Overview

TMS Diagram Studio is a set of components for Delphi and C++ Builder to easily add feature-rich and user friendly diagramming, flowcharting & graphing capabilities to your applications.

Diagram Studio provides TatDiagram component, a panel-like control where user can build diagrams by inserting blocks, lines and linked them together. TDiagramToolBar component is also provided to allow easy and fast editing of diagram component with no line of code.

The blocks in diagram can be customized by user by changing dozens of available properties. User can change shapes of blocks, shadow, bitmaps, among other features. Blocks can be rotated and resized.

Diagram Studio provides an open architecture to allow users to building their own blocks by inheriting from TCustomDiagramBlock class and registering then by using RegisterDControl procedure.

Among the many features that make Diagram Studio the ultimate diagramming and flowcharting component set are:

- Diagram editing behaviour similar to standard diagramming applications;
- Support clipboard operations;
- Block gradient, shadow and bitmap;
- Diagram navigator control for a full overview of diagram;
- Full block customization: pen, brush, color, selection color, minimum width and height;
- Block text customization: horizontal and vertical alignment, font, word wrap, clipping;
- Customizable link points in blocks;
- Inplace block text editing;
- Full line (link) customization: pen, source arrow shape, target arrow shape;
- Arc & bezier lines, polygon objects;
- Block rotation supported (including text, bitmap, metafiles and gradient);
- TDiagramToolBar component for easy diagram editing with no line of code;
- Several selector components (pen width, pen style, font, shadow, gradient) for easy editing of diagram;
- Design-time diagram editing support;
- Diagram snap grid;
- Diagram background image (stretched or tiled);
- Diagram rulers;
- Diagram printing and previewing;
- Saving/Loading diagram to/from file and stream;

- Diagram zoom in/out;
- Panning;
- Support for different layers;
- Support for node support in connected blocks and block hiding with node collaps/expand;
- Open architecture for building custom blocks and lines inherited from base classes;
- Helper class TBlockDrawer for easy custom drawing on custom blocks;
- Lots of ready-to-use TAction descendants available for specific diagram operations: clipboard operations, object deletion and inserting, zooming, and more.

Rebuilding Packages

If for any reason you want to rebuild source code, you should do it using the "Packages Rebuild Tool" utility that is installed. There is an icon for it in the Start Menu.

Just run the utility, select the Delphi versions you want the packages to be rebuilt for, and click "Install".

If you are using Delphi XE and up, you can also rebuild the packages manually by opening the dpk/dproj file in Delphi/Rad Studio IDE.

Do NOT manually recompile packages if you use Delphi 2010 or lower. In this case always use the rebuild tool.

In this section:

Getting Started

Using the TatDiagram component: overview of key features.

Working with Diagram Studio programmatically

How to programmatically handle diagrams and some examples.

Live Diagram

The "executable" version of TatDiagram.

Getting Started

Quick start - using diagram and toolbar components

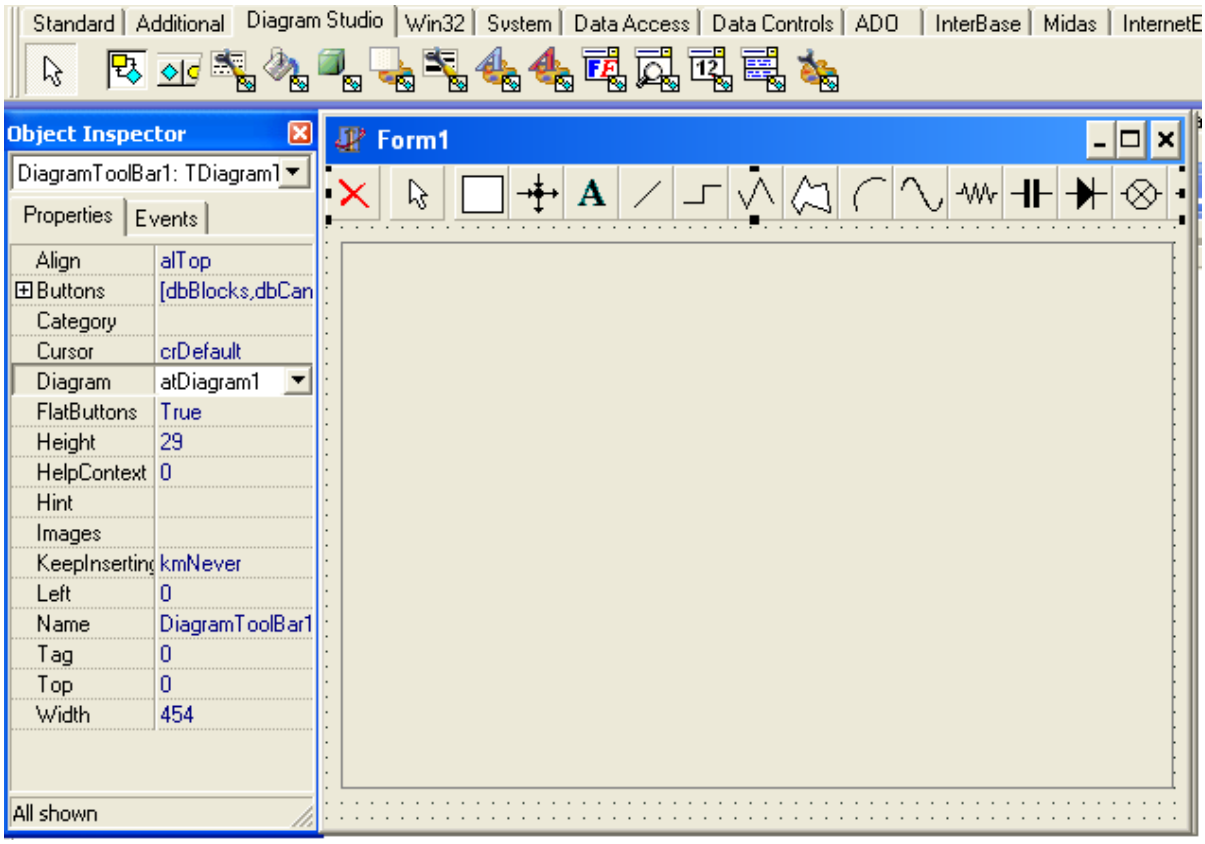
The main component in Diagram Studio is the *TatDiagram* component. The TatDiagram component is the visual control which holds and display the diagram, and allow editing of it.

Another helpful component is the *TDiagramToolBar*. Although TatDiagram component does not require the TDiagramToolBar component to work, the toolbar component is very useful to quickly get started, since it allows easy insertion of blocks diagram without requiring a single line of code.

For Delphi 2005 and higher, there is a new component named *TDiagramButtons*. It has the same purpose of TDiagramToolbar, however it's inherited from TCategoryButtons and has a more modern look. There is a screenshot of new TDiagramButtons at the end of this topic.

Getting started with Diagram Studio is as quick as dropping both TatDiagram and TDiagramToolBar (or TDiagramButtons) components in a form, and linking the toolbar to the diagram, by setting the Diagram property of the TDiagramToolBar component to make reference to the TatDiagram component.

With this setting, and with no line of code, you will have a running application with already provides diagram capabilities. Just choose an object from the toolbar and click the diagram to insert objects, and you can start editing, resizing, moving and deleting objects.



TDiagramButtons component which provides a toolbar with a more modern look:

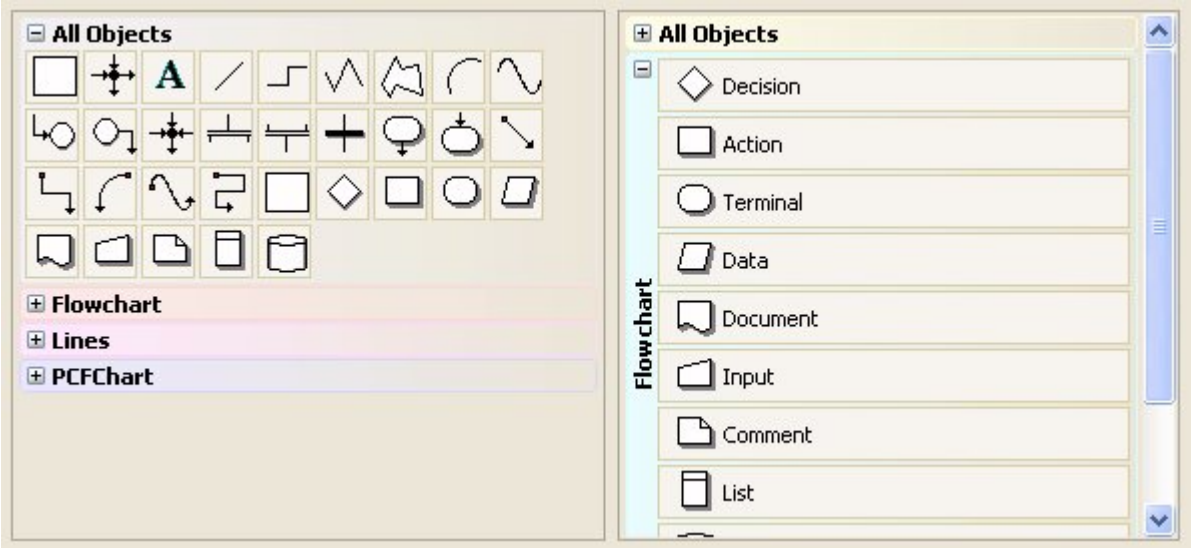


Diagram objects

There are two types of diagram objects: blocks and lines. All blocks descend from TCustomDiagramBlock class, and all lines descend from TCustomDiagramLine block. TDiagramControl class is the ancestor for both classes.

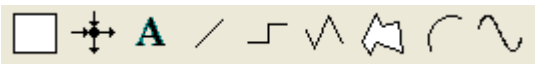
`TDiagramControl`

→ `TCustomDiagramBlock`

→ `TCustomDiagramLine`

Basic objects

By default, Diagram Studio provides some basic blocks and lines, which descend from both TCustomDiagramBlock and TCustomDiagramLine classes. These basic blocks and lines are the ones displayed in the TDiagramToolBar component:



From the left to right, the basic objects are:

- *TDiagramBlock*: a basic diagram block. Descends from TCustomDiagramBlock and published all of its properties, so this block is full-features, allowing setting of colors, shadows, gradient, pictures, linkpoints, shapes, and so on.
- *TDiagramLineJoin*: a TCustomDiagramBlock descendant which is just a link point container. You can attach lines to this block (you can attach lines to any block also, see "linking blocks").
- *TTextBlock*: a TCustomDiagramBlock descendant which sets itself some properties to look more like a text block (TDiagramBlock also provides text capabilities, as long all diagram obejcts).

- *TDiagramLine*: a TCustomDiagramLine descendant, it's just a single line (one segment).
- *TDiagramSideLine*: a TCustomDiagramLine descendant, it's a line with several perpendicular segments. The number of segments are calculated automatically.
- *TDiagramPolyLine*: a TCustomDiagramLine descendant, it's a line with several segments. The end-user draws the desired segments.
- *TPolygonBlock*: a TCustomDiagramBlock descendant, it's a polygon block with several sides. The end-user draws the desired sides of the polygon.
- *TDiagramArc*: a TCustomDiagramLine descendant, it's a curved line.
- TDiagramBezier*: a TCustomDiagramLine descendant, it's a bezier-like line.

Extra objects

Diagram Studio provides extra objects for usage in diagram. These extra objects are included for instant usage, but also as samples on how to extend Diagram Studio by creating more diagram blocks. The extra objects in Diagram Studio are grouped in the following categories:

- *Flowchart blocks*: Provides some basic blocks for flowcharting diagrams, like action, start, end, decision, etc.
- *Electric blocks*: Provides some basic blocks for eletricity diagrams, like resistor, capacitor, voltage source, ground, etc.
- *Arrow blocks*: Provides some arrow-shaped blocks.

All extra objects are available at design-time. To make them available at runtime, just include the respective unit in the uses clause of any unit of your project. For example, if you want to use flowchart blocks, include the FlowChartBlocks.pas unit to your project.

Flowchart blocks

To use flowchart blocks at runtime, include the `FlowchartBlocks.pas` unit to your Delphi project. The following blocks are provided:

- TFlowDecisionBlock
- TFlowActionBlock
- TFlowTerminalBlock
- TFlowDataBlock
- TFlowDocumentBlock
- TFlowInputBlock
- TFlowCommentBlock
- TFlowListBlock
- TDatabaseBlock

Electric blocks

To use electric blocks at runtime, include the `ElectricBlocks.pas` and the `ElectricBlocks2.pas` units to your Delphi project. The following blocks are provided:

In `ElectricBlock` unit:

- TResistorBlock
- TCapacitorBlock
- TDiodeBlock
- TLampBlock

In `ElectricBlocks2` unit:

- TZenerDiodeBlock
- TComparatorBlock
- TInductorBlock
- TNonLinearInductorBlock
- TGroundBlock
- TNPNTransistorBlock
- TPNPTransistorBlock
- TPTIGBTBlock
- TNPTIGBTBlock
- TPINBlock
- TThyristorBlock
- TMosfetBlock
- TSwitchBlock
- TDuoCoilXFormBlock
- TTriCoilXFormBlock
- TDCVoltageSourceBlock
- TDCCurrentSourceBlock

Arrow blocks

To use arrow blocks at runtime, include the `ArrowBlocks.pas` unit to your Delphi project. The following blocks are provided:

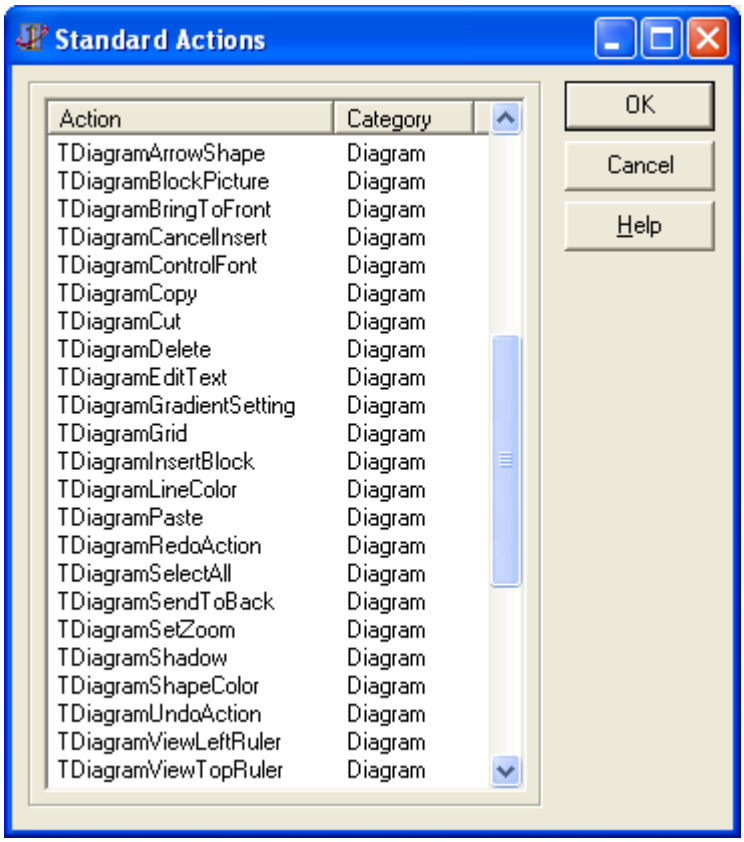
- TStandardArrowBlock
- TDoubleArrowBlock
- TQuadArrowBlock
- TTripleArrowBlock
- TChevronArrowBlock
- TBlockSingleArrowBlock
- TBlockDoubleArrowBlock
- TCornerSingleArrowBlock
- TCornerDoubleArrowBlock

Using predefined diagram actions

Although TatDiagram component is full-featured and very easy to use, it's still hard to build an IDE for editing the diagram. You must provide visual interfaces in your application for many diagram actions, like saving/opening, printing, clipboard operations, inserting, deleting, setting block properties, and so on.

In order to turn this task less difficult to you, Diagram Studio provides some actions (TAction descendants) for some common operations, like clipboard operations for example.

They work similar to any other action descendant available in the Delphi interface: just create a new action at design-time, based on a standard action, then attach that action to a visual control that support actions, commonly buttons or menu items.



The available actions in Diagram Studio are:

- *TDiagramDelete*: Deletes the selected objects in diagram.
- *TDiagramSelectAll*: Selects all objects in diagram.
- *TDiagramInsertBlock*: Put diagram in inserting mode (see Controlling visual editing of diagram). The control to be inserted is defined in the DControlID property, and the KeepInsertingMode defines if the object must be inserted multiple times, only once, or depending on the Shift key (if pressed, multiple times, if not pressed, once).
- *TDiagramCancelInsert*: Cancel inserting objects and bring diagram back to browsing mode.
- *TDiagramEditText*: Start inplace editing of the text of a diagram object.
- *TDiagramCopy*: Copy selected objects to clipboard.
- *TDiagramCut*: Cut selected objects to clipboard.
- *TDiagramPaste*: Paste selected objects from clipboard.
- *TDiagramViewLeftRuler*: Toggle left ruler visible/invisible.
- *TDiagramViewTopRuler*: Toggle top ruler visible/invisible.
- *TDiagramSetZoom*: Set the diagram zoom to the zoom specified by Zoom property.
- *TDiagramGrid*: Toggle snap grid visible/invisible.
- *TDiagramShapeColor*: Opens a color dialog to choose background color of selected blocks.

- *TDiagramLineColor*: Opens a color dialog to choose line color of selected objects.
- *TDiagramBlockPicture*: Opens a dialog to choose a background image/picture for the selected blocks.
- *TDiagramArrowShape*: Sets the arrow shape of the selected lines. The arrow shape to be set is defined in the Shape property. The Source property defines if the arrow to be changed is the source arrow (true) or the target arrow (false).
- *TDiagramSendToBack*: Send selected objects to back.
- *TDiagramBringToFront*: Bring selected objects to front.
- *TDiagramControlFont*: Opens a font dialog for editing the font of selected blocks.
- *TDiagramGradientSetting*: Opens a dialog for editing gradient settings of selected blocks.
- *TDiagramShadow*: Opens a dialog for editing shadow settings of selected blocks.
- *TDiagramUndoAction*: Undo last operation performed in diagram.
- *TDiagramRedoAction*: Redo last operation undoed in diagram.
- *TDiagramAlign*: Perform alignment operations in two or more blocks (align left, align tops, same size, etc.).

Using special selector components for diagram editing

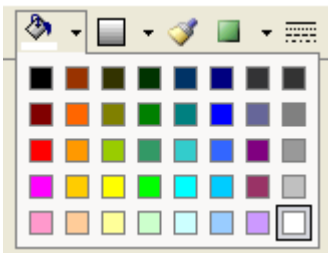
The selector components are another collection of tools to make it easy for the programmer to build a diagram IDE, along with the predefined actions (see Using predefined diagram actions).

