dot *software* ltd
39 High Street
Weston
Bath
B&NES BA1 4BX
ENGLAND

Tel +44 (0)1225 481100
Fax +44 (0)1225 482100
web http://www.dotsoftware.co.uk
email info@dotsoftware.co.uk
Registered No. 3387287

Article

Exploring HyperGraphics – Interactive Diagramming Capabilities

Article: 2, issue 2, 4 September 2003
Software Version: Delphi 6.0, HyperGraphics 2.220
Written by: Dom Jenkins

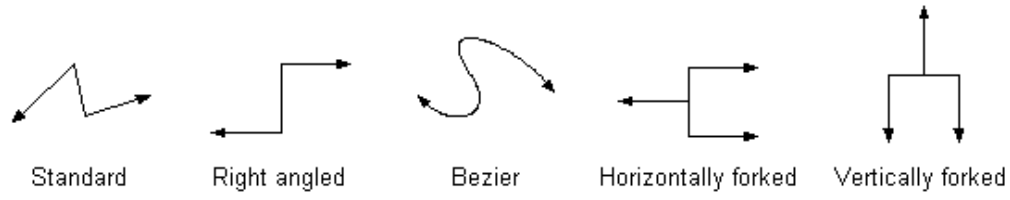
Getting Started

HyperGraphics is a new interactive graphics library ideally suited to engineering applications, particularly those requiring more dynamic graphics capabilities. It also has a number features that make it an ideal choice for interactive graphics and diagramming applications. By this I mean applications such as Microsoft Visio, or a UML designer, which allow a user to manipulate objects graphically and link them together.

Some of the diagramming features of **HyperGraphics** include:

1. *Plug* and *Socket* components allowing objects to be linked visually
2. *Connecting* and *Connected* events allowing code to accept or decline connections and maintain the links "behind" the diagram
3. In-line editing of text objects
4. Built-in serialisation of all objects and links, allowing easy "cut and paste", and loading and saving of files.

Many existing **HyperGraphics** objects incorporate socket components already, but if not, these can be added to any object as child components, as we shall see. There are also a number of versatile *Connectors* (lines with plug objects):



Different types of connectors

New connectors can be derived from any of these, and again any number of plug objects can be added.

With these features in mind we can build a simple example…

Creating a Database Visually

Wouldn't it be nice to create a database visually? For me, this is definitely preferable to using lots of dialogs anyway, namely because you can immediately see the tables and how they link together. So, with this in mind, how about something that allows us to:

- Add tables
- Add fields to the tables
- Set a field to be the primary key
- Set constraints (link primary fields to foreign fields)
- Save the whole lot to a file
- Generate the database (or at least something that could be used to do this!).

To do this, first we need to start a new Delphi project and drop an HGPanel object (from the **HG** tab of the component palette) onto a form, setting `Align:=alClient`. Now add an HGView using the designer (right click the HGPanel and select "HyperGraphics designer…"). As the view will be the parent to our diagram, set it to occupy the whole of the client area, i.e. normalised units for position and size, positioned at (0,0) with size (1,1). Figure 1 shows how your form should look.

