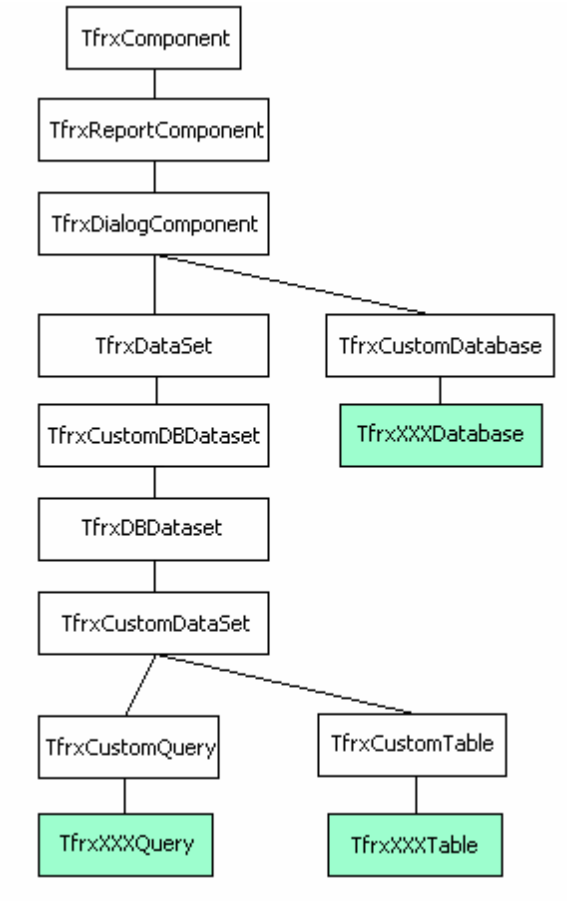
```
    if Result then
      SetOrdValue(Integer(FontDialog.Font));
  finally
    FontDialog.Free;
  end;
end;

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

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

{ TFont.Name プロパティエディターは、使用可能なフォントの
  ドロップダウン リストを表示します }
  { プロパティは文字列型であるとして、StringProperty から継承する }
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;

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


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

{ TPen.Style プロパティエディターは、選択したスタイルの
  例となる図を表示します }
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;

{ メソッドは、選択されたスタイルで太い横線を描画します }
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;

{ ドロップダウン リストを描画する }
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;

{ プロパティ値を描画する }
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;

{ 図の幅を返す }
function TfrxPenStyleProperty.GetExtraLBSize: Integer;
begin
  Result := 36;
end;

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

カスタム DB エンジンの記述

FastReport は、Delphi アプリケーションからだけでなく、レポート自体の中で作成されるデータソース (DB への接続、クエリ) から供給されるデータに基づいて、レポートを構築することができます。FastReport には、ADO、BDE、IBX、DBX、および FIB のエンジンが付属しています。独自のエンジンを作成し、それを FastReport に接続することができます。

下の図は、DB エンジンの作成に必要なクラス階層を示しています。新しいエンジン コンポーネントは、緑色で強調されています。



DB エンジン コンポーネントの標準セットには、Database、Table、および Query が含まれます。これらのコンポーネントの全部または一部を作成できます (たとえば、多くの DB は Table 型のコンポーネントを持ちません)。また、標準セットに含まれないコンポーネント、たとえば StoredProc などを作成することもできます。

基本クラスについて詳しく見てみましょう。

TfrxDialogComponent は、FastReport デザインのダイアログ フォームに配置することができる、すべての非ビジュアル コンポーネントの基本クラスです。この中には、重要なプロパティやメソッドは何も定義されていません。

TfrxCustomDatabase クラスは、Database 型の DB コンポーネントの基本クラスです。