The selector components are graphic controls which allow editing some properties of diagram, blocks and lines, just by dropping the control in the form and assigning the Diagram property to make reference for the diagram component.

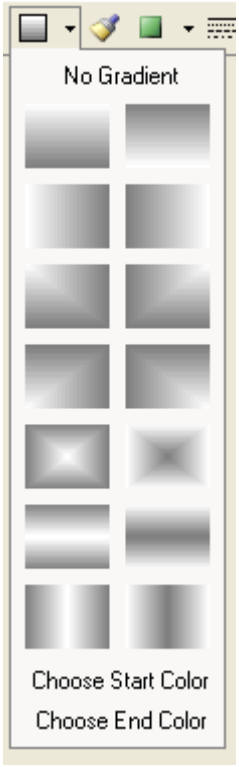
For example, to use the pen style selector, just drop the TDgrPenStyleSelector component in the form, set the Diagram property of the component, and there it is. You can now choose a pen style visually in the control, and all selected objects in diagram will automatically have its pen style set for the one you choose.

The available selectors are:

- *TDgrColorSelector*: Allows selection of background color of selected blocks.

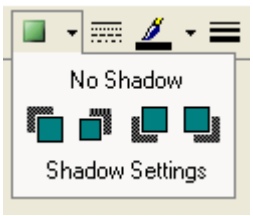


- *TDgrGradientDirectionSelector*: Allows setting the gradient of selected blocks.



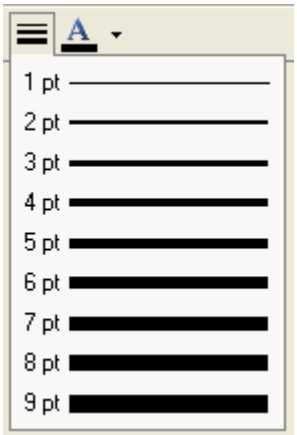
- *TDgrBrushStyleSelector*: Allows setting the brush style of selected blocks.

- *TDgrShadowSelector*: Allows setting the shadow of selected blocks.



- *TDgrPenStyleSelector*: Allows setting the pen style of selected objects.

- *TDgrPenWidthSelector*: Allows setting the pen width of selected objects.



- *TDgrPenColorSelector*: Allows selection of pen color of selected objects.
- *TDgrTextColorSelector*: Allows selection of text color of selected objects.

TDiagramNavigator control

You can use *TDiagramNavigator* visual control to provide a "navigator" to the main diagram. The TDiagramNavigator must be attached to a TatDiagram component through the Diagram property. When this is done, the navigator will display a minimized representation of the whole TatDiagram component, indicating with a rectangle the current area in TatDiagram that is being displayed.

User can move the rectangle to navigate through the main diagram, or can resize the rectangle to zoom in/zoom out. The two controls are always in sync. If the user moves the cursor in the navigator, the diagram will scroll accordingly. If the user changes the diagram by scrolling or editing, the navigator will automatically reflect the changes.

There are several properties in TDiagramNavigator that you can use to change its look and behavior. Some of those properties are:

BackgroundColor: TColor
Specifies the color of the area representing the diagram.

Color: TColor
Specifies the background color of the control (area which does not represent the diagram).

BlockShape: TDiagramNavigatorBlockShape
Specifies how the blocks will be painted in the navigator. Options are:

- nbsOriginal: the block will be painted with the original shape (default);
- nbsRectangle: any block will be painted as a rectangle.

Diagram: TatDiagram
The associated diagram control which will be represented by the navigator.

EnableZoom: boolean
If true, user will be able to resize the highlight cursor rectangle to zoom in/out the diagram. Default is false.

HighlightColor: TColor
The color of the rectangle cursor. Default is red.

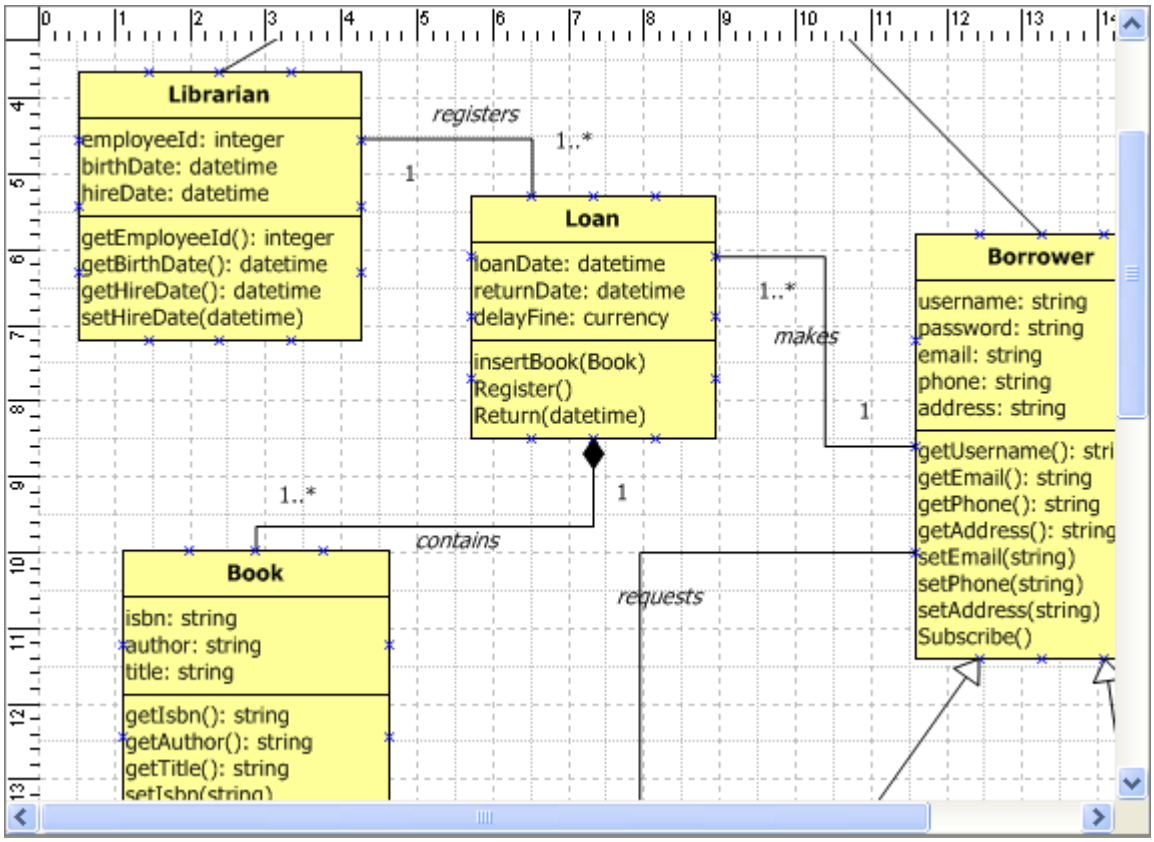
HighlightWidth: integer
The width of the rectangle cursor. Default is 2.

MoveCursor: TCursor
The mouse cursor to be displayed when user is about to move the rectangle cursor.

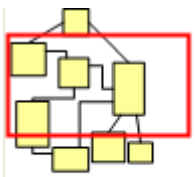
PaintLines: boolean
If false, lines won't be painted in the navigator. Default is true.

ZoomCursor: TCursor
The mouse cursor to be displayed when user is about to resize the rectangle cursor.

Diagram:



Navigator:



Controlling visual editing of diagram

The toolbar component makes it easier to allow visual editing of diagram: end-user just click the toolbar to choose a block, and then click the diagram to edit. However, there are several methods in the diagram component that allows you to control visual editing of diagram.

In summary, TatDiagram implements the concept of "state", so depending on the state of the diagram component, the end-user is able to do something with the current diagram.

Diagram states

The diagram component can be in any of these states:

- *Browsing*: end-user is able to select, move, resize, rotate blocks. This is the default state.
- *Inserting*: end-user is able to insert a block in the diagram.
- *Panning*: end-user is panning the diagram (dragging the diagram).
- *Zooming*: end-user is zooming in/out the diagram.

This are the basic four states. They define what end-user actions do in the diagram, in terms of visual editing.

For example, what happens if the end-user left-clicks the diagram, keeps the left button of the mouse pressed, moves the mouse some pixels and then release the button? It depends on the state of the diagram. If the state is "browsing", then the end-user action will select some diagram objects. If state is inserting, it will insert a control in the diagram. If state is panning, it will move the diagram some pixels in the mouse direction. If state is zooming, it will zoom the diagram.

The default state is browsing. So, to put the diagram in the other three states, you can use some key methods.

Inserting blocks - diagram in insert mode (state)

Call *StartInsertingControl* method to put diagram in insert mode. After calling this method, any mouse operation in diagram like clicking or dragging will insert a new control. The control to be inserted is the first param of StartInsertControl, and it must be identified by its control id. The second param (AKeepInserting) specifies if the diagram should go back to browsing mode after the block is inserted (false), or if the diagram will continue in the inserting mode (true), allowing the same object to be inserted many times.

At any time, the diagram can be put back in the browsing mode by calling *CancelInsertingBlock* method.

```
procedure TForm1.NewBlockButtonClick(Sender: TObject);
begin
  atDiagram1.StartInsertingControl('TDiagramBlock'); //Insert the TDiagramBlock
object once
end;

procedure TForm1.NewTextButtonClick(Sender: TObject);
begin
  atDiagram1.StartInsertingControl('TTextBlock', true); //Insert the TTextBlock
object multiple times
end;

procedure TForm1.CancelInsertButtonClick(Sender: TObject);
begin
  atDiagram1.CancelInsertingBlock;
end;
```

Panning mode (state)

To put the diagram in panning mode, just call *StartPanning* method. To bring diagram back to browsing state, call *CancelPanning*.

```
procedure TForm1.StartPanningButtonClick(Sender: TObject);
begin
  atDiagram1.StartPanning;
end;

procedure TForm1.CancelPanningButtonClick(Sender: TObject);
begin
  atDiagram1.CancelPanning;
end;
```

Zooming mode (state)

To put the diagram in zooming mode, call *StartZooming*. You must inform if the zoom is in (more zoom) or out (less zoom). To bring diagram back to browsing state, call *CancelZooming*.

```
procedure TForm1.ZoomInButtonClick(Sender: TObject);
begin
  atDiagram1.StartZooming(zsZoomIn);
end;

procedure TForm1.ZoomOutButtonClick(Sender: TObject);
begin
  atDiagram1.StartZooming(zsZoomOut);
end;

procedure TForm1.CancelZoomButtonClick(Sender: TObject);
begin
  atDiagram1.CancelZooming;
end;
```

Loading, saving and printing a diagram

Saving and loading diagram is a very common task, and you can do that just by using the methods *SaveToFile* and *LoadFromFile*.

```
atDiagram1.SaveToFile('mydiagram1.dgr');
atDiagram1.LoadFromFile('mydiagram2.dgr');
```

As an option, you can load/save from/to stream. This can be useful for saving diagrams in different storages, like databases for example:

```
atDiagram1.LoadFromStream(AStream);
atDiagram1.SaveToStream(AStream);
```

You can also print the diagram, directly to printer, or ask for a print preview on screen. Use the following methods to do that:

```
atDiagram1.Preview; // preview on screen
atDiagram1.Print; // print the diagram in the printer with current printer
settings
```

Design-time diagram editing

Diagram Studio provides design-time editing of diagram. To do that, right-click the TatDiagram component at design-time, and choose option "Design diagram...". A new window with a toolbar will be displayed and you will be able to use that toolbar to insert and delete objects to/from the diagram.

Full-feature runtime editor

If you want to provide a full-featured runtime diagram editor for your end-user with minimum effort, you can use *TDiagramEditor* component. Just drop a TDiagramEditor component in a form, call its Execute method, and a diagram editor form will be opened with lots of options for diagram editing (menu and toolbars with options for clipboard operations, alignment, formatting, etc.).

The diagram editor uses a TDiagramToolbar (for Delphi 7 and previous versions) or a TDiagramButtons (for Delphi 2005 and higher versions) depending on the Delphi version you're using.

Working with Diagram Studio programmatically

Inserting objects in diagram

To insert objects in diagram, you must:

1. Creates the object, instantiating the class of object.
2. Sets the owner of object as the same owner of the diagram.
3. Sets all other visual properties of the object you might want to change.
4. Sets the diagram property to the diagram component.

Example:

```
MyBlock := TDiagramBlock.Create(atDiagram1.Owner);
with MyBlock do
begin
  Left := 10;
  Top := 10;
  Width := 150;
  Height := 50;
  Diagram := atDiagram1;
end;
```

Removing an object from diagram

To remove an object from the diagram, simply destroy the object. Some examples:

```
MyDiagramBlock1.Free; //Destroys the object MyDiagramBlock1

{Destroy all lines in diagram}
while atDiagram.LinkCount > 0 do
  atDiagram.Links[0].Free;
```

Alternatively, you can remove the currently selected objects in diagram:

```
atDiagram1.DeleteSelecteds;
```

Adding a line between two blocks (linking blocks)

A line start (or end) can be attached to a link point. To link two blocks, you must attach the start of the line to a block linkpoint, and attach the end of line to another block linkpoint. Example:

```
MyLine := TDiagramLine.Create(atDiagram1.Owner);
with MyLine do
begin
  Diagram := atDiagram1;
  SourceLinkPoint.AnchorLink := SomeBlock.LinkPoints[0]; //link start point to
someblock
  TargetLinkPoint.AnchorLink := AnotherBlock.LinkPoints[1]; // link end point to
anotherblock
end;
```

Creating a TDiagramPolyLine: line with multiple points

The Handles property of a diagram line control the position of its intermediate points. That's valid for all line types, including polylines, bezier, side lines, etc:

```
curDataBlock := TDiagramPolyLine.Create(atDiagram1.Owner);
with curDataBlock do
begin
  Handles.Clear;
  Handles.Add.OrPoint := Dot(49,115);
  Handles.Add.OrPoint := Dot(151,179);
  Handles.Add.OrPoint := Dot(191,124);
  Handles.Add.OrPoint := Dot(212,84);
  Diagram := atDiagram1;
end;
```

Using a metafile as the block shape

If you want to use a metafile picture as the block shape, you need to load the metafile to block, and also set the shape to none, so the shape block is not drawn, only the picture. Example:

```
with atDiagram1.Blocks[0] do
begin
  Shape := bsNoShape;
  Picture.LoadFromFile('mymetafile.wmf');
end;
```

Creating linkpoints in a block

To create linkpoints, the LinkPoints collection must be used. Each linkpoint is a collection item. The position of the linkpoint must be indicated as a relative point related to the rect specified by the Drawer.OriginalRect property. By default, OriginalRect is (0, 0, 100, 100), making it easy to define link points positions in terms of % of block width/height.

The following example, from `FlowchartBlocks.pas` unit, shows how to define four link points for the block, in the top, left, right and bottom sides of the block, in the middle of each side segment.

```
procedure TFlowchartBlock.UpdateLinkPoints;
begin
  LinkPoints.Clear;
  with Drawer.OriginalRect do
  begin
    LinkPoints.Add((Right - Left) / 2, Top, aoUp);
    LinkPoints.Add((Right - Left) / 2, Bottom, aoDown);
    LinkPoints.Add(Left, (Bottom - Top) / 2, aoLeft);
    LinkPoints.Add(Right, (Bottom - Top) / 2, aoRight);
  end;
end;
```

Accessing TextCell of a line object

The line objects has at least one text cell defined by default. If you want to access a text cell of a TDiagramLine object, for example, you can use this code:

```
TextCell := LinkLine.TextCells[0];
```

Changing category of blocks in toolbar

Calling *RegisterDControl* for any existing block will re-register it. So it's possible to change the category, or caption for the block. The following code "moves" all the blocks to the "MyBlocks" category, and sets that category in the toolbar.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DiagramToolBar1.Category := 'MyBlocks';

  RegisterDControl(TDiagramBlock, '', 'Simple block', 'MyBlocks');
  RegisterDControl(TDiagramLine, '', 'Line', 'MyBlocks');
  RegisterDControl(TTextBlock, '', 'Text block', 'MyBlocks');
  RegisterDControl(TDiagramSideLine, '', 'Side line', 'MyBlocks');

  DiagramToolBar1.Rebuild;
end;
```

Prevent a line from being dettached from a block

The following sample code is obsolete since version 2.1, which introduced a new *TDiagramLine.RequiresConnections* property. If you don't want a line to be dettached from a block, just set its RequiresConnections property to true:

```
DiagramLine1.RequiresConnections := true;
```

But for learning purposes, we will keep the old code here:

The following sample code does not allow an user to remove a line from a linkpoint unless it connects it to another linkpoint

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, atDiagram, ExtCtrls, DiagramExtra, FlowChartBlocks;

const
  CM_CONNECTLINKPOINTS = WM_USER + 1000;

type
  TForm1 = class(TForm)
    DiagramToolBar1: TDiagramToolBar;
    atDiagram1: TatDiagram;
    procedure atDiagram1LinkRemoved(Sender: TObject;
      ADControl: TDiagramControl; ALink: TCustomDiagramLine;
      ALinkPoint: TLinkPoint);
  private
    procedure CMConnectLinkPoints(var Msg: TMessage); message CM_CONNECTLINKPOINT
S;
  public
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.atDiagram1LinkRemoved(Sender: TObject;
  ADControl: TDiagramControl; ALink: TCustomDiagramLine;
  ALinkPoint: TLinkPoint);
var
```

```
  c: integer;
  BlockLinkPoint: TLinkPoint;
begin
  if not (csDestroying in ALink.ComponentState) then
    if ADControl is TCustomDiagramBlock then
    begin
      BlockLinkPoint := nil;
      // Find the linkpoint of the block where the line was connected to
      for c := 0 to TCustomDiagramBlock(ADControl).LinkPoints.Count - 1 do
        if ALinkPoint.AnchorLink = TCustomDiagramBlock(ADControl).LinkPoints[c] t
hen
        begin
          BlockLinkPoint := TCustomDiagramBlock(ADControl).LinkPoints[c];
          break;
        end;
      // Reconnect line to the block link point
      PostMessage(Handle, CM_CONNECTLINKPOINTS, integer(ALinkPoint), integer(Bloc
kLinkPoint));
    end;
end;

procedure TForm1.CMConnectLinkPoints(Var Msg: TMessage);
begin
  if TLinkPoint(Msg.wParam).AnchorLink = nil then
    TLinkPoint(Msg.wParam).AnchorLink := TLinkPoint(Msg.LParam);
end;

end.
```

Using Diagram Studio in a DLL

Diagram Studio has some restrictions in order to be used in a DLL. It's a limitation of GDI+ graphical library from Microsoft. Diagram Studio initializes and finalizes the GDI+ library automatically, except when it's inside a DLL (IsLibrary = true), since doing that can cause deadlocks or system crashes.

So in order to use Diagram Studio in a DLL, you must initialize and finalize GDI+ library by yourself, from your EXE application. We have public procedures in Diagram units that make that job for you, so you may just export those functions in your DLL and call them from your application.