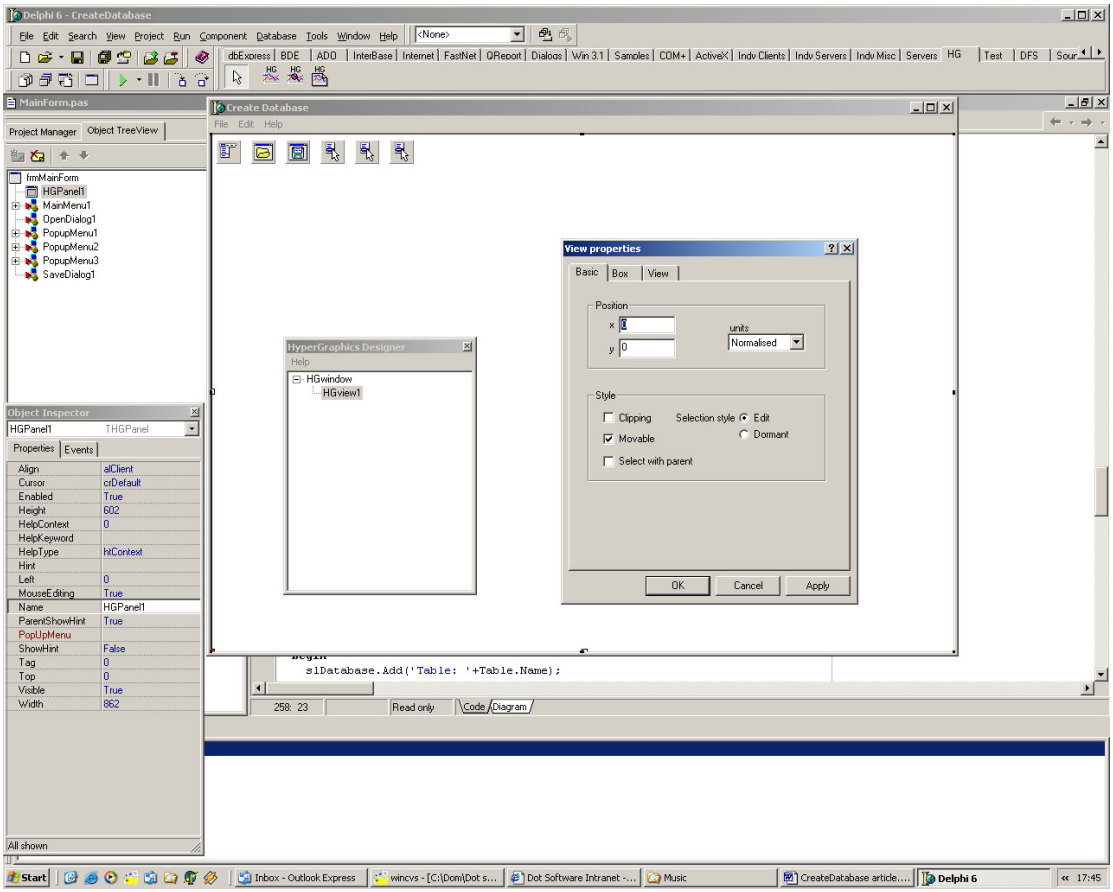


Figure 1 HyperGraphics designer

Now we need some visual objects to represent the tables and connectors, and we can derive them from a table (`THGTable`) and a right-angled connector (`THGRAConnector`), respectively.

Multi-line and multi-column tables

The `THGTable` object supports multi-line and multi-column tables, with each cell being a `THGCell` object that positions its child controls and sizes itself to surround them (a bit like the way an HTML table sizes itself to its contents). We can derive a new component from a `THGTable`, and use a text object (`THGText`) to display the table name.

Displaying the field names is a bit trickier as we want to have an indication that the field is a primary key. For this we'll use a picture object with a caption object positioned to its left, as this will draw itself without any extra coding.

Listing 1 – Creating the table shows the code required to create the table and add fields. Note that the "perimeter socket" [a special socket that allows plugs to connect anywhere along the perimeter of its parent, rather than at a specific location] is freed in the constructor. This is because it is already part of the `THGTable`, but as we do not need this behaviour, we simply delete the object. You can also see that we have added two socket (`THGSocket`) components to the cell, one on each side of the cell. These will be used with the connector object to set the database constraints. The object hierarchy is as follows:

```
THGDatabaseTable
        THGCell
                THGText – used for the table name
        THGCell
                THGSocket – used to connect fields together
                THGSocket
                THGPicture – used to indicate a primary key field
```

THGCaption – used for the field name

THGCell

…

…

Now we have created the object, we can add some properties to manipulate the fields, set the primary key and return the selected field (if there is one). Note that a user can select the child components of an object by holding down the control key while clicking the parent component. We can make use of this functionality to let the user select an individual field in a table. This can be used to provide a different pop-up menu in the main form. To set a field as the primary key we set the picture object's `ImageObj` to a pre-loaded bitmap. See Listing 2 – Setting a primary key.