```
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;
```

以下のプロパティが、このクラスで定義されています。

- Connected　　DB 接続がアクティブかどうか
- DatabaseName　　データベースの名前
- LoginPrompt　　DB への接続時にログインを要求するかどうか
- Params　　接続パラメーター

TfrxXXXDatabase 型のコンポーネントを作成するには、このクラスから継承します。作成したら、すべての仮想メソッドをオーバーライドし、必要なプロパティをpublished セクションに置く必要があります。さらに、そのコンポーネント固有のプロパティを追加します。

TfrxDataset、**TfrxCustomDBDataset**、および**TfrxDBDataset** クラスは、データアクセスの機能を提供します。FastReport のコアは、ナビゲーションと、データフィールドの参照のために、これらのコンポーネントを使用します。それらは共通の階層の一部なので、私たちには関係ありません。

TfrxCustomDataSet は、TDataSet から派生するDB コンポーネントの基本クラスです。このクラスから継承されるコンポーネントは、Query、Table、およびStoredProc の複製です。実際のところ、このクラスはTDataSet のラッパーです。

```
  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;
```

以下のプロパティが、このクラスで定義されています。

- DataSet　　TdataSet 型の密閉されたオブジェクトへのリンク
- Fields　　DataSet.Fields へのリンク
- Active　　DataSet がアクティブかどうか
- Filter　　フィルターの式
- Filtered　　フィルターリングがアクティブかどうか
- Master　　マスター/詳細関係のマスター データセットへのリンク
- MasterFields　　'field1=field2' のような、フィールドの一覧。マスター/詳細の関係で使用されます

TfrxCustomTable クラスは、Table 型のDB コンポーネントの基本クラスです。このクラスは、TTable クラスのラッパーです。

```
  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;
```

以下のプロパティが、このクラスで定義されています。

- TableName　テーブルの名前
- IndexName　インデックスの名前
- IndexFieldNames　インデックス フィールドの名前

Table 型のコンポーネントは、このクラスから継承されます。このクラスの子孫を作成するときに、Database のようないくつかの不足しているプロパティを追加する必要があります。また、TfrxCustomDataset クラスおよび TfrxCustomTable クラスの仮想メソッドをオーバーライドする必要があります。

TfrxCustomQuery は、Query 型の DB コンポーネントの基本クラスです。このクラスは、TQuery クラスのラッパーです。

```
  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;
```

すべての Query コンポーネントに見つかる SQL プロパティと Params プロパティは、このクラスで宣言されています。Query コンポーネントは、TParams や TParameters などのように、さまざまな方法でパラメーターを実装できることから、Params プロパティは TfrxParams 型です。この型は、すべてのパラメーター型のラッパーです。

以下のメソッドが、このクラスで宣言されています。

- SetSQL　Query 型の SQL プロパティを設定します
- GetSQL　Query 型の SQL プロパティを取得します
- UpdateParams　Query 型のコンポーネントにパラメーター値をコピーします。Query コンポーネントのパラメーターが TParams 型である場合は、frxParamsToTParams の標準プロシージャを用いてコピーが行われます

IBX の例を使用して、DB エンジンの作成について説明しましょう。エンジンのフル テキストは、SOURCE\IBX フォルダーにあります。以下の例は、元のテキストから一部を抜粋して、コメントを付けたものです。

ラッパーを構築する IBX コンポーネントは、TIBDatabase、TIBTable、および TIBQuery です。対応するコンポーネントの名前は、"TfrxIBXDatabase"、"TfrxIBXTable"、および "TfrxIBXQuery" になります。

"TfrxIBXComponents" は、作成する必要のあるもう1 つのコンポーネントです。これは、Delphi 環境でエンジンを登録するときに、FastReport のコンポーネント パレットに配置されます。このコンポーネントをプロジェクトで使用するとすぐに、Delphi はエンジン ユニットへのリンクを "Users" リストに自動的に追加します。このコンポーネントを完成させるためのタスクがもう1 つあります。それは、DB への既存の接続を参照する、DefaultDatabase プロパティを定義することです。既定で、すべての TfrxIBXTable および TfrxIBXQuery コンポーネントは、この接続を使用します。TfrxIBXComponents コンポーネントは、TfrxDBComponents クラスから継承する必要があります。