The DLL containing Diagram Studio must export the InitializeGdiPlus and FinalizeGdiPlus procedures, both declared at `DgrGdipObj` unit.

```
library DiagramDLL;

uses
  SysUtils,
  Classes,
  DgrGdipObj;

{$R *.res}

exports
  InitializeGdiPlus,
  FinalizeGdiPlus;

begin
end.
```

The application using DLL must call InitializeGdiPlus before using Diagram Studio (or at beginning of program) and FinalizeGdiPlus before it terminates.

```
procedure InitializeGdiPlus; external 'DiagramDLL.dll';
procedure FinalizeGdiPlus; external 'DiagramDLL.dll';

procedure TestLoadUnloadDiagramDLL;
var
  HInst: HMODULE;
begin
  HInst := LoadLibrary('DiagramDLL.dll');
  if HInst <> 0 then
  begin
    InitializeGdiPlus;
    ShowMessage('Loaded');
    FinalizeGdiPlus;
    if FreeLibrary(HInst) then
      ShowMessage('Unloaded');
  end
  else
    ShowMessage('Not Loaded!');
end;
```

Auto-layout

You can perform an automatic layout of the diagram. To do that, just call method *ApplyLayout* passing a valid layout algorithm object:

```
atDiagram1.ApplyLayout(Layout);
```

Currently there is only one layout algorithm available, which is *TForceLayout*, declared in unit `AutoLayout.Force`:

```
uses
  AutoLayout.Force;

var
  Layout: TForceLayout;
begin
  Layout := TForceLayout.Create;
  try
    Layout.DisplacementThreshold := 10;
    Layout.DefaultSpringLength := 100;
    Layout.MaxIterations := 500;
    Layout.Damping := 0.5
    Layout.RepulsionConstant := 100000;
    Layout.AttractionConstant := 0.2
    atDiagram1.ApplyLayout(Layout);
  finally
    Layout.Free;
  end;
end;
```

Each layout algorithm has its own parameters. The code above shows the available parameters for force algorithm. You don't need to set any of those, since default values are already defined.

Live Diagram

Overview

Live Diagram is a library derived from Diagram Studio. It has all the features of the base suite and moreover it can execute the diagrams.
The main component is the Live Diagram (TLiveDiagram), a graphic control able of managing the thread-safe execution of its Live Blocks (TCustomLiveBlock descendants).

To have an executable Diagram, basically we need of design rules and execution strategy. Design rules state how the blocks can be connected each other to form a "well formatted" execution chart, execution strategy states how the blocks must be executed i.e. their flow.
These rules are not set in the diagram, TCustomLiveBlock descendants implement them, so they can be easily extended.

In a Live Diagram can be inserted Live Blocks and traditional components at the same time, but only live blocks will be "managed" (checked, linked, executed).

Purpose

- Computer science fundamentals training and teaching.
- Algorithms check.
- Rapid prototype.
- Extreme user-customization of applications.

Features overview

- Inheritance of all features of TatDiagram without restrictions.
- Design time support for executing native Delphi code.
- Separate threads for execution and visualization, both synchronized.
- Balancing of execution thread CPU consumption.
- Thread safe model, multiple diagram can run at the same time.
- Step by step execution mode.
- Possibility of executing diagrams without a line of code.
- Specialized blocks : Error handler, Line jointer, Connectors.
- Multiple executions paths.

Lacks overview

- Not suitable for industrial automation or critical applications.

- Non time-deterministic execution.
- Simulated multithread (due to Synchronize method).

TLiveDiagram component

It's the Live Diagram main component, it inherits from TatDiagram.

Key Properties

State: TDiagramState = (dsEdit, dsRunning, dsPaused)
Reflects the status of the diagram. It can be used to enable/disable actions and buttons. When not in edit mode, the editing is disabled. Do not modify State directly, use *DoAction* method instead.

RunError: integer
Contains the value of the last Error flag (see *TCustomLiveBlock.Execute*).

ExitCode: integer
After the execution termination contains the exit code (see *TCustomLiveBlock.Execute*).

Liveds[index : integer]: TCustomLiveBlock
Contains the list of all LivedBlocks. It's only filled after a call to Link method. Do not use it in edit mode.

IdlePercent: DWORD
Allows the CPU balancing. Every block execution is time-marked, so if a block is executed in 20ms (for instance) and IdlePercent=50 the execution thread will sleep for 20ms.
It's possible to modify it during the diagram execution.

SleepVisual: DWORD
It refers to the visual thread and is a fixed value of sleep time.

RunColor: TColor
It's the background diagram color during the execution.

Key Methods

function Link: boolean
Checks the diagram consistence and links the blocks, returns true if everything is ok. After a successful link, Liveds property contains the lived block list. Everything that doesn't inherits from TCustomLiveBlock or from a wire is ignored.
Don't call Link directly, override it or PrepareRun.

function PrepareRun: boolean
It's invoked every time the run is request if the diagram was modified. Just calls Link, override this method to perform special init operations.

procedure DoAction(Action: TDiagramAction)
TDiagramAction = (daRun, daRunPaused, daPause, daStop, daForceStop, daReset, daStep)
Performs a diagram operation:

- *daRun*: Starts the diagram execution;

- *daRunPaused*: Starts the diagram execution in pause mode, a *daStep* command is necessary to continue;
- *daPause*: Pauses the execution;
- *daStop*: Stops the execution;
- *daForceStop*: Forces the stop without firing *OnTerminate* event;
- *daReset*: Resets the execution - Start became the current block;
- *daStep*: Executes the next block and returns in pause mode.

An operation to be performed must be compatible with the Diagram state, otherwise it's ignored.

	dsEdit	**dsRunning**	**dsPause**
daRun	OK	Ignored	OK
daRunPaused	OK	Ignored	Ignored
daPause	Ignored	OK	Ignored
daStop	Ignored	OK	OK
daForceStop	Ignored	OK	OK
daReset	Ignored	Ignored	OK
daStep	Ignored	Ignored	OK

Key Events

property OnStart: TStartEvent = procedure(Sender: TLiveDiagram; StartMode: TStartMode)
TStartMode = (smCold, smWarm)
This event is fired whenever the *daRun* action is requested, StartMode=smCold if the diagram starts the run from dsEdit, StartMode=smWarm if the diagram starts the run from dsPaused.

> **NOTE**
> *daStep* action doesn't fire this event.

property OnChangeState: TChangeStateEvent = procedure(Sender: TLiveDiagram; NewState: TDiagramState)
This event is fired whenever the diagram state changes.

property OnUnhandledRunError: TRunErrorEvent = procedure(Sender: TLiveDiagram; Block: TCustomLiveBlock; var Error: integer; var ResumeNext: boolean = false)
This event is fired when an error occurred (User Error flag>0, see *TCustomLiveBlock.Execute*) and an Error handler doesn't exists.
Block is the LiveBlock that raised the error, if ResumeNext is set to true the error is zeroised and the execution continues with Block.Next, otherwise the program is terminated with Exitcode=Error.
If this property is not assigned and Error handler doesn't exists the program terminates with Exitcode=xUnhandled_error.

property OnShift: TShiftEvent = procedure(Sender: TLiveDiagram; OldBlock, NewBlock: TCustomLiveBlock)
This event is fired when NewBlock is being to execute, OldBlock is the block just executed.
The method that fires the event is DoShift (virtual).

> **WARNING**
> *OldBlock* or *NewBlock* could be nil.

property OnTerminate: TTerminateEvent = procedure(Sender: TLiveDiagram; ExitCode: integer)
This event is fired when the diagram terminates the execution.

TCustomLiveBlock base class

It is (and must be) the ancestor of all live blocks. It inherits from TCustomDiagramBlock. It's a base class, so isn't registered with RegisterDControl.

Key Properties

ControlState: TControlState = (csEdit, csRunON, csRunOFF, csRunERR)
It's the block state, this property is used to show different color during the block execution.

PassThrough: boolean
It's true when the block is not "executable", a passthrough block is used to perform special connections (see Jointers and Connectors) and it's completely transparent to the execution. When implementing a passthrough block a mechanism to avoid endless loop during the link process must be set (see at TLiveJoin.GetNext).

IsErrorHandler: boolean
A descendent block that implements error handling must set to true this property. Many error handler can be defined in the same set, but only one of these can be inserted in a chart.

IsStartBlock: boolean
Every chart must contain a start block, it must be unique, this property must be true when the descendant is defined as start point.

IsEndBlock: boolean
Every block can terminate the diagram, but only the End Block will terminate it with the code xNormal_termination (see Execute procedure). The End Block must have this property set to true.

Next: TCustomLiveBlock
Points to the next executable block. The execution strategy is implemented here (the value is read through GetNext method).

RunColors: TBlockColors = class(TPersistent)
Stores the three run colors associated to the ControlState (when in edit mode the block colors are stored in Color and SelColor properties).
It would be better to avoid the use of bitmaps in Liveblocks, the gradient is automatically disabled during the execution, the shadow doesn't create any problem.

Key Methods

function Link: boolean
Finds the next block and stores it into FNext (the field image of Next), returns true when FNext is not nil. Every block which can have an unassigned Next must return true to avoid a design error (see TLiveHeader and TLiveErrorHandler).
Link is called by TLiveDiagram.

function Execute(var Error, Decision: integer): TCustomLiveBlock
Executes the block. Do not call Execute directly, it's declared as public only to be visible outside the unit, OnExecute must be used instead. The variable *Error* is an user passthrough flag in accord to:

- **0**: the execution continues with the Next block;
- **-1**: the block requested the "End of Program", if the block has IsEndBlock=true the program exitcode is xNormal_Termination (0), otherwise the exitcode is xProgram_Termination (2);
- **-2**: there was an exception in the OnExecute event, the program will terminate with xCode_Exception (4) exit code.
- **>0**: the Error handler block is the next block to be executed. If it doesn't exists in the chart, TLiveDiagram fires the *OnUnhandledRunError* event.

The variable *Decision* must be set only for special blocks that perform decisions, like TLiveDecisionBlock and TLiveCaseBlock.
The result of the function is the next block that have to be executed, to perform special operation it would be better to override GetNext method.

procedure DoBeforeRun
This method is called whenever the diagram is about to be executed, override it to perform initialization stuff.

function AcceptLink(ALink: TCustomDiagramLine; LinkIndex: integer; Direction: TLinkDirection): boolean
TLinkDirection = (ldInput, ldOutput)
It's the key method for implementing the design rules. LinkIndex is the block linkpoint number, Direction reflects the link (TCustomDiagramLine) direction source or target.
This function is called by TLiveDiagram whenever a link between two link points is about to be made. Do not call this method directly, override it to perform special checks.

function GetNext: TCustomLiveBlock
It's the read function of the property Next, override this function to setup the execution strategy.

> **NOTE**
> Do not perform heavy operation in this function to avoid slowing down, since it's called at runtime.

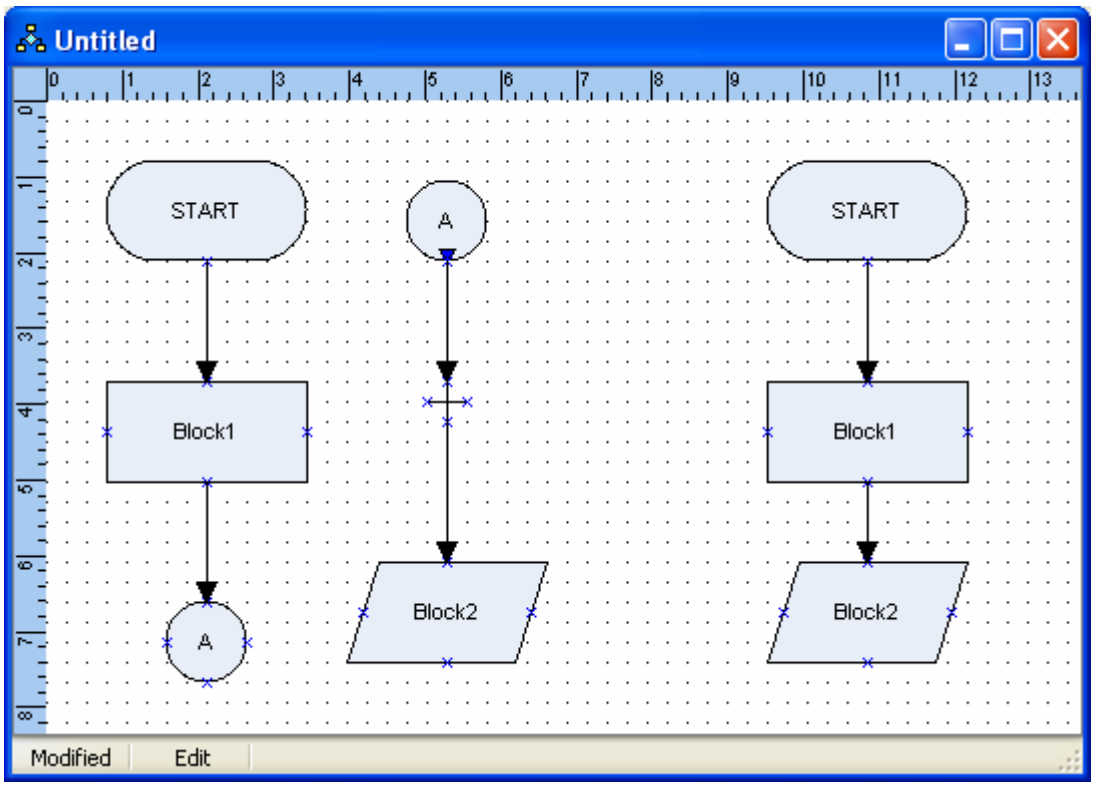
Key Events

property OnExecute: TExecuteEvent = function(Sender: TCustomLiveBlock; var Error, Decision: integer): integer
This event is fired on the block execution. See *Execute* method.

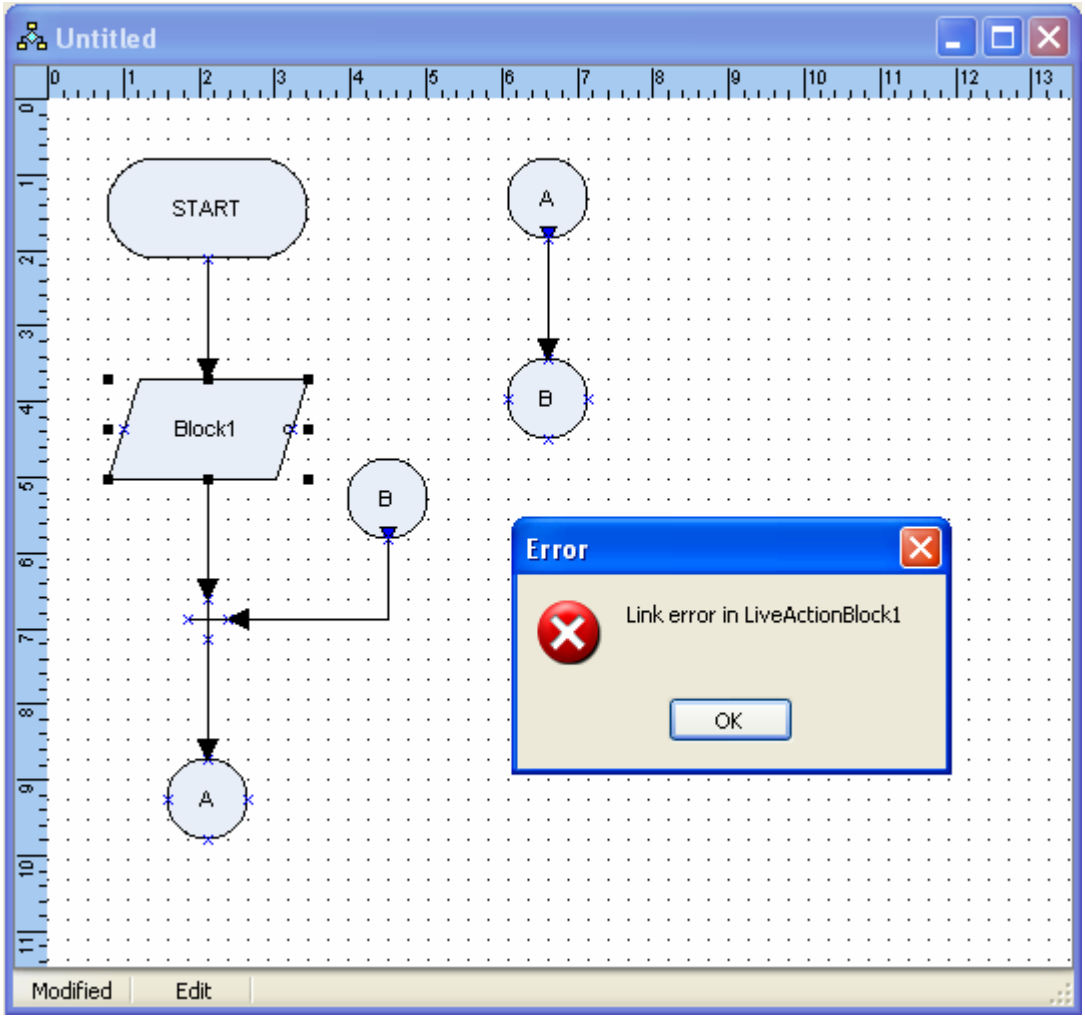
Visual examples of Passthrough, Next and Link

In the diagram below, the flowchart at the left performs the same thing as the flowchart at the right. The "A" blocks are TLiveConnectorSource and TLiveConnectorTarget blocks, and after (following) the second "A" there is a TLiveLineJoin. Those three blocks have the *Passthrough* property set to true, so they perform no action at all.

Also, in both diagrams, the Block1.Next property points to Block2.



In the diagram below, after a call to Link method, the error message appears. Regardless of the strange "path", Block1 is unlinked.



Basic live diagram objects

All live diagram objects are available under the category "Live Diagram". This category is displayed in the diagram toolbar.

For now, there are 3 groups of live diagram objects: basic, flowchart and statechart. This topic covers the basic objects.