We must also remember to register our new object with **HyperGraphics**, as we need to use the built-in serialisation to save our diagram:

```
  // Register with HG (only used for serialisation)
  RegisterHGclass( '','Database Table', TDatabaseTable);
```

We must also restore any references to **HyperGraphics** objects after serialisation, as our constructor will not be called (because **HyperGraphics** has already created our table and all its children from information in the serialisation file). The post serialise event is called after serialisation has been completed, and this is the place to do it:

```
procedure TDatabaseTable.PostSerialiseEvent;
begin
  // get references to objects created in the constructor
  FText:=FindObj('Title') as THGText;
end;
```

Finally we want to stop a connection being made between two sockets on the same field (remember that each field in the table has two sockets, one on each side). To do this, we can add a utility method that can be called from the connector when the user is attempting to connect two fields:

```
function TDatabaseTable.CanConnect(Socket1, Socket2: THGSocket): boolean;
begin
  // check that both sockets don't belong to the same field
  Result:=not (GetFieldInxFromSocket(Socket1)=GetFieldInxFromSocket(Socket2));
end;
```

Connecting tables

The `THGRAConnector` object already has the behaviour we need, but we must also make sure that it can only connect to a table, and that within that table, the `THGRAConnector` never reconnects a field with itself (i.e. connections should only be made between two different fields). To do this, we override the methods that are called when the user drops a plug on a socket:

```
function IsValidConnectEvent( id: longint; psocket: THGsocket; cm: HGconnectmode
): boolean; override;
procedure ConnectEvent( id: longint; psocket: THGsocket; cm: HGconnectmode );
override;
```

The first method is used to control the making or breaking of a connection. If `True` is returned, the user is given visual feedback that the plug can be "dropped", and the connection can be made (and broken). The second method can be used to store information after the connection has been made or broken, and we can use it to set the name of the connector when both ends are connected:

```
procedure TFieldConnector.ConnectEvent(id: Integer; psocket: THGsocket;
  cm: HGconnectmode);
var
  PrimaryTable,ForeignTable: TDatabaseTable;
  PrimaryInx,ForeignInx: integer;
```

```
begin
  inherited;
  // reset the name
  GetConnectedTables(PrimaryTable,ForeignTable,PrimaryInx,ForeignInx);
  if Assigned(PrimaryTable) and Assigned(ForeignTable) then
    Name:=ForeignTable.Name+'_'+PrimaryTable.Name;
end;
```

Main Form

We can start by implementing the opening and saving of files. The serialisation built into **HyperGraphics** will write the entire object hierarchy to a buffer, so we will use a `TMemoryStream` to write this to a file:

```
procedure TfrmMainForm.Saveas1Click(Sender: TObject);
var
  msFigure: TMemoryStream;
  iSize: integer;
begin
  if SaveDialog1.Execute then
  begin
    msFigure:=TMemoryStream.Create;
    try
      // write out the view data
      iSize:=FView.SaveAllToBuffer(msFigure.Memory,0);
      msFigure.SetSize(iSize);
      FView.SaveAllToBuffer(msFigure.Memory,iSize);
      msFigure.SaveToFile(SaveDialog1.FileName);
    finally
      msFigure.Free;
    end;
  end;
end;
```

Loading a file is a bit more complicated as we must replace any existing diagram. This is achieved by simply destroying the `THGView` (as it is the parent of the diagram) and replacing it with the diagram we saved in the file:

```
procedure TfrmMainForm.Open1Click(Sender: TObject);
var
  msFigure: TMemoryStream;
begin
  if OpenDialog1.Execute then
  begin
    msFigure:=TMemoryStream.Create;
    try
      msFigure.LoadFromFile(OpenDialog1.Filename);
      // display it in the window (view is already in the file)
      FView.Free;
      FView:=HGPanel1.HGWindow.
        RestoreFromBuffer(PByte(msFigure.Memory),msFigure.Size) as THGView;
    finally
      msFigure.Free;
    end;
    HGPanel1.Invalidate;  // redraw
  end;
end;
```

Now we can implement the menu items for adding our table and connector objects. This is fairly trivial as we can use the `AddPositionalObject` method and let the user position the object with the mouse:

```
procedure TfrmMainForm.Addtable1Click(Sender: TObject);
var
  Table: TDatabaseTable;
begin
  Table:=TDatabaseTable.Create(FView);
```

```
  if Table.AddPositionalObject=-1 then Table.Free;
end;
```