```
  TfrxDBComponents = class(TComponent)
```

```
public
  function GetDescription: String; virtual; abstract;
end;
```

"IBX コンポーネント" などの説明は、1 つの関数によってのみ返される必要があります。TfrxIBXComponents コンポーネントは次のように宣言されます。

```
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;
```

TfrxIBXComponents コンポーネントのコピーを参照する、IBXComponents グローバル変数を宣言します。プロジェクトにコンポーネントを数回配置した場合、この行為は無意味ではありますが、前のコンポーネントへのリンクを保存し、コンポーネントを削除した後にそれを復元することが可能になります。

プロジェクトに既存する DB 接続へのリンクを、DefaultDatabase プロパティに設定することができます。後述の、TfrxIBXTable および TfrxIBXQuery コンポーネントを記述する方法では、コンポーネントは既定でこの接続を使用できるようになります（実際には、これは IBXComponents グローバル変数の目的です）。

以下は TfrxIBXDatabase コンポーネントです。これは、TIBDatabase クラスのラッパーです。

```
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
  { TIBDatabase プロパティを一覧表示する。
    メモ - 一部のプロパティは、基本クラスに既に存在します }
  property DatabaseName;
  property LoginPrompt;
  property Params;
  property SQLDialect:Integer read GetSQLDialect write SetSQLDialect;
  { Connected プロパティは最後に配置する必要があります }
  property Connected;
end;

constructor TfrxIBXDatabase.Create(AOwner: TComponent);
begin
  inherited;
  { コンポーネントを作成する – 接続 }
  FDatabase := TIBDatabase.Create(nil);
  { コンポーネントを作成する - トランザクション（IBX に固有） }
  FTransaction := TIBTransaction.Create(nil);
  FDatabase.DefaultTransaction := FTransaction;
  { この行を忘れないでください }
  Component := FDatabase;
end;

destructor TfrxIBXDatabase.Destroy;
begin
  { トランザクションを削除する }
  FTransaction.Free;
  { 接続は、親クラスで自動的に削除されます }
  inherited;
end;

{ オブジェクト ツール バー内のアイコンの隣に、コンポーネントの説明が表示されます }
class function TfrxIBXDatabase.GetDescription: String;
begin
  Result := 'IBX Database';
end;

{ プロパティをカバーするために、コンポーネントのプロパティをリダイレクトする。逆の場合も同じ }
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;

{ このメソッドは、DB 接続ウィザードで使用されます }
procedure TfrxIBXDatabase.SetLogin(const Login, Password: String);
begin
  Params.Text := 'user_name=' + Login + #13#10 + 'password=' + Password;
end;
```

ご覧のように、これはそれほど複雑ではありません。FDatabase: "TIBDatabase" オブジェクトを作成して、デザイナーに表示したいプロパティを定義します。"Get" と"Set" メソッドは、各プロパティに書き込まれました。

次のクラスは、**TfrxIBXTable** です。上述したように、TfrxCustomDataSet 標準クラスから継承します。基本的な機能（フィールドの一覧や、マスター/詳細、および基本プロパティでの動作）はすべて、既に基本クラスに実装されています。行う必要があるのは、このコンポーネントに固有のプロパティを宣言することだけです。

```
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
  { コンポーネントを作成する – テーブル }
  FTable := TIBTable.Create(nil);
  { 基本クラスの DataSet プロパティへのリンクを割り当てる
    - この行を忘れないでください }
  DataSet := FTable;
  { 既定として、DB への接続へのリンクを割り当てる }
  SetDatabase(nil);
  { 基本コンストラクターを呼び出すことができた後 }
  inherited;
end;

{ このコンストラクターは、レポートにコンポーネントを追加する時点で呼び出されます。
  TfrxIBXDatabase コンポーネントが既に存在する場合は、
  自動的にそれにテーブルを接続します。
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;