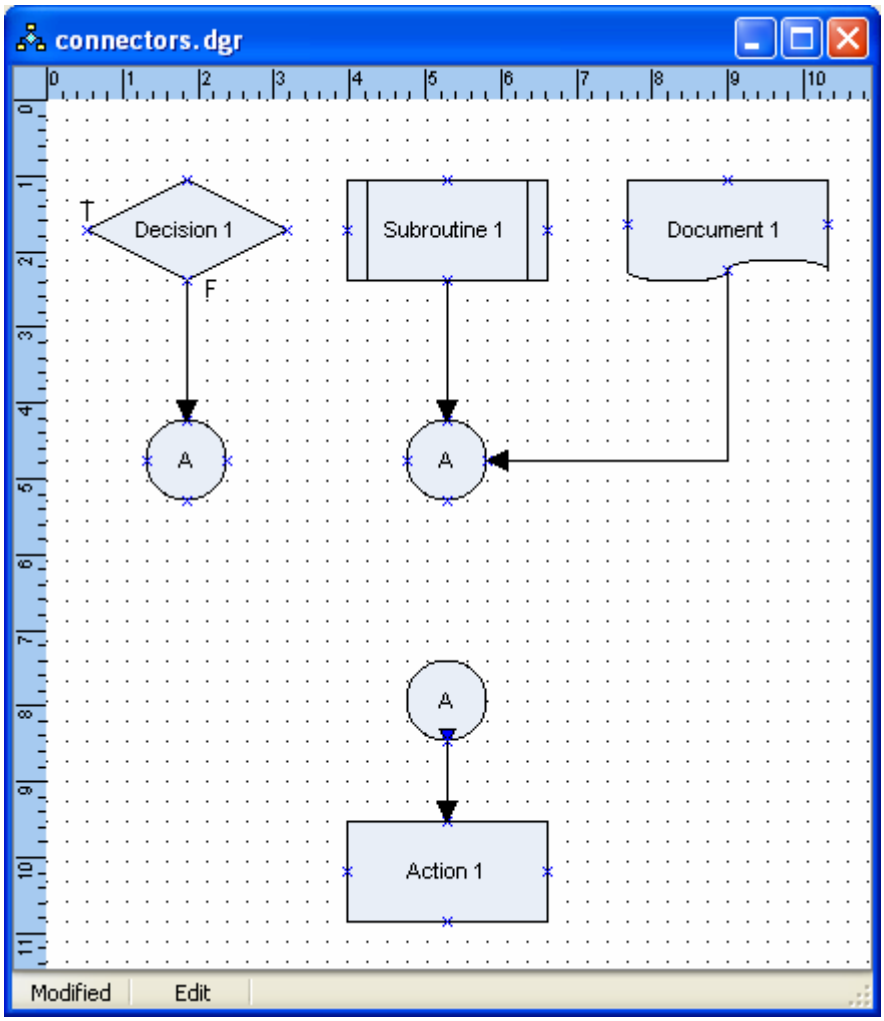
From the left to right in the above picture:

- TLiveConnectorSource (Source connector)
- TLiveConnectorTarget (Target connector)
- TLiveLineJoin (Line jointer)
- TLiveWire (Live wire)
- TLiveSideWire (Live side wire)
- TLiveArc (Live arc)
- TLiveBezier (Live bezier)

TLiveConnectorSource and TLiveConnectionTarget

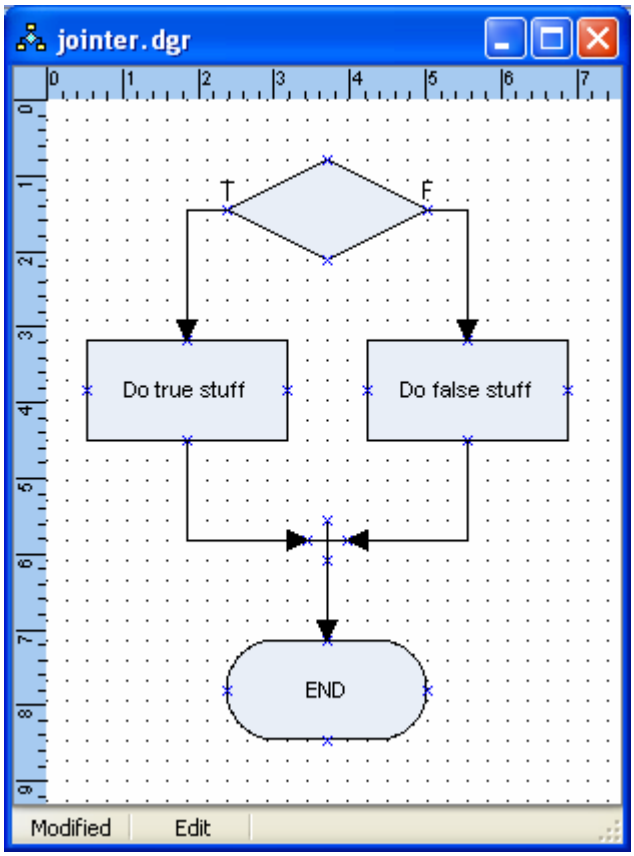
These blocks connect two or more points of a diagram to increase the readability. They are not executable i.e. PassThrough=true.

To be linked they must have the same text (case insensitive). Many source connectors can be linked to an unique target connector. In the picture below, Decision1.Next = Subroutine1.Next = Document1.Next = Action1.



TLiveLineJoin (Line jointer)

Joints multiple input links with an unique output link, it's not executable i.e. PassThrough=true.

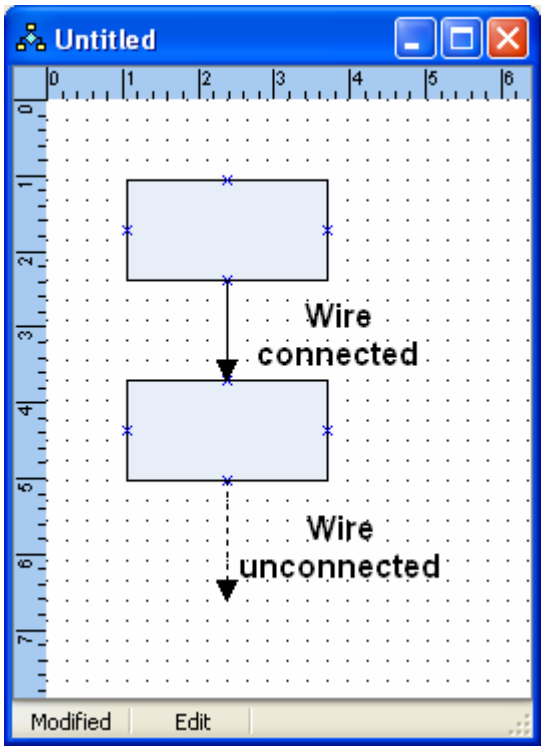


Wire objects

The wire connection objects are TLiveWire (Live wire), TLiveSideWire (Live side wire), TLiveArc (Live arc) and TLiveBezier (Live bezier). Other wire objects can be created, they behave the same way.

The wire connections are "Vector" DiagramLines i.e. the target arrow is always explicit. They are connection aware, when a wire is not connected it's line appears dotted.

When their SourceLinkpoint is connected to a Case block an unique integer value must be written in the text cell.



Key Methods

function Transition(Sender: TCustomLiveBlock): boolean
This function fires the OnTransition event, returns false if the event is not assigned.

Wires are not TCustomLiveBlock descendants, so the blocks that need to manage transition must call this function during their execution (see StateChart blocks implementation), Sender is the Block that requests the transition check.

Key Events

property OnTransition: TTransitionEvent = function(Sender: TCustomDiagramLine; FromBlock: TCustomLiveBlock): boolean
Is the user event fired by the Transition Function. Sender is the Wire and FromBlock is the Block that called the function.

Flowchart objects

This set of objects (avaible in the `LiveFlowChart.pas` unit) allows the creation/execution of Flowcharts. The base rule implemented is that every block can have many inputs and only an output (except for decision and selection).

The integer selector (case) and the error handler are not present in the base flowchart specification, they represent a power extension.

The execution of a flowchart only depends on the blocks, the wires are only links.

From the left to right:

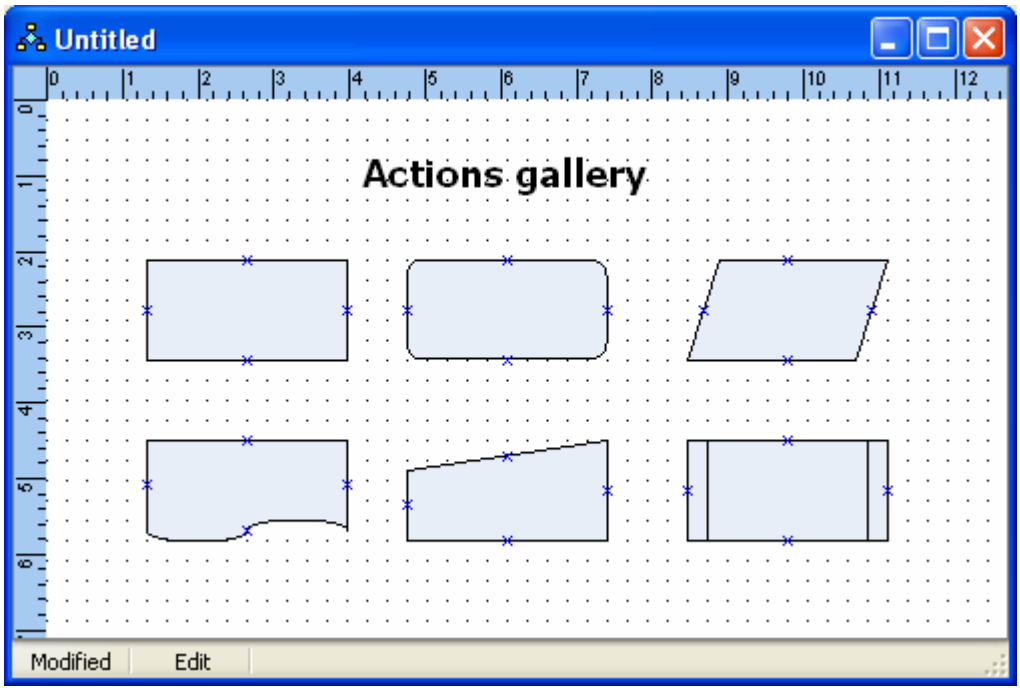
- TLiveActionBlock (Action)
- TLiveDecisionBlock (Decision)
- TLiveCaseBlock (Case)
- TLiveStartBlock (Start program)
- TLiveEndBlock (End program)
- TLiveErrorHandler (Error handler)
- TLiveHeader (Header)

TLiveActionBlock (Action block)

Is the main live executable block, it inherits from TCustomLiveBlock. PassThrough property is false and OnExecute event is published.

Key Properties

ActionShape: TActionShape = *(asBox, asAlternate, asData, asDocument, asInput, asSubRoutine)*
It's only a cosmetic property, it can be changed at runtime without loosing the linkpoints information i.e. the wires connected remain valid.

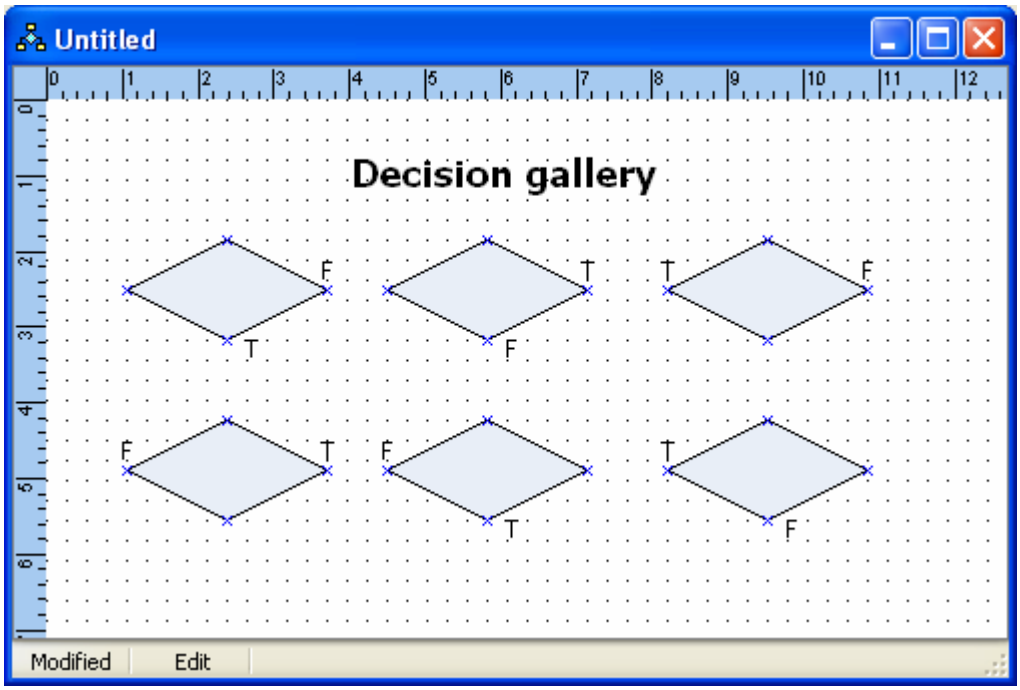


TLiveDecisionBlock (Decision)

Performs the decision of flowchart, it inherits from TCustomLiveBlock. PassThrough property is false and OnExecute event is published. The Decision variable in the OnExecute method tells the digram if the decision was false (value 0) or true (value different from 0).

Key Properties

DecisionShape: TDecisionShape = *(dsBottomRight, dsRightBottom, dsLeftRight, dsRightLeft, dsBottomLeft, dsLeftBottom)*
It's only a cosmetic property but is very useful to have nice diagrams, it can be changed at runtime without loosing the linkpoints information i.e. the wires connected remain valid.

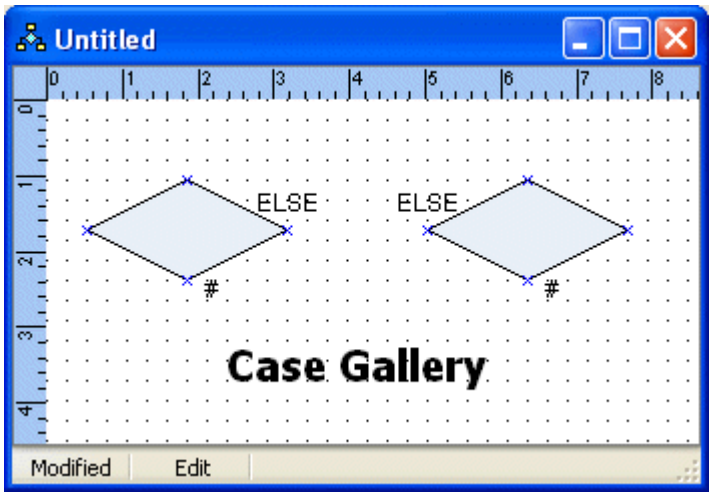


TLiveCaseBlock (Case)

Performs an integer selection (called case in more programming languages). It inherits from TCustomLiveBlock. PassThrough is false and OnExecute event is published. The next block executed depends of the integer value of parameter Decision of OnExecute event. If no case was matched, the block connected to ELSE output will be executed. Many wires can be connected to case output, and the chosen wire will be the one which text (caption) is the same as the number returned in the Decision parameter of OnExecute event.

Key Properties

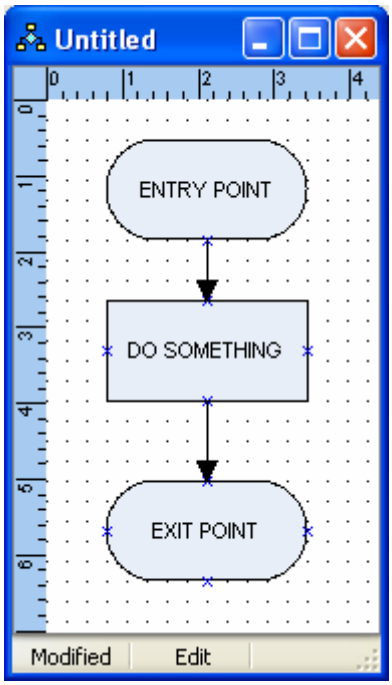
CaseShape: TCaseShape = *(csBottomRight, csBottomLeft)*
It's the cosmetic property that allows different orientation of the block, it can be changed at runtime without loosing the linkpoints information i.e. the wires connected remain valid.



TLiveStartBlock (Start program)

It's the start point of the program, it's executable and can be executed once, so usually should contain initialization code.

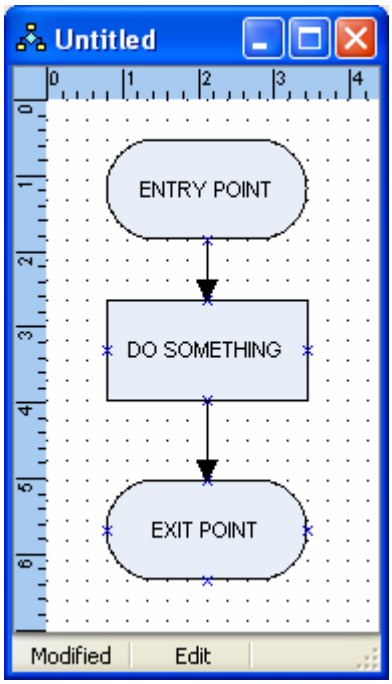
Must be always present and must be unique. The text "START" can be freely edited.



TLiveEndBlock (End program)

It's the endpoint of the program, it's executable and can be executed once, so usually should contain deinitialization code.

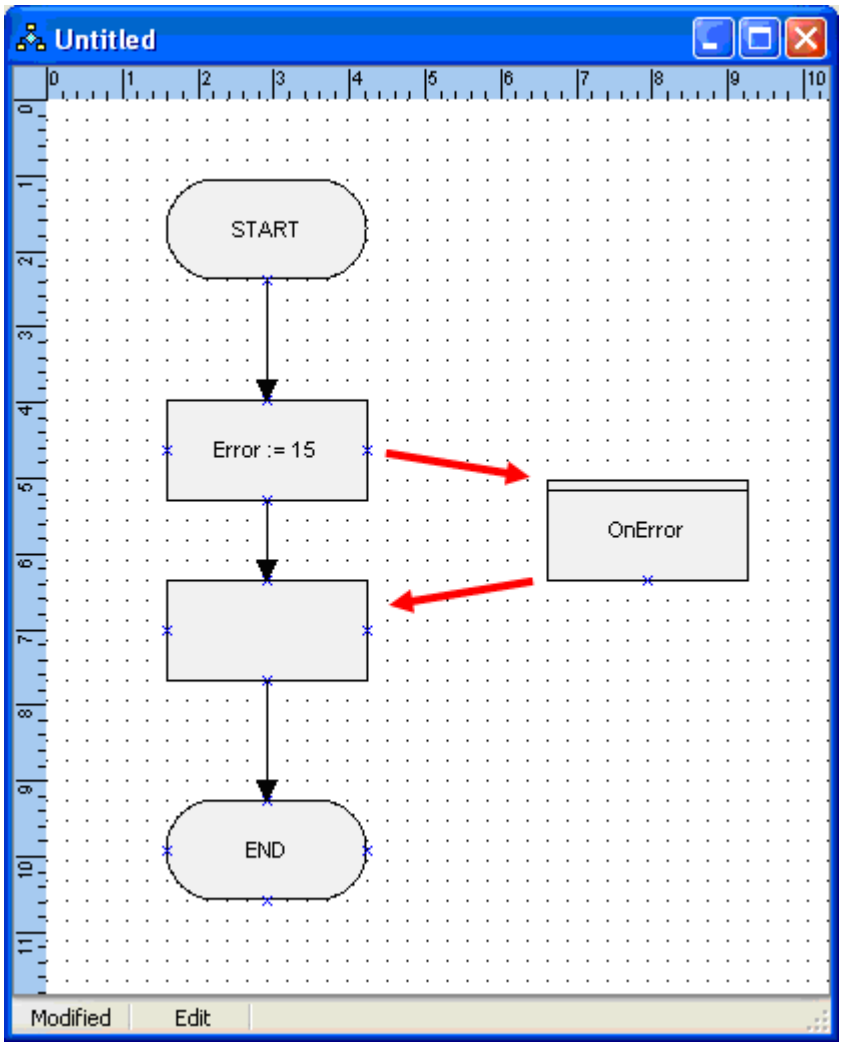
Can be absent. The text "END" can be freely edited.