Next we would like the user to be able to select items, and the program to track what they have selected so that pop-up menus can be context sensitive. If we set the `HGPanel MouseEditing` property to `True`, **HyperGraphics** will control the selection of objects with the mouse. For this example, the way that **HyperGraphics** does this is exactly what we want, but we could write our own mouse events to control the selection ourselves. Next we set event handlers for the `OnObjSelected` and `OnObjDeSelected` events so we can track the user's selections and respond correctly to right-click events. Note that the user can select a child of an object (and this is the way they will select a field in a table), so we search the selected object's parents to see if they are a `THGDatabaseTable` or `TFieldConnector`:

```
procedure TfrmMainForm.HGPanel1ObjSelected(Sender: TObject);
var
  Obj: THGDrawingObject;
begin
  FCurrentTable:=nil;
  FCurrentConnector:=nil;
  Obj:=Sender as THGDrawingObject;
  // search parents of selected object as user can select a child of a table
  // or connector
  while Assigned(Obj) do
  begin
    if Obj=FView then
      Break
    else if Obj is TDatabaseTable then
    begin
      FCurrentTable:=TDatabaseTable(Obj);
      Break;
    end
    else if Obj is TFieldConnector then
    begin
      FCurrentConnector:=TFieldConnector(Obj);
      Break;
    end;
    Obj:=Obj.Parent;
  end;
end;
```

The final (most important) thing to add is the output of the tables, fields and constraints to a text file. The output is generated from two loops, one for tables and the other for connectors. See Listing 3 – Output of database structure. The loop for tables simply outputs the table name and all the field names, with an indicator for primary fields. The second loop must find both tables for each connector (if it is not connected to two tables, then we are not interested in it) and output the table name and field name for each. To simplify the code and remove any dependence on the internal structure of our components, the `GetConnectedTables` method of the connector is used, which then uses the `GetTableFromSocket` and `GetFieldInxFromSocket` methods from the table. See Listing 4 – Getting the constraint information. Note that the output is a simple list. Converting it to SQL is left as an exercise for the reader!

So what does it look like? Figure 2 shows an example database with five tables, and this is the text file it produces:

```
Table: Projects
  Primary key ID
Table: Settings
  DefaultProjectID
Table: ProjectUsers
  Primary key ID
  ProjectID
  UserID
Table: Users
  Primary key ID
```

```
Table: Sessions
  Primary key ID
  ProjectID
  UserID
Constraint: Settings_Projects
  Primary Projects ID Foreign Settings DefaultProjectID
Constraint: ProjectUsers_Projects
  Primary Projects ID Foreign ProjectUsers UserID
Constraint: Sessions_Projects
  Primary Projects ID Foreign Sessions ProjectID
Constraint: Sessions_Users
  Primary Users ID Foreign Sessions UserID
Constraint: ProjectUsers_Users
  Primary Users ID Foreign ProjectUsers UserID
```