{ TfrxIBXDatabase コンポーネントの削除を追跡する。FDatabase プロパティで
  このコンポーネントをアドレス指定します。指定しないと、エラーが発生します。}
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
  { TfrxIBXDatabase 型の Database プロパティ。TIBDatabase 型ではありません }
 FDatabase := Value;
```

```
  { value <> nil の場合は、選択したコンポーネントにテーブルを接続する }
  if Value <> nil then
    FTable.Database := Value.Database
  { そうでない場合は、TfrxIBXComponents コンポーネントで定義されている、
    既定の DB への接続を試みる }
  else if IBXComponents <> nil then
    FTable.Database := IBXComponents.DefaultDatabase
  { 何らかの理由で TfrxIBXComponents が存在しない場合は、nil にリセットする }
  else
    FTable.Database := nil;
  { 接続が成功した場合は、DBConnected フラグを置く必要があります }
  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;

{ 場合によっては、このメソッドを実装する必要があります }
procedure TfrxIBXTable.BeforeStartReport;
begin
  SetDatabase(FDatabase);
end;
```

いよいよ、最後のコンポーネントの "**TfrxIBXQuery**" を見てみましょう。これは TfrxCustomQuery 基本クラスから継承します。こ

のクラスには、必要なプロパティが既に宣言されています。行う必要があるのは、Database プロパティを宣言することと、SetMaster メソッドをオーバーライドすることだけです。その他のメソッドの実装は、TfrxIBXTable コンポーネントと同様です。

```
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
  { コンポーネントを作成する – クエリ}
  FQuery := TIBQuery.Create(nil);
  { 基本クラスの DataSet プロパティへのリンクを割り当てる
    - この行を忘れないでください }
  Dataset := FQuery;
  { 既定として、DB への接続へのリンクを割り当てる }
  SetDatabase(nil);
  { 基本コンストラクターを呼び出すことができた後 }
  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
  { このメソッドでは、Params の値を FQuery.Params に
    代入するだけで十分です }
  { これは、標準のプロシージャを介して実行されます }
  frxParamsToTParams(Self, FQuery.Params);
end;

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

すべてのエンジン コンポーネントの登録は、"Initialization" セクションで実行されます。

```
initialization
  { 図の代わりに、標準の図のインデックス 37、38、39 を使用する }
  frxObjects.RegisterObject1(TfrxIBXDataBase, nil, '', '', 0, 37);
  frxObjects.RegisterObject1(TfrxIBXTable, nil, '', '', 0, 38);
  frxObjects.RegisterObject1(TfrxIBXQuery, nil, '', '', 0, 39);

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

end.
```

レポートでエンジンを使用するには、これで十分です。この段階で残っているのは、あと2 つです。エンジン クラスをスクリプト システムに登録して、スクリプトでそれらのクラスを参照できるようにすることと、いくつかのプロパティ エディター（たとえば TfrxIBXTable.TableName）を登録して、コンポーネントをより簡単に操作できるようにすることです。

エンジンの登録コードは、末尾にRTTI を付けた別のファイルに格納することをお勧めします。スクリプトシステムへのクラスの登録の詳細については、適切な章を参照してください。ファイルの例を以下に示します。

```
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.
```