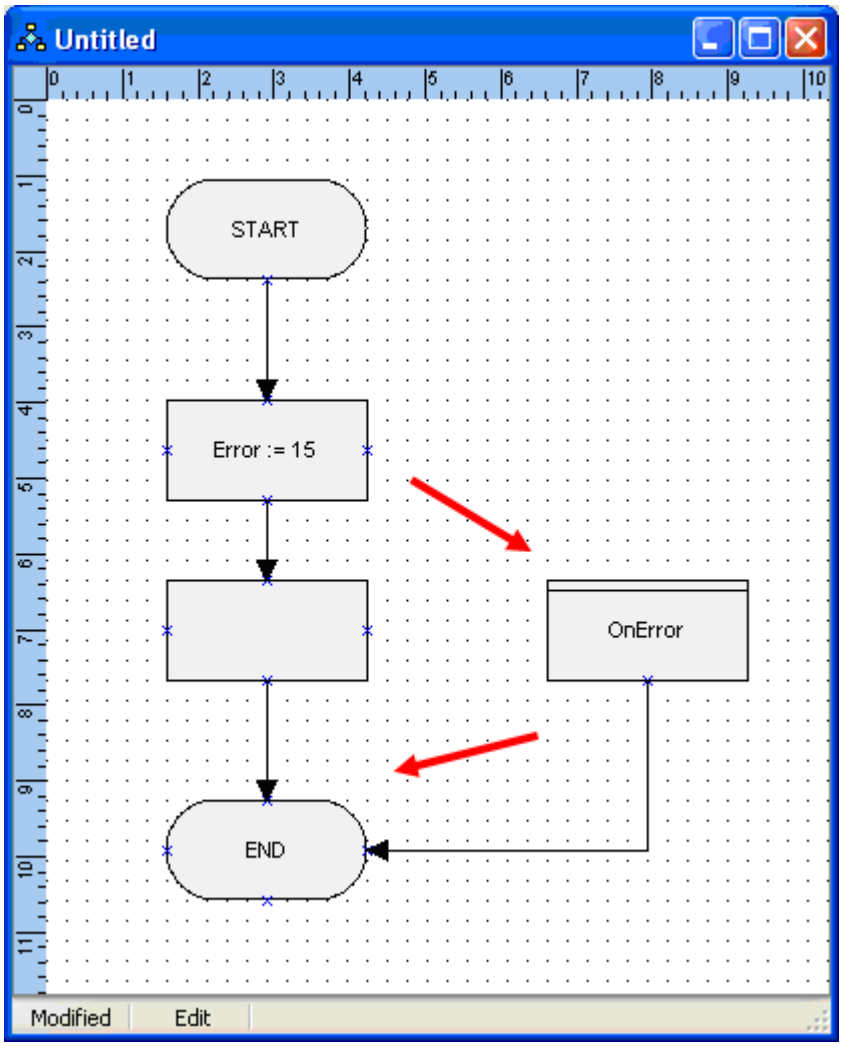
TLiveErrorHandler (Error handler)

It's an asynchronous block, it's executed when a block exits with Error > 0.
It has no input, if it's Next is assigned the execution continues there, otherwise is performed an implicit "resume next", i.e. is executed the next block in the chain.
Error handler can be absent, but if present must be unique. To avoid loop it must reset the error variable. The text "OnError" can be free edited.

For historical reasons, the Error handler doesn't exists in the traditional flowchart component set, it creates a "not structured" point in the diagram.
Its use should be limited to error handling, use the decision for normal branch.



Fixed output



Resume next

TLiveHeader (Header)

It's just a container and is not executable.

State chart objects

This set of objects allows the creation/execution of Statecharts. The base rule implemented is that every block can have many inputs and many outputs.

The execution of a statechart depends on the blocks and on the wires which are named transitions here, important examples of state diagrams are Petri's Net and Grafcet.

In this implementation the *Transition* method of wires is used, an extension could be possible to write a specialized block (TLiveStateTransition for example) to accomplish the shift check.

Since they are descendant of TCustomLiveBlock, the States blocks also have the Execute method.

The base class for all state blocks is the TCustomLiveState class. All state blocks inherit from it and make use of its key properties.

The avaiable state objects, from the left to right in the above picture, are:

- *TLiveStateBlock (State)*: It's the "usable" state block, it inherits from TCustomLiveState and only publishes some properties.
- *TLiveStateStart (Start state)*: It's the initial state in the chart i.e. it's the program start point. It accepts only an unique output link freely anchorable at any linkpoint, and there is no transition check i.e. the output link transition is always true.
  Its shape reflects the UML standard (a small circle).
- *TLiveStateEnd (End state)*: It's the program end. It accepts only input links, after it's execution the program terminates.
  Its shape reflects the UML standard (a small circle with a black inscribed solid circle).
- *TLiveStateError (Error state handler)*: It's the StateChart error handler. Since every state can have multiple outputs, here is not possible for the diagram to implement the "resume next" feature, so an output link must always be specified.
  This block accepts only an unique output link freely anchorable at any linkpoint.

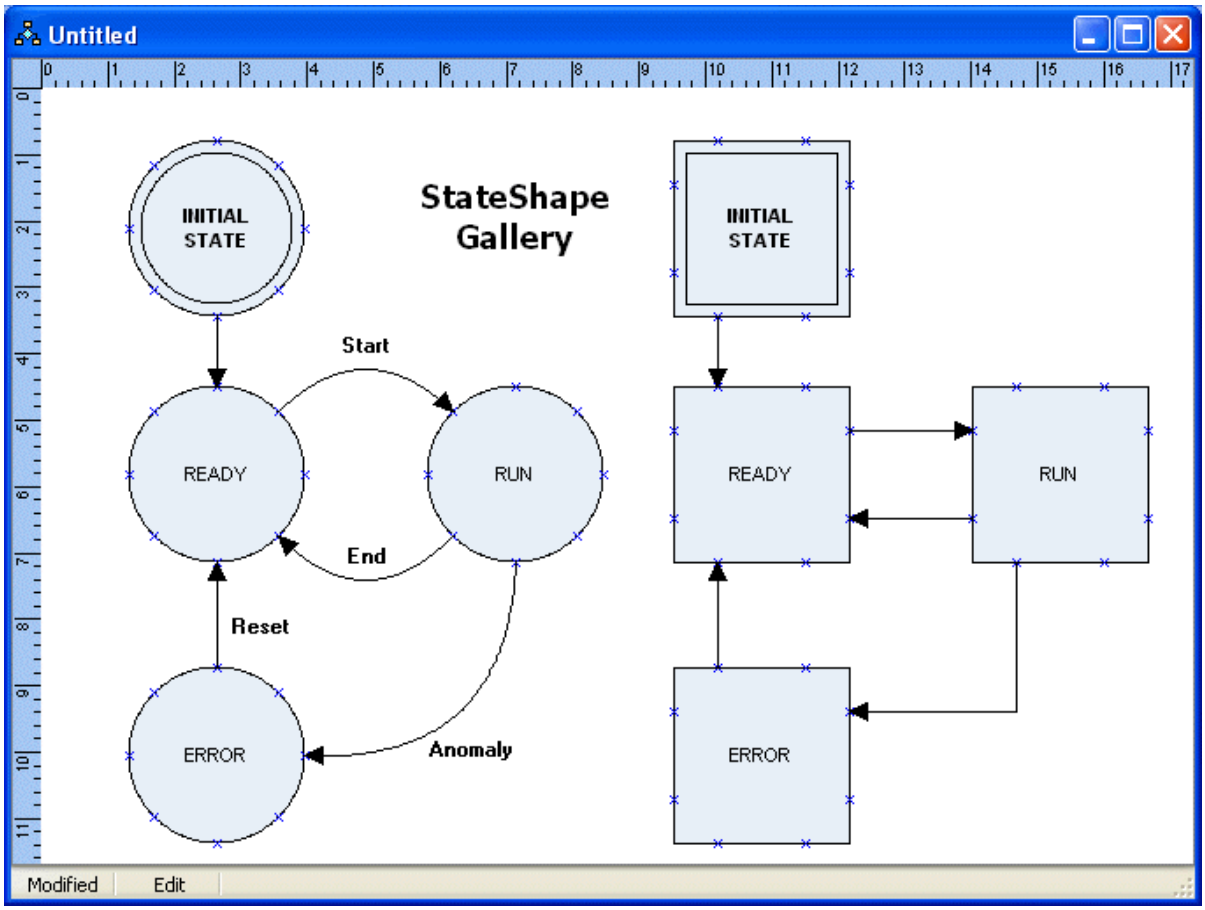
TCustomLiveState class

It's the ancestor of all "state" components, it implements the design rules and the execution strategy. It inherits from TCustomLiveBlock.

Key Properties

OneShot: boolean
Normally a state is active until a transition is satisfied, if OneShot = true the event *OnExecute* is fired once, otherwise it's execute before every transitions scan.

StateShape: TStateShape = (ssCircle, ssSquare)
It's the cosmetic property that allows different shapes of the block, it can be changed at runtime without loosing the linkpoints information i.e. the wires connected remain valid.



Extending Live Diagram

Due to its modular design, the extension of Live Diagram is quite simple. You should proceed through the following steps (after the reading the topics about the live diagram classes like TLiveDiagram, TCustomLiveBlock, flowchart blocks, statechart blocks and wires).

An extension is a set of blocks that must implement design rules and execution strategies; this set, for design coherence should be contained in a separated file (*LiveXXXXChart* for example), writing it from scratch or inheriting it from an existing one.

1. Set up design rules

As said, they state how the blocks can be connected each other to form a "well formatted" chart.

To do this, override the method *TCustomLiveBlock.AcceptLink* in order to reflect the rule during the design. The function Link must check the connections and must return true if everything (a design time) is ok, by default it pre-calculates the next block in the execution chain.

Moreover there are some basic rules that are always checked directly by the diagram.

- Every chart must contain a Start Block and it must be unique.
- Every chart can contain an End Block, it's suggested but it's not mandatory.
- Every chart can contain an unique Error Handler Block.

2. Setup execution strategy

After the execution, a block must provide the "next" block that must be executed, by default this was pre-calculated by the function Link. In certain cases (when the block has more then one output) the block must determine at runtime his successor in according to its internal status. For example, the decision block and the case block in the flowchart set override the DoExecute method to do this.

Note that even though a block can have more than one output at design time, it must provide only one "next" at runtime.

NOTE
LiveChart requests only a well formatted diagram which is not automatically a structured diagram, especially if "case" blocks in flowchart are used (Search Böhm-Jacopini theorem on the Net for further information). Moreover the Diagram doesn't perform any "style check" - it's possible to insert in the same diagram flowchart blocks and statechart blocks.

Live Diagram limitations

- AutomaticNodes property doesn't affects the execution of the diagram but could generate confusion.

About

This documentation is for TMS Diagram Studio.

In this section:

Copyright Notice

What's New

Getting Support

Breaking Changes

Licensing and Copyright Notice

Diagram Studio components trial version are free for use in non-commercial applications, that is any software that is not being sold in one or another way or that does not generate income in any way by the use of the application.

For use in commercial applications, you must purchase a single license or a site license of Diagram Studio. A site license allows an unlimited number of developers within the company holding the license to use the components for commercial application development and to obtain free updates for a full version cycle and priority email support. A single developer license allows ONE developer within a company to use the components for commercial application development, to obtain free updates and priority email support. A single developer license is NOT transferable to another developer within the company or to a developer from another company. Both licenses allow royalty free use of the components when used in binary compiled applications.

The component cannot be distributed in any other way except through free accessible Internet Web pages or ftp servers. The component can only be distributed on CD-ROM or other media with written autorization of the author.

Online registration for Diagram Studio is available at https://www.tmssoftware.com/site/orders.asp. Source code & license is sent immediately upon receipt of check or registration by email.

What's New

Version 4.23 (Aug-2020)

- **Improved:** High DPI improvements: TDiagramToolbar has correct height and button sizes. Ruler has now correct division lines.

Version 4.22 (Jun-2020)

- **New: Support for RAD Studio 10.4 Sydney.**
- **Improved:** Better display of alternate line dashes in Direct2D rendering.
- **Fixed:** Sporadic Access Violation in TUMLBlock.

Version 4.21 (Feb-2020)

- **Fixed:** Node collapsing not correctly working for grouped blocks.
- **Fixed:** Some blocks not appearing when exporting to bitmap in some situations.

Version 4.20 (Jan-2020)

- **Fixed:** TatDiagram.MoveBlocks method behaving wrongly when moving group blocks and parameter AOnlySelected is false.

Version 4.19 (Nov-2019)

- **New: Experimental (beta) support for High DPI monitors.** For now only the TatDiagram component itself is supposed to the HDPI compatible. The full editor (provided by TDiagramEditor component) is not yet HDPI compatible.
- **Fixed:** Access Violation when clearing a diagram containing grouped blocks. The "clearing" operation happens actually in several situations, like undo, adding block to library, closing a diagram, etc.. The error didn't happen in every diagram with grouped blocks, usually only when groups had many diagram lines connected to each others.

Version 4.18 (Jul-2019)

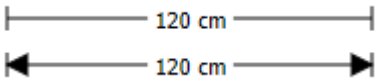
- **Improved:** TDiagramEditor.DialogMode allows displaying the diagram editor window as non-modal.
- **Fixed:** Memory leak when opening the diagram editor with existing custom blocks in user library.

Version 4.17 (Dec-2018)

- **New: Support for Delphi/C++Builder 10.3 Rio.**
- **New: TPictureMore.pmStretchProportional stretches the picture to the whole block size but still keeps original image ratio.**
- **New: PasteOffsetX and PasteOffsetY properties controls where a new pasted diagram object will be positioned compared to the original copied one.**
- **Improved:** PageLines now follows FixedPageWidth, FixedPageHeight, FixedPageCols and FixedPageRows, if they are defined.
- **Fixed:** UML Blocks showing wrong margins around title section depending on font size.
- **Fixed:** Sporadic Access Violation when destroying diagram with layers.

Version 4.16 (Jul-2018)

- **New: TArrowShape.ShowGuide property** allows showing "guides" (extension/dimension lines) at the end of lines.



- **New: CenterTextCellOffset property** allows controlling the center text cell position relative to the line center.
- **New: Properties FixedPageWidth, FixedPageHeight, FixedPageCols and FixedPageRows** allows setting a fixed size for the diagram, in pixels.
- **New: TExportSize esPage** allows exporting the diagram to a bitmap in the full size - the whole diagram will be exported with its exact size. Mostly used with FixedPage* properties.
- **Fixed:** Clipboard issues in Windows 10.

Version 4.15 (Mar-2018)

- **Fixed:** Round rectangles (shaped based on bsSquareShape) not showing correctly in diagram navigator.
- **Fixed:** Showing a modal form during OnDblClick would cause diagram to enter select mode even with mouse button unpressed.
- **Fixed:** LiveDiagram was not restoring visual settings of EndBlock after an execution.

Version 4.14 (Jul-2017)

- **Improved:** Changed order of handle detection - handles at the end of Handles collection will have higher priority on handle selection.

- **Fixed:** Wrong preview/print for landscape more (regression).
- **Fixed:** TDiagramButtons control displaying all categories regardless of the value of DiagramCategories property.

Previous Versions

Version 4.13 (Mar-2017)

- New: RAD Studio 10.2 Tokyo Support.

Version 4.12 (Mar-2017)

- Fixed: RoundRect blocks displaying differently when using GDI or GDIPlus/Direct2D.
- Fixed: Exporting to bitmap not including diagram background image when using Direct2D.

Version 4.11 (Sep-2016)

- Fixed: Anchored links not being moved together with blocks.

Version 4.10 (Aug-2016)

- New: TCustomDiagramBlock.CornerRadius property for fine-tuning of rounded corner when using rounded-square shape.
- Improved: Improved performance when clearing or deleting many diagram objects at once.
- Fixed: Preview bitmap for landscape orientation.
- Fixed: Moving groups that have lines anchored would cause misplacement of group members.
- Fixed: Diagram toolbar with wrong height at design-time.

Version 4.9.1 (Aug-2016)

- Fixed: C++ Builder 2007 headers not being generated correctly. Breaking change.

Version 4.9 (Apr-2016)

- New: Support for Delphi/C++Builder 10.1 Berlin.

Version 4.8.5 (Nov-2015)

- Fixed: Getting default printer failed with Unicode characters in Delphi 2009.
- Fixed: OnModified event not being fired when blocks were moved using arrow keys.

Version 4.8.4 (Sep-2015)

- New: RAD Studio 10 Seattle support.

Version 4.8.3 (Aug-2015)

- New: TDiagramControl.SelectableTextCells allows taking text cells into consideration when selecting blocks and scrolling the diagram.
- New: TatDiagram.OnOpenEditor event gives opportunity to execute code right before text editor is displayed (when editing text in diagram object).
- Fixed: Diagram navigator cropping right side of selection rectangle in some situations.
- Fixed: Using REMOVEGDIPLUS directive was not compiling.
- Fixed: Texture brush not working when using Direct2D graphic lib.

Version 4.8.2 (Apr-2015)

- New: Delphi/C++Builder XE8 support.
- Improved: Diagram buttons (objects panel) comes with first item selected, instead of last one.
- Fixed: Live Diagram blocks were not appearing in toolbar at design-time.

Version 4.8.1 (Mar-2015)

- Fixed: Error with C++ hpp files for FlowchartBlocks (and other blocks) generated incorrectly.

Version 4.8 (Fev-2015)

- New: Package changes now allow use of runtime packages with 64-bit applications. It's a breaking change.

Version 4.7 (Sep-2014)

- New: RAD Studio XE7 support.
- New: Clicking navigator in a position now automatically scrolls the diagram to the respective position.
- Fixed: Conflict between TeeChart and TMS Diagram when using C++ Builder.

Version 4.6.1 (May-2014)

- New: RAD Studio XE6 support.
- Fixed: Links were missing when using library controls and ungrouping them.
- Fixed: Navigator not being painted when diagram had no scrooll bars in Direct2D.

- Fixed: Wrong text alignment when using TVertAlign.vaBottom or TAlignment.taRightJustify.

Version 4.6 (Mar-2014)

- New: TDiagramNavigator control provides an overview and easy navigation of the whole diagram.
- New: TDiagramEditor.ShowNavigator and NavigatorVisible properties allows full control of navigator visibility in the built-in diagram editor dialog.
- New: TForceLayout.ApplyNodesOnError allows the layout to be applied to diagram even if errors occur while calculating the new layout.