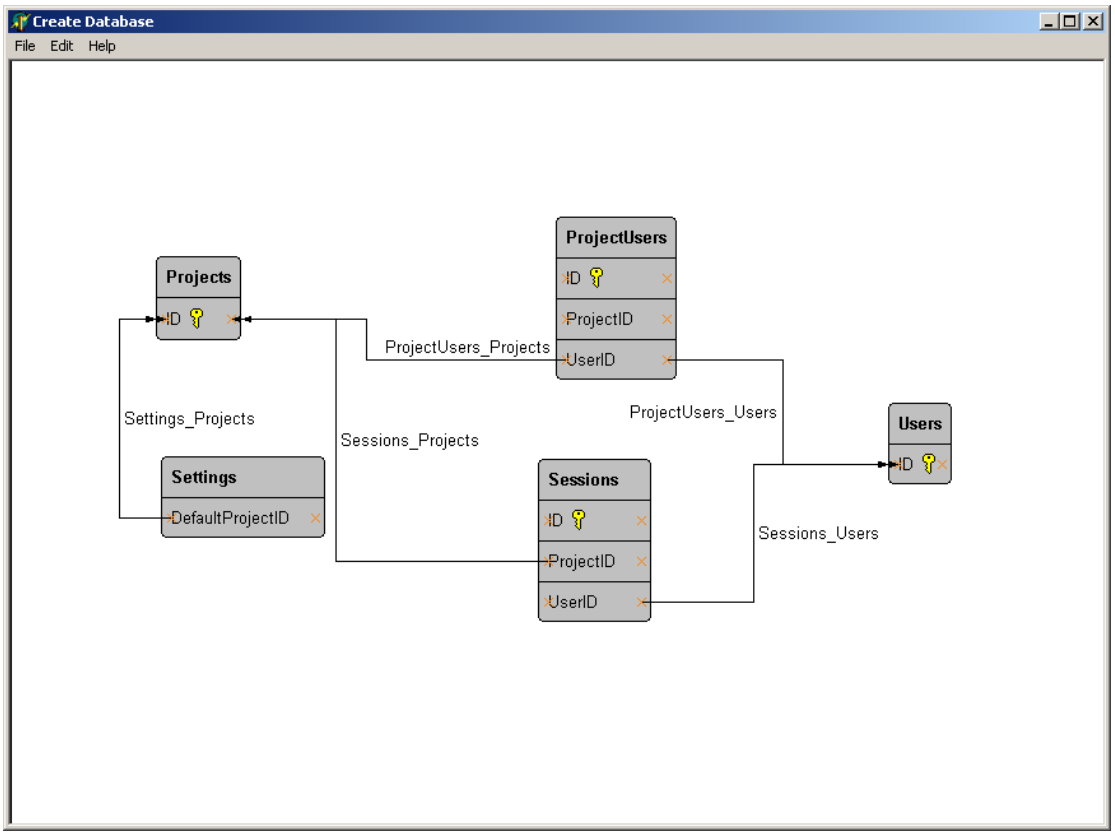


Figure 2 - Example database diagram

Conclusions

What have we achieved? Well, we've delved into the diagramming functionality of **HyperGraphics** and built an application for capturing the structure of a database. We've learnt how to serialise our objects and how to link them together, and we've also made good use of the functionality built into **HyperGraphics** for selecting and drawing objects.

There are plenty of additional features that we could have added using **HyperGraphics**, such as multiple-object cut and paste, and object selection using a rubber band box. Yet more features, such as real-time display of data and creating custom objects, are left to other examples.

Specialist software development does not have to mean writing everything from the ground up. For skilful developers needing re-usable graphical components that offer real power and flexibility, **HyperGraphics** is an essential tool.

Listing 1 – Creating the table

```
constructor TDatabaseTable.Create(par: THGdrawingobject);
```

```
var
  Cell: THGCell;
begin
  inherited Create(par);
  BorderObj.ChildSocket.Free; // don't want the perimeter socket
  // set rows/cols
  Ncols:=1;
  Nrows:=1;
  // add title to row 0
  Cell:=Self.FindCell(0,0);
  Cell.Cellhoralignment:=HG_LEFT;
  FText:=THGText.Create(Cell);
  FText.Name:='Title';
  FText.Font.Style:=[fsBold];
  Roundcorners:=True;
  Borderobj.Fillcolour:=clSilver;  // match to primary key bitmap background
  Name:='NewTable';
  // add first field
  AddField('ID');
end;

function TDatabaseTable.AddField(Name: string):integer;
var
  Caption: THGCaption;
  Cell: THGCell;
  Picture: THGPicture;
  Socket: THGSocket;
begin
  // add a cell
  NRows:=NRows+1;
  Cell:=FindCell(NRows-1,0);
  // set cell properties
  Cell.Cellhoralignment:=HG_LEFT;
  // add two sockets, one at each side
  Socket:=THGSocket.CreateSocket(Cell,1,0.5);
  Socket.Movable:=False;
  Socket:=THGSocket.CreateSocket(Cell,3,0.5);
  Socket.Movable:=False;
  // add a picture object for the primary key bitmap
  Picture:=THGPicture.Create(Cell);
  Picture.Ownresources:=False;
  Picture.Imageobj:=Primary;  // sets the size of the picture
  Picture.ChangeSizeunits(HG_CM); // recalculate size to cm for printer
  // add a caption to the picture object as it can position itself automatically
  Caption:=THGCaption.Create(Picture);
  Caption.Alignment:=HG_CAPLEFT;
  Caption.Textstring:=Name;
  Result:=NRows-2;
  PrimaryKey[Result]:=False;  // default to not a primary key
end;
```

Listing 2 – Setting a primary key

```
procedure TDatabaseTable.SetPrimaryKey(Inx: integer;Value: boolean);
begin
  if Value then
    GetPictureFromField(FindField(Inx)).ImageObj:=Primary
  else
  begin
    with FindField(Inx).Parent as THGPicture do ImageObj:=nil;
  end;
end;

initialization
  // bitmap for primary key
  Primary:=THGBitmap.CreateBitmap('Primary.bmp');
  // Register with HG (only used for serialisation)
  RegisterHGclass('','Database Table', TDatabaseTable);
```

```
finalization
  Primary.Free;
end.
```