同様に、プロパティエディターのコードは、末尾に'Editor' を付けた別のファイルに格納することをお勧めします。私たちの場合は、TfrxIBXDatabase.DatabaseName、TfrxIBXTable.IndexName、およびTfrxIBXTable.TableName プロパティのエディターが必要でした。プロパティエディターの記述方法の詳細については、適切な章を参照してください。ファイルの例を以下に示します。

```
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
  { このプロパティはエディターを処理します }
  Result := [paDialog];
end;

function TfrxDatabaseNameProperty.Edit: Boolean;
var
  SaveConnected: Bool;
  db: TIBDatabase;
begin
  { TfrxIBXDatabase.Database へのリンクを取得する }
  db := TfrxIBXDatabase(Component).Database;
  { 標準の OpenDialog を表示する }
  with TOpenDialog.Create(nil) do
  begin
    InitialDir := GetCurrentDir;
    { 関係があるのは *.gdb ファイルです }
    Filter := frxResources.Get('ftDB') + ' (*.gdb)|*.gdb|'
            + frxResources.Get('ftAllFiles') + ' (*.*)|*.*';
    Result := Execute;
    if Result then
    begin
      SaveConnected := db.Connected;
      db.Connected := False;
      { ダイアログが正常に完了したら 新しい DB 名を割り当てる }
      db.DatabaseName := FileName;
      db.Connected := SaveConnected;
    end;
    Free;
  end;
end;


{ TfrxTableNameProperty }

function TfrxTableNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { プロパティは、値の一覧を表します }
  Result := [paMultiSelect, paValueList];
end;
```

```
procedure TfrxTableNameProperty.GetValues;
var
  t: TIBTable;
begin
  inherited;
  { TIBTable コンポーネントへのリンクを取得する}
  t := TfrxIBXTable(Component).Table;
  { 利用可能なテーブルの一覧を入力する}
  if t.Database <> nil then
    t.DataBase.GetTableNames(Values, False);
end;


{ TfrxIndexProperty }

function TfrxIndexNameProperty.GetAttributes: TfrxPropertyAttributes;
begin
  { プロパティは、値の一覧を表します}
  Result := [paMultiSelect, paValueList];
end;

procedure TfrxIndexNameProperty.GetValues;
var
  i: Integer;
begin
  inherited;
  try
    { TIBTable コンポーネントへのリンクを取得する}
    with TfrxIBXTable(Component).Table do
      if (TableName <> '') and (IndexDefs <> nil) then
      begin
        { インデックスを更新する}
        IndexDefs.Update;
        { 利用可能なインデックスの一覧を入力する}
        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.
```

レポートでのカスタム関数の使用

FastReport は、レポートデザインに使用するための、多数の組み込み標準関数を備えています。また、カスタム関数を記述して使用することもできます。関数は、FastReport に含まれるFastScript ライブラリインターフェイスを使用して追加されます（FastScript の詳細については、付属するライブラリマニュアルを参照してください）。

どのようにしてFastReport にプロシージャや関数を追加できるのかを見てみましょう。パラメーターの個数や型は関数によって異なります。Set 型とRecord 型のパラメーターは、FastScript ではサポートされていないため、もっと簡単な型を使用して実装する必要があります。たとえば、TRect は4 つの整数（X0, Y0, X1, Y1）として渡すことができます。さまざまなパラメーターを持つ関数の使用については、FastScript ドキュメントに詳しく記載されています。

Delphi のフォームで、関数およびプロシージャとそのコードを宣言します。

```
function TForm1.MyFunc(s: String; i: Integer): Boolean;
begin
// 必要なロジック
end;

procedure TForm1.MyProc(s: String);
begin
// 必要なロジック
end;
```

レポートコンポーネントの"onUser" 関数ハンドラーを作成します。

```
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;
```

レポートコンポーネントのAdd メソッドを使用して、関数一覧に追加します（通常、Delphi フォームのonCreate またはonShow イベントで行います）。

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

これで、追加された関数をレポートスクリプトで使用することができ、TfrxMemoView 型のオブジェクトによって参照できるようになりました。また、この関数は、［データツリー］ウィンドウの［関数］タブにも表示されます。このタブでは、関数はカテゴリ別に分かれており、どれか選択すると、その関数に関するヒントがタブの下部ペインに表示されます。