Version 4.5 (Oct-2013)

- New: RAD Studio XE5 support.
- New: Keep lines vertical/horizontal-only while resizing by keeping Ctrl key pressed.
- New: OnlyFocused property in diagram actions cut, copy, paste and undo, allows such actions to not perform is diagram control is not focused.
- Fixed: List index out of bounds when no blocks are registered in diagram editor.
- Fixed: "Control has no parent window" error when loading blocks from library manager.

Version 4.4 (May-2013)

- New: Diagram auto-layout using force algorithm.
- New: RAD Studio XE4 support (version 4.3.1).

Version 4.3 (Sep-2012)

- New: RAD Studio XE3 support.
- New: TTextCell.ReadOnly property.
- Improved: Better handling of texts in automatic icon creation in library dialog.
- Improved: Property StringData now being saved in TGroupBlock and TLineJoin blocks.
- Fixed: Access Violation when a showmessage is displayed in OnInsertDControl event.
- Fixed: Block position getting rounded values in very specific situations.
- Fixed: Issue with gradient in some UML blocks.
- Fixed: Issue with resizing using left/top/right/bottom handles and keeping aspect ratio.
- Fixed: Collapsable nodes not being hidden when an object is behind another object.
- Fixed: Error when getting default printer when using Diagram Studio under Citrix.

Version 4.2 (Apr-2012)

- New: Support for 64-bit applications.
- New: Automatic scroll when selecting controls and mouse cursor is positioned outside diagram boundaries.
- New: TDiagramRuler.Offset property allows you to specify an initial number for the ruler different from zero.
- New: TDiagramRuler.Inverse property allows you to set the initial point of the ruler at the right (instead of left) or bottom (instead of top).
- New: Support for collation and duplex in diagram printing (TDiagramPrintSettings).
- Improved: New TGroupBlock BeginUpdateMembers and EndUpdateMembers allow faster member including in group blocks.
- Fixed: Isse when print previewing diagram in landscape orientation.
- Fixed: issue with print preview in Delphi/C++Builder XE2.
- Fixed: Issue with select cursor in some situations where line pen width changes.
- Fixed: Rare issue with block position getting rounded values when not needed.
- Fixed: Sporadic Access Violation when a showmessage is displayed in OnInsertDControl event.

Version 4.1 (Sep-2011)

- New: RAD Studio XE2 support.
- Fixed: Minor issue with hints when Application.HintShowPause is different than zero.
- Fixed: Issue with C++Builder XE installation.
- Fixed: Minor issue with TDiagramButtons.BevelFlat property.
- Fixed: Minor issue with polylines and export to WMF.
- Fixed: Issue with text cell autoframe not working properly with different vertical alignments.
- Fixed: TWinControlBlock not working properly with Direct2D.
- Fixed: Printing in Direct2D (forced to use GDI+).
- Fixed: Icons in palette toolbar for 3rd party blocks not showing when using runtime packages.
- Fixed: Issue with RequiredConnections and OnAcceptLink event.
- Fixed: Block palette not updating when a library is deleted.

Version 4.0.4 (Dec-2010)

- Fixed: Error when opening diagram design-time editor in Delphi 7 and previous versions.

Version 4.0.3 (Dec-2010)

- New: Context menu for layer assignment in diagram objects.
- Improved: Properties made public to allow creating descendants for TDgrLibraryFiler.
- Fixed: Painting issue moving blocks with DragStyle = dsOutline.
- Fixed: Access Violation selecting a PNG file in Open Picture dialog.
- Fixed: TextCell not resizing proportionally to the length of the text.
- Fixed: Issue calculating handles of arcs into a group block after moving, rotating or resizing.
- Fixed: Text cell bounds of some diagram blocks to avoid text going outside the shape.
- Fixed: Issue with block clipping (Direct2D).
- Fixed: Issues exporting/printing a diagram using Direct2D.
- Fixed: Issue drawing UMLStateBlock and FlowTerminalBlock with height larger than width.
- Fixed: Text bounds calculated incorrectly when drawing CellFrame with AutoFrame=true (Direct2D).
- Fixed: Error drawing linkpoints using GDI library.
- Fixed: Issue with GDI+ using Diagram inside a DLL.

Version 4.0.2 (Oct-2010)

- New: Properties TGdipPen.GPPen and TGdipBrush.GPBrush.
- Fixed: Icons not painted properly on library manager when the library contains more than 15 items.
- Fixed: Flickering when moving blocks on LiveDiagram.
- Fixed: Coordinates calculated incorrectly when using AutoFrame in TextCell.
- Fixed: Issue with painting of rounded blocks (Direct2D).
- Fixed: Incorrect drawing of Switch block (Electric blocks).
- Fixed: Access Violation in Layer Manager after changing diagram layers.

Version 4.0.1 (Sep-2010)

- New: RAD Studio XE support
- New: TDgrLibraryItem.CreateControl to allow adding a library control to diagram programmatically.
- New: TDgrLibrary.FindItem method.
- Fixed: Arrow blocks could not be selected when using GDI library.
- Fixed: Access Violation drawing collapsable link points.

- Fixed: Memory leaks.

Version 4.0 (Jul-2010)

- New: Direct2D support (Delphi 2010 and later).
- New: Improved positioning precision (from integer to floating point).
- New: New UML and DFD blocks.
- New: Generic classes and methods for drawing regardless of graphic library used (GDI+, Direct2D).
- New: TatDiagram.GraphicLib and TDiagramControl.GraphicLib properties.
- New: Fully documented source code.
- New: Load/clear background features in diagram editor.
- New: Menu option to show/hide Diagram Objects tab in diagram editor.
- New: TDiagramEditor.Diagram property.
- New: Parameter in TatDiagram.EndUpdate method for optimization.
- New: TatDiagram methods BeginUpdateSel and EndUpdateSel made public.
- New: TatDiagram.CustomGroupBlockClass property.
- New: TatDiagram.CustomDiagramMemoClass property.
- New: TatDiagram.PageColCount and TatDiagram.PageRowCount properties.
- Improved: Save / "Save as" features in diagram editor.
- Fixed: Incorrect width of line buttons in TDiagramEditor interface.
- Fixed: TDiagramToolBar.Mode property was not correctly updated on Object Inspector.
- Fixed: Issues with copy/paste of runtime created blocks.
- Fixed: Access Violation caused by copying/pasting of layered blocks.
- Fixed: Linked lines were not being moved together when moving blocks with ctrl+arrow.
- Fixed: Problem redrawing blocks with a rotation angle.

Version 3.4 (Jan-2010)

- New: TDgrLibraryManager component implements custom Library system that allows end-user to build blocks visually in diagram and add them to the toolbar as new custom blocks.
- New: Selecting one or more group members is allowed now. You can Ctrl+Click a group to select a member and individually resize, move or set properties for a block belonging to a group.
- New: Photoshop-like layer system. Named layers can be added/removed in a visual layer manager. Each layer can visually be hidden or protected from being edited.

- New: TDgrLibraryFolderFiler component allows custom blocks to be currently saved to files, but an abstract layer is provided to allow saving custom blocks to database or other media.
- New: Library manager window allows end-user to visually manage the libraries, create new libraries, define glyphs for custom blocks, among other tasks.
- New: TDgrLayerSelector combo box allows setting the layer for the selected objects.
- New: Several classes, methods and properties for using new layer system programatically.
- New: Support for diagram usage in frames, page controls, and other types of controls.
- New: TPolygonBlock.PointColor allows to define a color for the polygon point handles that is different from the regular block handles.
- New: Horizontal/vertical only moving when Shift key is pressed.
- New: TatDiagram.ZoomMoveToFit method to make all existing objects to be visible in the diagram client area.
- New: TRegDControl.Glyph and UseGlyph properties makes it easier to set an icon for a block in the diagram toolbar.
- New: TatDiagram.DesignMode provides easy way to set the whole diagram into read-only mode.
- New: Methods MMToMeasUnit and MeasUnitToMM make it easy unit conversion from mm to currently selected measurement unit.
- New: Additional object restrictions crNoLink (to prevent automatic linking) and crNoRotCenterMove (to prevent moving rotation center).
- New: Added several events for selectors like OnDropDown, OnCloseUp, OnKeyPress, among others.
- New: TDiagramUndoItem.Obj property allows custom data associated with an undo item.
- New: TatDiagram.RefreshToolbars method for requesting object toolbars to refresh its content.
- New: TTextCell.Transparency and ParentTransparency properties.
- New: TDiagramEditor.LibManager property.
- New: TatDiagram.OnBeforeCreatecontrol event.
- New: TatDiagram.MoveControlIndex method allows settings the index position of a diagram object in the internal diagram object list.
- Improved: Compatibility with ParentBackground property allows use of transparent diagram controls (following same restrictions of ParentBackground property).
- Improved: Diagram now gives an automatic generated name for blocks inserted programatically without a name. This minimizes losing connection links in some situations.
- Improved: Old layer system now supports 64 layers instead of 32 layers.

- Improved: Demo project with library manager component allows testing the new library system.
- Fixed: Several issues with color selectors.
- Fixed: Small issues with inplace text cell editor.
- Fixed: Issue with font name and size selectors disappearing in tms ribbon toolbars.
- Fixed: Issue with text cells losing text value in rare situations.
- Fixed: Issue with blocks being inserted with zero width or height.
- Fixed: Issue with auto created link points not being correctly positioned in some situations.
- Fixed: Pasting from clipboard operation was not setting diagram modified flag.
- Fixed: Cross indicators were not being displayed for side lines when the line object was TDiagramLine.
- Fixed: Small issue with diamond-type arrow not being painted correctly with zoomed diagram.
- Fixed: Diagram hanging when default installed printer had a default paper with small size.
- Fixed: Issue with custom glyphs in selectors not being displayed when the drop down button was pressed.

Version 3.3 (Jun-2009)

- New: Diagram print system improves overall diagram printing mechanism / dialog.
- New: TatDiagram.PageSettings property allows setting diagram page properties for printing, like paper size, margins and paper orientation.
- New: TatDiagram.PageSetupDlg method shows a dialog for page setup by end-user.
- New: TatDiagram.PrintSettings allows defining printing options like copies and pages to be printed.
- New: TatDiagram.WheelZoom, WheelZoomIncrement, WheelZoomMin, WheelZoomMax properties, allow diagram zooming in/out using Ctrl+Mouse Scroll.
- New: TatDiagram.DPrinter property returns a TDiagramPrinter object which current printer settings.
- New: TatDiagram.MeasUnit allows to set the global measurement unit (mm, cm, in) used for diagram page settings.
- New: TatDiagram.UnitSymbol property retrieves the symbol for the current unit used in diagram page.
- New: TDiagramEditor.OnSaveDiagram event.
- New: New menu item "File | Page Setup" in diagram editor.
- New: TDiagramButtons.KeepInsertingMode property.
- Improved: Send to back/bring to front operations when using grouped blocks.

- Improved: Print method now can optionally show a print dialog window before printing.
- Improved: TDgrZoomSelector (zoom combo box) update itself automatically, being in sync with the current diagram zoom value.
- Improved: Some properties and functions had types changed from single to double to improve accuracy.
- Fixed: Issue with saving/loading png and jpg images.
- Fixed: Several fixes in the diagram selectors.

Version 3.2 (Dec-2008)

- New: Grouping concept introduced. Diagram objects can now be grouped and behave like a single object. Multi-level grouping is supported.
- New: Movable rotation center allows defining a new center point for rotating objects.
- New: TDiagramLine.LineStyle property allows a single line object to behave like arc, side line, bezier and straight line. No need for using different line classes anymore.
- New: Group/ungroup popup menu and toolbar buttons available in diagram editor.
- New: Text toolbar in diagram editor dialog allows setting text alignment, vertical alignment and font style (bold, italic and underline).
- New: TatDiagram methods GroupSelectedBlocks, UngroupSelectedBlocks and AddGroup for handle groups programatically.
- New: TatDiagram.SelectionMode property allows to choose Visio-style block selection (objects selected are displayed as a whole group).
- New: TDiagramBackground.Picture property allows more image types for diagram background image (jpg, wmf, etc).
- New: TatDiagram.TextRenderingMode allows setting high-quality antialiasing text mode.
- New: TatDiagram.CanMoveOutOfBounds property allows avoiding blocks to moved to negative positions, preventing the whole diagram to be shifted.
- New: tmSpecific item in TTextCellsMode allows fixed text cells at source/target position of lines.
- New: Method in TatDiagram to export the diagram as bitmap/metafile to a stream instead of a file.
- New: DropDownListColor property for all selector combo boxes (font selector, font size selector, etc).
- New: TatDiagram.FixedSideLines property allows a different side line behavior, where intermediate handles are not automatically repositioned.
- New: Arrow shape asAngleDiamond is a diamond which rotates according to line angle.
- New: TDiagramControl properties IsGroup, IsMember, GroupBlock and MasterGroup provides information about block grouping.

- New: Protected method TDiagramControl.TextCellsChanged.
- New: TStretchHandle.AlwaysMovable property allows a handle to be moved even when crResize restriction is present.
- New: TLinkPoint properties CanAnchor, CanAnchorInMove and AcceptAnchoreds allow better control of link point behavior.
- Changed: TDiagramControl.GetTextCellRect method signature.
- Improved: Side line automatic positioning.
- Fixed: Database block text cell could not be edited.
- Fixed: Issue with arc line when the arc handles were too close to each other.
- Fixed: Error with block gradient in some specific situations.
- Fixed: Small visual issue in diagram editor: "target arrow" popup menu item caption was displayed incorrectly.
- Fixed: Several compilation warnings removed.

Version 3.1 (Oct-2008)

- New: Delphi/C++Builder 2009 support.
- New: Rounded square block shape (TBlockShape type bsSquareRound).
- New: Half-line arrow (TArrowShape type asHalfLine).
- New: TGPCanvas.DrawLine method.
- New: TLinkPoint.Visible property.
- New: TTextCell.Obj property.
- New: TatDiagram methods CanPaste and CanCopy.
- Improved: Bezier lines with a smoother look.
- Improved: TatDiagram CopyToClipboard, PasteFromClipboard and CutToClipboard methods now works with both selected objects and selected text.
- Improved: Copy/paste operations in designer now also works while editing text in blocks.
- Fixed: Several operations using selectores were not undoable nor notifying modifications in diagram.
- Fixed: Issue in machines without gdiplus32.dll installed.
- Fixed: Issue with background repainting in Delphi 2007 and above.
- Fixed: Issue with text cells being clipped wrongly.
- Fixed: Issue with side lines not being updated after loading diagram file in some situations.
- Fixed: Issue with hiding page lines in a machine with no printer installed.

Version 3.0 (Jul-2008)

- New: Overall drawing architecture redesigned to support GDI+ functions and provide more modern, Visio-like look and feel. (*)
- New: TDiagramControl.Transparency allows to specify the overall transparency for the diagram object (from 0% to 100%).
- New: TatDiagram.SmoothMode allows setting the anti-aliasing level (drawing in high quality).
- New: PNG, TIF, GIF and JPG images supported in blocks (when using GDI+ mode).
- New: TatDiagram.DragStyle allows more Visio-like dragging (objects are fully drawn when moved/resized, not only the frame cursor).
- New: TatDiagram.HandleStyles allows choose the visual style of the handles, between classic (small black squares) or Visio-like (medium-size green).
- New: TBlockShadow.Transparency allows to specify shadow transparency.
- New: gsRuler grid style for a more modern, Visio-like, ruler-attached snap grid.
- New: TatDiagram methods for pixel/unit convertion: ClientToInches, ClientToMilimeters, MilimetersToClient, InchesToClient.
- New: TDiagramControl.IsGdiPlus property specified is the object will be drawn using GDI+ functions.
- New: Diagram editor dialog now has a new toolbar selector for setting block transparency.
- New: Design-time context popup menu options to save/load diagram to/from file.
- New: TDgrTransparencySelector control for easy setting block transparency.
- New: TDiagramEditor.Title property.
- New: TDiagramEditor.OnCreateDesigner and TDiagramEditor.OnShowDesigner public events.
- New: TLiveDiagram protected virtual methods: DoBeforeExecuteNode and DoAfterExecuteNode.
- New: TLiveDiagram.MaximumIdle property specified maximum idle time for executing nodes.
- New: TLiveDiagram.RunErrorMsg property provides the description of error raised while executing the diagram.
- New: TStretchHandles.RotateHandle function retrieves the handle which performs block rotation.
- New: Property Anchors was made published in all selectors.
- Update: All existing core blocks (flowchart, electric, arrow, live diagram) updated to be drawn with more modern look (gdi plus functions).
- Improved: More modern look (and Visio-like look) in diagram editor dialog.

- Fixed: Preview page not being centered when resizing preview window.
- Fixed: Rare Access Violations while executing live diagrams.
- Fixed: NPTIGBT electric block being drawn incorrectly.
- Fixed: Issue with "No default printed selected" message.

(*) *GDI+ new features are not supported in C++ Builder 6.*

Version 2.4 (Sep-2007)

- New: TDiagramButtons control: a new diagram toolbar with a more Visio-like appearance. Only for Delphi version 2005 and higher.
- New: TDiagramAlign action, for making easy alignment and spacing of blocks.
- New: TDiagramEditor component. A ready-to-use and full-featured diagram editor including the new Visio-like toolbar (Delphi 2005 or higher) and new alignment palette.
- New: TatDiagram.KeyActions property.
- New: TTextCell.Visible property.
- New: TatDiagram.OnAfterDrawBlock event.
- New: TDiagramControl.IsBackgroundControl property.
- New: TDiagramControl.CalcSize virtual method.
- New: TSnapGrid.Force property.
- New: TLineArrow.DiagramLine property made public.
- New: AIgnoreZoom parameter in TatDiagram.HandleAtPos method.
- Improved demo: Now the demo just uses the TDiagramEditor component.
- Improved: Warnings removed.
- Fixed: Shadow of blocks were sometimes being drawn with wrong pen width.

Version 2.3 (Jul-2007)

- New: TatDiagram.ConnectionLineId method allows automatic linkpoint linking without need to choose a line object from the toolbar (see DiagramDemo).
- New: TatDiagram.PageLines property allows to make page lines visible to indicate the printed page size.
- New: TatDiagram.AutoPage property allows automatic redimension of diagram area according to the printed page size.
- New: TLiveDiagram save/load state mechanism. The state of the chart being executed can be saved and retrieved later by using LoadState methods and OnSaveState event. Any live block can signal the chart to be saved (by setting ExecInfo.WaitState property in OnExecuteEx event).