Listing 3 – Output of database structure

```
procedure TfrmMainForm.Outputdatabase1Click(Sender: TObject);
var
  slDatabase: TStringList;
  // local procedure
  procedure OutputTable(Table: TDatabaseTable);
  var
    i: integer;
    stTemp: string;
  begin
    slDatabase.Add('Table: '+Table.Name);
    for i:=0 to Table.Fields-1 do
    begin
      if Table.PrimaryKey[i] then
        stTemp:='  Primary key '
      else
        stTemp:='  ';
      slDatabase.Add(stTemp+Table.Field[i]);
    end;
  end;
  // local procedure
  procedure OutputConstraints(Connector: TFieldConnector);
  var
    PrimaryTable,ForeignTable: TDatabaseTable;
    PrimaryInx,ForeignInx: integer;
  begin

Connector.GetConnectedTables(PrimaryTable,ForeignTable,PrimaryInx,ForeignInx);
    if Assigned(PrimaryTable) and Assigned(ForeignTable) then
    begin
      slDatabase.Add('Constraint: '+Connector.Name);
      slDatabase.Add('  Primary '+PrimaryTable.Name+'
'+PrimaryTable.Field[PrimaryInx]+
        ' Foreign '+ForeignTable.Name+' '+ForeignTable.Field[ForeignInx]);
    end;
  end;
var
  Obj: THGDrawingObject;
begin
  // write out database as a text file (could be translated to SQL)
  slDatabase:=TStringList.Create;
  try
    // loop round tables
    Obj:=FView.Child;
    while Assigned(Obj) do
    begin
      // output fields
      if Obj is TDatabaseTable then OutputTable(TDatabaseTable(Obj));
      Obj:=Obj.Next;
    end;
    // loop round tables
    Obj:=FView.Child;
    while Assigned(Obj) do
    begin
      // output constraints
      if Obj is TFieldConnector then OutputConstraints(TFieldConnector(Obj));
      Obj:=Obj.Next;
    end;
    slDatabase.SaveToFile('Database.txt');
  finally
    slDatabase.Free;
  end;
end;
```

Listing 4 – Getting the constraint information

```
procedure TFieldConnector.GetConnectedTables(var PrimaryTable,ForeignTable:
TDatabaseTable;
  var PrimaryInx,ForeignInx: integer);
var
  Plug: THGPlug;
begin
  PrimaryTable:=nil;
  ForeignTable:=nil;
  PrimaryInx:=-1;
  ForeignInx:=-1;
  Plug:=ChildPlug;
  if Assigned(Plug.Socketobj) then
  begin
    ForeignTable:=TDatabaseTable.GetTableFromSocket(Plug.Socketobj);
    ForeignInx:=ForeignTable.GetFieldInxFromSocket(Plug.Socketobj);
  end;
  Plug:=GetOtherPlug(Plug);
  if Assigned(Plug.Socketobj) then
  begin
    PrimaryTable:=TDatabaseTable.GetTableFromSocket(Plug.Socketobj);
    PrimaryInx:=PrimaryTable.GetFieldInxFromSocket(Plug.Socketobj);
  end;
end;

function TFieldConnector.GetOtherPlug(Plug: THGPlug):THGPlug;
var
  OtherPlug: THGPlug;
begin
  Result:=nil;
  OtherPlug:=ChildPlug;
  while Assigned(OtherPlug) do
  begin
    if OtherPlug<>Plug then
    begin
      Result:=OtherPlug;
      Break;
    end;
    OtherPlug:=OtherPlug.NextPlug;
  end;
end;

function TDatabaseTable.GetFieldInxFromSocket(Socket: THGSocket): integer;
var
  i: integer;
  Cell: THGCell;
begin
  Result:=-1;
  for i:=0 to Fields-1 do
  begin
    Cell:=GetCellFromField(FindField(i));
    if HasChildSocket(Cell,Socket) then
    begin
      Result:=i;
      Break;
    end;
  end;
end;

class function TDatabaseTable.GetTableFromSocket(Socket: THGSocket):
TDatabaseTable;
begin
  if not (Socket.Parent.Parent is TDatabaseTable) then
    Result:=nil
  else
    Result:=TDatabaseTable(Socket.Parent.Parent);
```

```
end;
```