関数を別々のカテゴリに登録し、説明的なヒントを表示するように、上のコードサンプルを変更します。

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

追加した関数が"My Functions" カテゴリの下に現れるようになります。

関数を既存のカテゴリに登録するには、次のカテゴリ名のいずれかを使用します。

- 'ctString'　文字列関数
- 'ctDate'　日付と時刻関数
- 'ctConv'　変換関数
- 'ctFormat'　書式

- 'ctMath' 数学関数
- 'ctOther' その他の関数

カテゴリ名が空白の場合、その関数は関数ツリーのルート下に置かれます。多数の関数を追加する場合は、すべてのロジックを別個のライブラリユニットに置くことをお勧めします。例を示します。

```
unit myfunctions;

interface

implementation

uses SysUtils, Classes, fs_iinterpreter;
        // ここに、他の外部ライブラリへの参照を追加することもできます

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
// 必要なロジック
end;

procedure MyProc(s: String);
begin
// 必要なロジック
end;


{ TFunctions }

constructor TFunctions.Create;
begin
  inherited Create(AScript);
  with AScript do
    AddMethod('function MyFunc(s: String; i: Integer): Boolean', CallMethod,
          'My functions', ' MyFunc function always returns True');
    AddMethod('procedure MyProc(s: String)', CallMethod,
          'My functions', ' MyProc procedure does not do anything'');
  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.
```

ファイルを拡張子 .pas で保存し、そのファイルへの参照をDelphi プロジェクトのフォームのuses 節で追加します。そうすると、すべてのカスタム関数が、任意のレポートコンポーネントで使用できるようになります。これらの関数を各 "TfrxReport" に追加するためにコードを書く必要はなく、各レポートコンポーネントの "onUser" 関数ハンドラーに追加のコードを書く必要もありません。

カスタム ウィザードの記述

カスタム ウィザードを使用して、FastReport の機能を拡張することができます。たとえば、FastReport には、『ファイル | 新規作成』メニュー項目から呼び出される標準の『レポート ウィザード』が含まれています。

FastReport でサポートされるウィザードは 2 種類あります。1 種類目としては、既に述べた、『ファイル | 新規作成』メニュー項目から呼び出されるウィザードが挙げられます。2 種類目としては、『ウィザード』ツール バーから呼び出すことができるウィザードが挙げられます。

あらゆるウィザードの基本クラスは、frxClass ファイルに定義されている**TfrxCustomWizard** です。

```
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;
```

独自のウィザードを記述するには、このクラスから継承して、少なくとも GetDescription メソッドと Execute メソッドをオーバーライドします。最初のメソッドは、ウィザード名を返します。2 つめのメソッドは、ウィザードの実行中に呼び出されます。これは、ウィザードが正常に完了し、レポートに変更を加えた場合には、"True" を返す必要があります。ウィザードを実行している間、デザイナーやレポートのメソッドとプロパティには、通常どおり Designer プロパティおよび Report プロパティを使用してアクセスすることができます。

ウィザードの登録と削除は、frxDsgnIntf ファイルに定義されているプロシージャを使用して実行されます。

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

登録では、ウィザードのクラス名とその図、そして、ウィザードを『ウィザード』ツール バーに配置するかどうかを提供します。ウィザードをツール バーに配置する場合は、ButtonBmp のサイズを 16x16 ピクセルまたは 32x32 ピクセルにする必要があります。

『ファイル | 新規作成』メニュー項目からアクセスされる基本ウィザードを見てみましょう。新しいページをレポートに追加します。

```
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
  { デザイナー内のすべての描画をロックする }
  Designer.Lock;

  { レポートで新しいページを作成する }
  Page := TfrxReportPage.Create(Report);
  { ページの一意の名前を作成する }
  Page.CreateUniqueName;
```

```
  { 用紙サイズと印刷の向きを既定に設定する}
  Page.SetDefaults;

  { レポートのページを更新し、最後に追加されたページにフォーカスを切り替える}
  Designer.ReloadPages(Report.PagesCount - 1);
end;

var
  Bmp: TBitmap;

initialization
  Bmp := TBitmap.Create;
  { リソースから図を読み込む。当然ながら、それは既に存在している必要があります}
  Bmp.LoadFromResourceName(hInstance, 'frxMyWizard');
  frxWizards.Register(TfrxMyWizard, Bmp);

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

end.
```