- New: LiveDiagram execution paths concept. A live diagram can have multiple execution paths, by the use of TLiveForkBlock (to split a single execution path in multiple execution paths) and TLiveJoinBlock (to join multiple execution paths in a single execution path).
- New: TDiagramControl.MakeVisible method.
- New: TatDiagram.MakeControlVisible method.
- New: PageUp/PageDown key support for scrolling the diagram area.
- New: TDiagramToolbar.ButtonSize property.
- New: dbLines in TDiagramToolbarButtons allows to make the diagram toolbar to show only lines (to be used in conjunction with TDiagramToolbar.Mode property.
- New: TDiagramToolbar.Mode property (tmObjects, tmConnections). It makes it possible to use the diagram toolbar to choose the line object for automatic connections (to be used in conjunction with TatDiagram.ConnectionLineId).
- New: kmLinesOnly item in TKeepInsertingMode for the diagram toolbar.
- New: TDiagramConnectionID action.
- New: AsText parameter in TatDiagram.LoadFromStream and TatDiagram.SaveToStream methods allows text format saving of diagram content.
- New: Public methods TDiagramControl.HasDefaultTextCell and TDiagramControl.DefaultTextCell.
- New: TLiveDiagram.OnExecuteEx event. This event passes a TExecuteNodeInfo object which gives more flexibility over the execution flow and state saving.
- New: LiveDiagram: TDiagramState type dsView shows the current state of live diagram chart.
- New: TLiveDiagram.MakeActiveNodeVisible shows the current node being executed in diagram.
- New: TLiveDiagram.ViewStateMode property specifies if the start, end or current executed node (block) will be highlighted.
- New: Live diagram fork block demo.
- Improved: Simple (main) demo has a connection toolbar for choosing automatic connections.
- Improved: Clipboard operations in live diagram now supported with default diagram actions.
- Fixed: OnDControlMouseUp not being called when right-button was clicked.

Version 2.2 (Apr-2007)

- New: TTextCell.CellFrame property with Pen, Brush, Color, Transparent, AutoFrame, AutoFrameMargin, Visible subproperties.
- New: TatDiagram.ShowCrossIndicators property - valid only for side lines.

- New: Added control restrictions: crKeepRatio, crNoClipboard, crNoSelect.
- New: TLinkPoint.LinkConstraint property.
- New: TatDiagram events: OnBeforeResize, OnAfterResize, OnDControlResizing.
- New: TDiagramArc.InvertOrientation method.
- New: unPixel option in TDiagramRuler.Units property.
- New: TatDiagram.IgnoreScreenDPI property.
- New: Diagram toolbar now works vertically.
- Fixed: Transparency not being saved or undoed.
- Fixed: RequiredConnections was not working properly.
- Fixed: Some images were not being displayed in print preview.
- Fixed: Minimum width and height of blocks are now being respected.

Special thanks to Phil Scadden.

Version 2.1 (May-2006)

- New: Live Diagram framework. A diagram descendant component which "runs" flowchart and statechart diagrams. Includes new TLiveDiagram component and several live flowchart blocks and live statechart blocks. Includes two new demos to ilustrate Live Diagram usage, and updated documentation.
- New: TDiagramControl.AutoCreateLinkPoints property.
- New: TCustomDiagramLine.RequiresConnections property.
- New: TatDiagram.PaintLinesFirst property.
- New: TDiagramControl.GetLinkPointClass method.
- New: TatDiagram.OnCloseEditor event.
- New: TatDiagram.OnDControlMouseDownEx event.
- New: TatDiagram.HandleAtPos method.
- New: TLinkPoint.Obj property.
- New: TCustomDiagramLine.GetLineArrowClass protected method.
- Improved: TatDiagram.DControlAtPos method.
- Improved: TDiagramControl.DrawCells method turned from static to virtual.
- Improved: TLineArrow.Draw method turned from static to virtual.
- Fixed: Several bug fixed, specially issues with zooming and scrolling while editing texts.

Special thanks to Davide Nardella and Martijn Tonies.

Version 2.0 (Dec-2005)

- New: TDgrPenStyleSelector, TDgrPenColorSelector, TDgrPenWidthSelector, TDgrBrushStyleSelector, TDgrShadowSelector, TDgrGradientDirectionSelector, TDgrColorSelector, TDgrTextColorSelector components which are ready-to-use, diagram-integrated buttons/combo boxes which allow easy edit of block properties like pen style, color and with, brush style, shadow, gradient, background color and font color.
- New: TDgrFontSelector, TDgrFontSizeSelector, TDgrZoomSelector components which are ready-to-use, diagram-integrated, ms office-like combo boxes for easy edit of font name and font size of block, and also set the zoom ratio of diagram.
- New: TatCustomDiagramBlock.Picture and PictureMode properties allow different image format in blocks, like JPG (using jpeg.pas unit) or any other format derived from TGraphic class.
- New: Methods StartZooming and EndZooming allows putting diagram in zoom mode, where end-user click the diagram to zoom, or select a zoom area by drawing a selection rectangle.
- New: TatDiagram.ExportBackgroundColor property allows defining a background color while exporting diagram to a bitmap or metafile.
- New: TCustomRuler.AutoFactor property allows disabling autofactoring of ruler divisions while zooming.
- New: TCustomDiagramBlock.RotationStep property allows changing the angle step while end-user is rotating a block.
- New: DrawBlockInCanvas method, which is a public version of DrawBlock method, allows drawing the block in a different canvas.
- Changed: Clicking objects with Shift key pressed was only selecting the object. Now it selects/deselects the object which as clicked.

Version 1.9 (09-Feb-2005)

- New: Polygon property and ChangePolygon method in polygon block allows defining the polygon progamatically.
- New: Properties LinkCursor and PanCursor.
- New: Added Layer and StringData properties for arrow blocks.
- New: Public method CloseEditor.
- New: Public property PaintStyle.
- New: Public method FindCompName.
- New: Property ClipboardNamePrefix.
- New: Now end-user can insert the same block multiple times, without having to click the toolbar button again to insert a new block (controlled with TDiagramToolbar.KeepInsertingMode property).

- New: ExportToFile method allows exporting diagram to bmp or wmf *(thanks to Ruediger Kabbasch)*.
- Improved: CopyBitmapToClipboard and CopyMetafileToClipboard: new parameter to export only diagram area (remove empty area around diagram).
- Changed: WriteText method is now virtual.
- Updated: help file with full reference for diagram and toolbar components.

Version 1.8.2 (30-Nov-2004)

- Delphi 2005 support added.

Version 1.8.1 (31-Oct-2004)

- Issue with parent window handling fixed.

Version 1.8 (23-Oct-2004)

- New: Expanding/collapsing nodes system. Linkpoints can be turned into nodes and nodes can be expanded/collapsed, showing/hiding the other blocks attached to the link point.
- New: TatDiagram.AutomaticNodes property controls this behaviour.
- New: TLinkPoint properties Collapsable and Collapsed controls individual behaviour of link points.
- New: Gradient background supported with new TatDiagram.Background.Gradient property.
- New: Layers system: added Layer property for diagram controls. Added ActiveLayer, DeactivateLayer and LayerActive methods in TatDiagram component.
- New: Visible property for diagram controls.
- New: TDiagramUndoAction and TDiagramRedoAction actions.
- New: TatDiagram events: OnBeforePaint and OnAfterPaint.
- New: TatDiagram event: OnDrawLinkPoint.
- New: StringData property for diagram controls.
- New: TatDiagram method CopyImageToMetaFile.
- Improved: TatDiagram.PaintToBitmap method turned public.
- Fixed: Problem while saving diagram if text block is being edited.
- Fixed: Bugs in copy/paste operations.
- Fixed: No OnInsertBlock and OnInsertLink events when pasting from clipboard.
- Fixed: OnModified event being fired on block selection.
- Fixed: DiagramSideLine position was not being saved.

- Fixed: Copy/Paste changes diagram properties.

Version 1.7 (06-Jul-2004)

- New arrow blocks: Standard, Double, Quad, Triple, Chevron, Block Single, Block Double, Corner Single, Corner Double *(thanks to Nick Glasier)*.
- New electric blocks: Zener Diode, Comparator, Inductor, Non-linear inductor, Ground, NPN/PNP Transistor, PTIGB, PIN, Thyristor, MOSFET, Switch, Duo Coil XForm, Tri Coil XForm, DC Voltage Source, DC Current Source *(thanks to Steve Evans)*.
- New: TatDiagram.Redo method.
- New: Gradient designer & action.
- New: Shadow designer & action.
- Improved: Undo behaviour.
- Solved: SaveToStream erroneously storing some DesignTime properties.
- Solved: Snap to grid was sometimes not snapping correctly.
- Solved: OnDControlMouseUp event was not being generated when a block was resized from larger to smaller size.
- Solved: Polyline insertion with snap to grid on was not being done correctly.

Version 1.6.2 (07-May-2004)

- New: Method TatDiagram.DControlAtPos.
- New: Property TatDiagram.MovingStartPoint.
- New: Events for TatDiagram: OnBeforeMove, OnAfterMove, OnMoving.
- New: Category tabs in design-time diagram toolbar.
- New: TCustomDiagramBlock.Resize virtual method.
- New: Stretch handle style: csCustom.
- Improved: Solved Problem with 'blk_tcustomdiagramblock' error message.
- Improved: AV after deleting blocks during OnMouseMove.

Version 1.6.1 (21-Apr-2004)

- New: Workflow demo shows how to use Diagram to integrate it to your application logic.
- New: TatDiagram.MouseWheelMode property.
- New: TatDiagram events: OnBlockMouseEnter e OnBlockMouseLeave.
- New: Method TatDiagramControl.PointInControl.
- Fixed: Minor bug when printing arrows (arrow size was wrong).

- Fixed: Minor bug: Changing TDiagramSideLine.Orientation1 property was not updating diagram.

Version 1.6 (08-Apr-2004)

- Polygon block.
- Arc and Bezier lines.
- Metafile support in blocks (metafiles do not rotate in W95/98/Me systems).
- New TatDiagram events: OnDrawShape, OnGetSurroundPts, OnAcceptAnchor, OnAcceptLink, OnRemoveDControl.
- New TatDiagram properties: HorzScrollBar and VertScrollBar. These new properties allow setting a fixed diagram size (width/height), by changing Range property of scroll bars.
- New TLinkPoint properties: AnchoredCount and Anchoreds.
- New block restriction: noDelete.
- New TatDiagram methods: ClientToCanvas and CanvasToClient.
- Panning behaviour corrected.
- Fixed visual bug with scroll bars.
- Fixed bug with transparent bitmap drawing.
- Removed protected property TCustomDiagramLine.Points.

Version 1.5.1

- New panning support.
- Fixed issue with line captions.
- Fixed issue with line dragging.
- Fixed issue with copy as image.

Version 1.5

- New in the Diagram Studio components:
    - TDiagramPolyLine object;
    - TDiagramControl.TextCells property - blocks can have multiple text cells;
    - Captions (text cells) for line objects;
    - Improved text editing when vertical alignment is set to vaCenter or vaBottom;
    - Public TCustomDiagramLine.Handles property allow to set lines position from code;
    - New diode and capacitor blocks;
    - BringToFront and SendToBack methods;

    - Keyboard events OnKeyPress, OnKeyDown and OnKeyUp added;
    - Multi-purpose TDiagramControl.Obj property added;
    - Hint, ShowHint, ParentShowHint properties in blocks;
    - Improved print preview rendering;
    - Print button in preview now opens printer setup dialog;
    - Cursor property in diagram controls;
    - TAction descendant objects for easy add interface actions to diagram applications;
    - New actions supported: Show/hide left ruler, show/hide top ruler, show/hide snap grid, change shape (block) color, change line (pen) color, block picture, bring to front, send to back, change arrow shape, change text font;
    - BorderColor property.
- New flowchart blocks: TFlowCommentBlock, TFlowListBlock, TDatabaseBlock.
- New Electric blocks: TCapacitorBlock, TLampBlock, TDiodeBlock.
- Minor bug fixes.
- New in the Diagram Studio demo: added popup menu, toolbar and menu options for
    - Change block color;
    - Change line color;
    - Change block picture;
    - Bring to front;
    - Send to back;
    - Change arrow shape;
    - Change text font;
    - Show/hide grid.
- Added categories tab in diagram objects toolbar.

Getting Support

General notes

Before contacting support:

- Make sure to read the tips, faq and readme.txt or install.txt files in component distributions.
- Make sure you have the latest version of the component(s).

When contacting support:

- Specify with which component you have a problem.
- Specify which Delphi or C++Builder version you're using and preferably also on which OS.
- In case of IntraWeb or ASP.NET components, specify with which browser the issue occurs.
- For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee the fastest route to a solution.

Send email from an email account that

1. allows to receive replies sent from our server;
2. allows to receive ZIP file attachements;
3. has a properly specified & working reply address.

Getting support

For general information: info@tmssoftware.com

Fax: +32-56-359696

For all questions, comments, problems and feature request for VCL components : help@tmssoftware.com.

To improve efficiency and speed of help, refer to the version of Delphi, C++Builder, Visual Studio .NET you are using as well as the version of the component. In case of problems, always try to use the latest version available first.

Breaking Changes

List of changes in each version that breaks backward compatibility.

Version 4.9.1

Unit names have been changed:

- AutoLayout.Diagram → AutoLayoutDiagram
- AutoLayout.Executer → AutoLayoutExecuter
- AutoLayout.Force → AutoLayoutForce

for backward compatibility with old Delphi versions (Delphi 7 up to Delphi/C++ Builder 2007).

Version 4.8

TMS Diagram packages have been restructured. The packages are now separated into runtime and design-time packages, allowing a better usage of the packages in an application using runtime packages (allows it to work with 64-bit applications using runtime packages, for example). Also, Libsuffix option is now being used so the dcp files are generated with the same name for all Delphi versions. Here is an overview of what's changed:

Before version 4.8, there was a single package named adgrm<version>.dpk (where <version> is the "name" of delphi version), which generated BPL and DCP with same names:

Previous versions:

Version	Package File Name	BPL File Name	DCP File Name
Delphi 7	adgrm7.dpk	adgrm7.bpl	adgrm7.dcp
Delphi 2007	adgrm2007.dpk	adgrm2007.bpl	adgrm2007.dcp
Delphi 2009	adgrm2009.dpk	adgrm2009.bpl	adgrm2009.dcp
Delphi 2010	adgrm2010.dpk	adgrm2010.bpl	adgrm2010.dcp
Delphi XE	adgrm2011.dpk	adgrm2011.bpl	adgrm2011.dcp
Delphi XE2	adgrmxe2.dpk	adgrmxe2.bpl	adgrmxe2.dcp
Delphi XE3	adgrmxe3.dpk	adgrmxe3.bpl	adgrmxe3.dcp
Delphi XE4	adgrmxe4.dpk	adgrmxe4.bpl	adgrmxe4.dcp
Delphi XE5	adgrmxe5.dpk	adgrmxe5.bpl	adgrmxe5.dcp
Delphi XE6	adgrmxe6.dpk	adgrmxe6.bpl	adgrmxe6.dcp
Delphi XE7	adgrmxe7.dpk	adgrmxe7.bpl	adgrmxe7.dcp

From version 4.8 and on, there are two packages:

- TMSDiagram.dpk (runtime package)
- dclTMSDiagram.dpk (design-time packages)

DCP files are generated with same name, and only BPL files are generated with the suffix indicating the Delphi version. The suffix, however, is the same used by the IDE packages (numeric one indicating IDE version: 160, 170, etc.). The new package structure is as following:

Version	Package File Name	BPL File Name	DCP File Name
Delphi 7	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram70.bpl dclTMSDiagram70.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi 2007	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram100.bpl dclTMSDiagram100.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi 2009	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram120.bpl dclTMSDiagram120.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi 2010	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram140.bpl dclTMSDiagram140.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi XE	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram150.bpl dclTMSDiagram150.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi XE2	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram160.bpl dclTMSDiagram160.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi XE3	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram170.bpl dclTMSDiagram170.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi XE4	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram180.bpl dclTMSDiagram180.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi XE5	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram190.bpl dclTMSDiagram190.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi XE6	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram200.bpl dclTMSDiagram200.bpl	TMSDiagram.dcp dclTMSDiagram.dcp
Delphi XE7	TMSDiagram.dpk dclTMSDiagram.dpk	TMSDiagram210.bpl dclTMSDiagram210.bpl	TMSDiagram.dcp dclTMSDiagram.dcp

Version 4.0

Major changes in the drawing system. More info below.

Upgrade Notes: Diagram Studio 4.0 introduces several changes from previous versions. Those changes might cause compiling error in existing code using Diagram Studio, requiring you to adjust your source code.

Most changes are only at syntax level, so to make your code work you don't need to fully understand why the changes were made, but only how to change your source code to make it compatible. So to allow you to quickly change your source code we provide here a quick guide with the most compiling problems you might encounter, and how to quickly adjust your source code. More details about the changes and how/why they were done are explained later in the chapter.

Increased position precision - from integer to float

One of the major changes in Diagram Studio 4.0 is the improvement in the internal calculations. Types like TRect, TPoint and several Integer fields and properties were changed to TSquare, TDot and Double types, so that all coordinates and internal variables were changed from integer types to floating point types. This increased calculation precision and solved some problems with objects being positioned in a slightly different position where they should be.

We will not list here all variables, parameters and methods changed, because they were many. We will list here mostly what has been changed that might cause some incompatibility, and later we will describe common pieces of code that are broken in current version and how to solve it.

Main changes:

1. Many properties, variables, function results and parameters had their types changed this way:

   a. From TRect to TSquare

   b. From integer to Double

   c. From TPoint to TDot

   d. From TPointArray to TDotArray

So if your existing code breaks, take into consideration that you might need to change the data types of your variables, or make your code compatible with the new types of some Diagram Studio properties and variables. Check the "Quick guide for upgrading from Diagram 3.5 and earlier versions" chapter to see the most common incompatibilities that you might encounter in your code.

Quick guide for upgrading from Diagram 3.5 and earlier versions

List of common compile errors and how to fix them.

Problem #1

Overridden method signature differs from ancestor declaration.

Solution: change method signature to match the new correct signature. The list below contains all methods that have their signature changed, with the previous signature and the new one you must use.

In *TDiagramControl* class:

```
function ControlRect: TRect; //Old
function ControlRect: TSquare; //New

procedure DrawCell(ACanvas: TCanvas; ACell: TTextCell); //Old
procedure DrawCell(AInfo: TDiagramDrawInfo; ACell: TTextCell); // New

procedure DrawCell(ACanvas: TCanvas); //Old
procedure DrawCell(AInfo: TDiagramDrawInfo); //New

procedure DrawHandle(ACanvas: TCanvas; AHandle: TStretchHandle); //Old
procedure DrawHandle(AInfo: TDiagramDrawInfo; AHandle: TStretchHandle); //New

procedure DrawHandles(Canvas: TCanvas); //Old
procedure DrawHandles(AInfo: TDiagramDrawInfo); //Old

function GetHandlePoint(AHandle: TStretchHandle): TPoint; //Old
function GetHandlePoint(AHandle: TStretchHandle): TDot; //New

function GetLinkPoint(ALinkPoint: TLinkPoint): TPoint; //Old
function GetLinkPoint(ALinkPoint: TLinkPoint): TDot; //New

procedure PaintControl(Canvas: TCanvas; ARect: TRect); //Old
procedure PaintControl(AInfo: TDiagramDrawInfo); //New

function SurroundRect: TRect; //Old
function SurroundRect: TSquare; //New
```

In *TCustomDiagramBlock* class:

```
procedure DrawBlock(Canvas: TCanvas; ARect: TRect); //Old
procedure DrawBlock(AInfo: TDiagramDrawInfo; ABlockInfo:
TDiagramDrawBlockInfo); //New

procedure DrawBlockCursor(Canvas: TCanvas; ARect: TSquare; AAngle: double); //Old
procedure DrawBlockCursor(AInfo: TDiagramDrawInfo; ABlockInfo: TDiagramDrawBlockI
nfo); //New

procedure DrawShape(Canvas: TCanvas; ARect: TRect; AAngle: double); //Old
procedure DrawShape(AInfo: TDiagramDrawInfo; ABlockInfo:
TDiagramDrawBlockInfo); //New

function SurroundRgn: TPointArray; //Old
function SurroundRgn: TDotArray; //New
```

In *TCustomDiagramLine* class:

```
procedure DrawLine(Canvas: TCanvas; AHandles: TStretchHandles; DrawArrows: boolea
n); //Old
procedure DrawLine(AInfo: TDiagramDrawInfo; ALineInfo: TDiagramDrawLineInfo); //
New
```

In *TLineArrow* class:

```
procedure Draw(Canvas: TCanvas; AFrom, ATo: TPoint; AZoomRatio: double); //Old
procedure Draw(AInfo: TDiagramDrawInfo; AArrowInfo: TDiagramDrawArrowInfo); //New
```

Problem #2

Undeclared identifier: 'Canvas' (in TDiagramControl methods with changed signature like PaintControl, DrawBlock etc.). See problem 1.

Example (removed parameter Canvas: TCanvas):

```
Canvas.LineTo(x1, y1);
```

Solution: replace any Canvas reference by AInfo.Canvas.

Example (new parameter AInfo: TDiagramDrawInfo):

```
AInfo.Canvas.LineTo(x1, y1);
```

Problem #3

Undeclared identifier: 'ARect' (in TCustomDiagramBlock methods with changed signature like DrawBlock, DrawShape etc.). See problem 1.

Example (removed parameter ARect: TRect):

```
Drawer.CurRect := ARect;
```

Solution: replace any ARect reference by ABlockInfo.Rect.

Example (new parameter ABlockInfo: TDiagramDrawBlockInfo):

```
Drawer.CurRect := ABlockInfo.Rect;
```

Problem #4

Undeclared identifier: 'AAngle' (in TCustomDiagramBlock methods with changed signature).

Parameter AAngle was removed from some virtual methods like DrawShape and DrawBlockCursor. Information comes now in 'Angle' field of new record parameter (ABlockInfo: TDiagramDrawBlockInfo). See problem 1.

Solution: replace any AAngle reference by ABlockInfo.Angle.

Problem #5

Undeclared identifier: 'AHandles' or 'DrawArrows' (in TCustomDiagramLine methods with changed signature).

Parameters AHandles and DrawArrows were removed from virtual method DrawLine. Information comes now in fields of new record parameter (ALineInfo: TDiagramDrawLineInfo). See problem 1.

Solution: replace any AHandles reference by ALineInfo.Handles and DrawArrows by ALineInfo.DrawArrows.

Problem #6

Undeclared identifier: 'AFrom', 'ATo' or 'AZoomRatio' (in TLineArrow methods with changed signature).

Parameters AFrom, ATo and AZoomRatio where removed from virtual method Draw. Information comes now in fields of new record parameter (AArrowInfo: TDiagramDrawArrowInfo). See problem 1.

Solution: replace any AFrom reference by AArrowInfo.FromPoint, ATo by AArrowInfo.ToPoint and AZoomRatio by AArrowInfo.ZoomRatio.

Problem #7

Incompatible types: 'TSquare' and 'TRect' (using TRect when TSquare is expected).

Example:

```
Drawer.OriginalRect := Rect(0, 0, 100, 100);
```

Solution: Replace TRect by TSquare. If needed change integer parameters by floating point parameters. You can use Square function in place of Rect function.

Example:

```
OriginalRect := Square(0, 0, 100, 100);
// OriginalRect is a new TDiagramControl property. See tip in problem 14.
```

Problem #8

Incompatible types: 'TRect' and 'TSquare' (using TSquare when TRect is expected).

Example:

```
Canvas.FillRect(ABlockInfo.Rect);
Canvas.MoveTo(ABlockInfo.Rect.Left, ABlockInfo.Rect.Top);
```

Solution: Use ToRect function to convert a TSquare back to a TRect, when needed. Some methods still use TRect, but since many TRect values were replaced by TSquare (see problem 7), you might need to convert it back. Maybe will be needed also convert double parameters back to integer parameters.

Example:

```
var R: TRect;
R := ToRect(ABlockInfo.Rect);
Canvas.FillRect(R);
Canvas.MoveTo(R.Left, R.Top);
```

Problem #9

Incompatible types: 'TDotArray' and 'TPointArray'.

Example:

```
RotPoly(PointArray([PP(40,85), PP(60,85), PP(50,100), PP(40,85)]);
```

Solution: Replace TPointArray (array of TPoint) by TDotArray (array of TDot). If needed change integer parameters by floating point parameters. You can use DotArray in place of PointArray function.

Example:

```
RotPoly(DotArray([PP(40,85), PP(60,85), PP(50,100), PP(40,85)]);
```

If you had a BlockPolygon method that was returning a TPointArray, you may need change it to return a TDotArray.

Some TPoint functions/methods and its new TDot corresponding:

```
AddPoint => AddPointX
ConcatPoints => ConcatPointsX
Rot => RotX
SubPoint => SubPointX
```

Problem #10

Incompatible types: 'Array' (TPointArray) and 'TDotArray'.

Example:

```
AInfo.Canvas.Polygon(BlockPolygon);
```

Solution: Use RoundDotArray function to convert a TDotArray back to a TPointArray, when needed. Some methods still use TPointArray, but since you replaced TPointArray by TDotArray (see problem 9), you might need to convert it back.

Example:

```
AInfo.Canvas.Polygon(RoundDotArray(BlockPolygon));
```

Problem #11

Incompatible types: 'TPoint' and 'TDot'.

Example:

```
var P: TPoint;
P := ClientToCanvas(Point(10, 1));
```

Solution: Replace TPoint by TDot. If needed, change integer parameters by floating point parameters. You can use Dot function in place of Point function.

Example:

```
var P: TDot;
P := ClientToCanvas(Dot(10, 1));
```

Problem #12

Operator not applicable to this operand type.

Example:

```
LinkPoints.Add((Right - Left) div 2, Top, aoUp);
```

Solution: replace integer operators by floating point operators.

Example:

```
LinkPoints.Add((Right - Left) / 2, Top, aoUp);
```

Problem #13

Undeclared identifier: bmSolid/bmClear/etc. (when using GPCanvas.Brush for GDI+).

Solution: uses `DgrClasses` unit (add it to your unit's uses clause). The new type TDgrBrushMode is declared in that unit now.

Problem #14

Symbol TPointX/TRectX is deprecated.

Solution: use TDot/TSquare instead.

> **TIP**
> Instead of set both Drawer.OriginalRect and GPDrawer.SourceRect properties, now you can just set TDiagramControl.OriginalRect property, so that all used Drawers will be up to date.

Example:

```
// before
  Drawer.OriginalRect := Rect(0, 0, 100, 80);
  {$IFDEF GDIPLUS}
  GPDrawer.SourceRect := RectX(0, 0, 100, 80);
  {$ENDIF}

// now
  OriginalRect := Square(0, 0, 100, 80);
```

Problem #15

Method GetBlockPath/GetTranslatedBlockPath/GetSurroundBlockPath is deprecated.

Solution: override new generic methods for path drawing. Old ones were specific to GDI+, while the new ones are applicable to both GDI+, Direct2D, and other libraries that might be supported in future.

```
procedure GetBlockPath(APath: TDgrGraphicsPath; ADrawer: TDgrBlockDrawer);
procedure GetTranslatedBlockPath(APath: TDgrGraphicsPath; ADrawer: TDgrBlockDrawe
r);
procedure GetSurroundBlockPath(APath: TDgrGraphicsPath; ADrawer:
TDgrBlockDrawer);
```

Problem #16

Incompatible types: 'TDiagramDrawInfo' and 'TCanvas' (diagram events).

This error can occur in some event handler methods of TatDiagram, that have their signature modified (TCanvas replaced by TDiagramDrawInfo).

Solution: change method signature to match the new correct signature. The list below contains all events whose handler method must be adjusted.

```
OnAfterDrawBlock: TDrawBlockEvent
OnDrawBlock: TDrawBlockEvent
OnDrawShape: TDrawShapeEvent
```

TDrawShapeEvent type was removed and now these three events are TDrawBlockEvent. Below the previous signatures and the new one you must use.

```
TDrawBlockEvent = procedure(Sender: TObject; ABlock: TCustomDiagramBlock;
  Canvas: TCanvas; ARect: TRect; var APainted: boolean) of object; //Old
TDrawShapeEvent = procedure(Sender: TObject; ABlock: TCustomDiagramBlock;
  Canvas: TCanvas; ARect: TRect; AAngle: double; var APainted: boolean) of
object; //Old

TDrawBlockEvent = procedure(Sender: TObject; ABlock: TCustomDiagramBlock;
  AInfo: TDiagramDrawInfo; ABlockInfo: TDiagramDrawBlockInfo;
  var APainted: boolean) of object; //New
```

Direct2D support and generic drawer classes

The 4.0 version of Diagram Studio introduces some generic classes for drawing blocks with both GDI+ and Direct2D. GDI+ was already supported but the new version now supports usage of Direct2D (available for Delphi 2010 and later, supported only in Windows Vista and later). To use Direct2D in your custom blocks you must use the new drawing classes provided by Diagram Studio, so some code tuning might be needed. The new classes are very similar to the ones provided in earlier version for GDI+ usage, so if you have existing code that already paint blocks using GDI+, you can adapt your code to use Direct2D very easily. The good part is that once your custom painting code uses the new classes, you can easily switch between Direct2D and GDI+. And in future if more graphic libraries are supported by Diagram Studio, your existing code will be compatible. There is no need to use specific code for GDI+, Direct2D, and others. (Please note that drawing with GDI still requires specific code).

Compatibility with existing code for drawing with GDI+ was partially kept (although a lot of functions and methods have been marked as deprecated), so if you don't want to support Direct2D now you don't have to change your code to use the new drawing classes, although it's highly recommended. If you want diagram to keep using only GDI+, see the topic about graphic libraries.

Here we will list the main changes that have been done in Diagram Studio in respect to Direct2D support, changes made to Diagram Studio source code, and comments about what you need to change in your existing source code. It's also highly recommended that you check the topic "Quick guide for upgrading from Diagram 3.5 and earlier versions", which lists the most common changes you need to do in your source code to be compatible with the new Diagram Studio.

Units in uses clause

Units `DgrGdipApi`, `DgrGdipObj` and `GdipClasses` are specific for GDI+ drawing. You can replace them by the new unit `DgrClasses`, which contains the generic drawing classes. You don't need to put `DgrClasses` under the {$IFDEF GDIPLUS} directive, so if the directive is there, you can removed it.

Overview of new generic drawing classes

- Abstract/base classes (unit `DgrClasses`): TDgrBlockDrawer, TDgrCanvas, TDgrGraphicsPath, TDgrPaintEngine.
- GDI+ drawing classes (unit `GdipClasses`): TGPBlockDrawer, TGPCanvas, TGdipGraphicsPath, TGdipPaintEngine.
- Direct2D drawing classes (unit `Direct2DClasses`): TD2DBlockDrawer, TD2DCanvas, TD2DGraphicsPath, TD2DPaintEngine.

GDI+ and Direct2D classes descend from the base ones.

Signature changed for paint/draw methods

Several drawing methods (either virtual or static) have their signature changed. The reason was that many methods received parameters like Canvas, Rect, etc., which only made sense for GDI drawing. Now the methods receive a parameter of type TDiagramDrawInfo, which is a record type with lots of information for drawing the block, including the parameters removed from method signature. Here is a piece of TDiagramDrawInfo declaration, used in TDiagramControl methods:

```
TDiagramDrawInfo = record
  ...
  // Canvas Object of diagram
  Canvas: TCanvas;
  // Instance for a TBlockDrawerClass to help in the drawing of controls.
  Drawer: TBlockDrawer;
  // Instance for a TDgrBlockDrawer descendant to help in the drawing of
controls using
  // specific GraphicLib functions (GDI+/TGPBlockDrawer and Direct2D/
TD2DBlockDrawer)
  DgrDrawer: TDgrBlockDrawer;
  ...
end;
```

Note that the record contains references to a TBlockDrawer and TDgrBlockDrawer instances. It's recommended that you use those objects to perform block drawing. For compatibility reasons, the properties Drawer and GPDrawer were kept in TDiagramControl class, but it's recommended that you use those in the record (using DgrDrawer in place of GPDrawer, since the latter is GDI+ specific while the former is generic).

Along with TDiagramDrawInfo, additional parameters of TCustomDiagramBlock methods were put together in a TDiagramDrawBlockInfo record:

```
TDiagramDrawBlockInfo = record
  Rect: TSquare;
  Angle: double;
end;
```

Similarly, TDiagramDrawLineInfo record is used by TCustomDiagramLine methods:

```
TDiagramDrawLineInfo = record
  Handles: TStretchHandles;
  DrawArrows: boolean;
end;
```

For a complete list of methods changed, check the "Quick guide for upgrading from Diagram 3.5 and earlier versions".

New virtual methods for specifying block shape using paths

There are three new virtual methods in TCustomDiagramBlock component for retrieving the shape of blocks using paths. The methods are generic and used for both GDI+ and Direct2D:

```
GetBlockPath(APath: TDgrGraphicsPath; ADrawer: TDgrBlockDrawer);
GetTranslatedBlockPath(APath: TDgrGraphicsPath; ADrawer: TDgrBlockDrawer);
GetSurroundBlockPath(APath: TDgrGraphicsPath; ADrawer: TDgrBlockDrawer);
```

Note that the new methods have the same names as the methods previously used for GDI+ only. To keep compatibility, the old GDI+ methods were kept as overloaded methods (but marked as deprecated), and if you just want to use GDI+ you can keep your code using those methods, there is no need to replace them by the generic ones. Here are the specific GDI+ methods kept:

```
GetBlockPath(APath: TGPGraphicsPath);
GetTranslatedBlockPath(APath: TGPGraphicsPath);
GetSurroundBlockPath(APath: TGPGraphicsPath);
```

Property OriginalRect

When creating new custom blocks, you needed to specify the original rectangle of the drawing. You needed to do that for Drawer (GDI drawing) and GPDrawer (GDI+ drawing) objects, as showed in this example:

```
Drawer.OriginalRect := Rect(0, 0, 100, 100);
{$IFDEF GDIPLUS}
GPDrawer.SourceRect := RectX(0, 0, 100, 100);
{$ENDIF}
```

Now you just need to set the new property OriginalRect (in TDiagramControl). By setting OriginalRect property all the available drawers are updated regardless of each graphic library is being used. So the previous code can be replaced by:

```
OriginalRect := Square(0, 0, 100, 100);
// note that Rect and RectX were replaced by Square function
```

Property PathDrawingMode

In previous versions, if you wanted to know if GDI or GDI+ was being used to perform the drawing, you used the property IsGdiPlus. Now there is a new boolean property named PathDrawingMode which indicates if the drawing mode is being done using standard GDI (direct painting to Canvas) or using the newly created generic classes (GDI+, Direct2D) which is based on graphic paths for performing drawing. So, in general, this construction:

```
if IsGdiPlus then
  GpDrawer.SomeDrawingMethod;
```

Use this:

```
if PathDrawingMode then
  DgrDrawer.SomeDrawingMethod;
```

Converting specific GDI+ code to generic (GDI+/Direct2D) code

The new generic drawing classes were built to be very similar to the existing classes for specific GDI+ drawing. It was done this way so that you can easily change your code to use the generic classes in order to get benefit of Direct2D drawing.

Basically the new classes map to existing GDI+ classes. For example, if you use TGPCanvas or TGPDrawer objects, just change them to TDgrCanvas or TDgrDrawer and your code will be almost done. The methods and properties in the new classes were built to have same names, signatures and types of the GDI+ classes. The only exceptions cases in which you will need to fine tune your source code will be listed here.

a. The new class TDgrGraphicsPath is used instead of TGPGraphicsPath. So in old GDI+ methods where you used TGPGraphicsPath objects, now use the new one TDgrGraphcisPath.

b. Methods AddArc, AddBezier, AddEllipse and AddLine have four different overloaded signatures in GDI+ classes. In the new generic class only one signature is available, so if you are using one of the other overloaded versions of the method in your old GDI+ code, you will need to adapt them to use one of these methods in TDgrCanvas:

```
function AddArc(x, y, width, height, startAngle, sweepAngle: number): integer;
function AddBezier(x1, y1, x2, y2, x3, y3, x4, y4: number): integer;
function AddEllipse(x, y, width, height: number): integer;
function AddLine(x1, y1, x2, y2: number): integer;
```

c. Methods AddLines e AddPolygon, in old GDI+ classes, receive two parameters: a pointer to an array of TGPPoint, and the number of elements in array. In the new generic class only one parameter is received: an array of TDot.

```
function AddLines(points: TDotArray): integer;
function AddPolygon(points: TDotArray): integer;
```

Example

```
// old GDI+ classes
  path.AddLines(PGPPoint(@lines[0]), 10);

// new generic class
  path.AddLines(lines); //lines is a TDotArray
```

d. Method AddRectangle, in old GDI+ classes, receives a TGPRect or a TGPRectF as parameter. In new generic class you must provide a TSquare parameter:

```
function AddRectangle(rect: TSquare): integer;
```

e. Method SetFillMode, in old GDI+ classes, receives a parameter of type DgrGdipApi.FillMode. The new generic class receives a parameter of type Graphics.TFillMode = (fmAlternate, fmWinding).

f. In your block paint methods, instead of using a reference to GPDrawer (of type TGPDrawer), use the new class type TDgrDrawer. Usually a reference to a TDgrDrawer object is available in a new parameter "AInfo: TDiagramDrawInfo" passed in drawing methods like DrawBlock, DrawShape, etc. In summary:

```
GPDrawer => AInfo.DgrDrawer
GPDrawer.GPCanvas => AInfo.DgrDrawer.Canvas
```

Methods that handle paths (like for example GetBlockPath or GetTranslatedBlockPath) already receive a parameter "ADrawer: TDgrBlockDrawer", which should be used instead of GPDrawer.

g. Path creation, in old GDI+ classes, was done directly but instantiating a TGPGraphicsPath. For example:

```
{$IFDEF GDIPLUS}
path := TGPGraphicsPath.Create;
{$ENDIF}
```

In new generic classes you must use the CreatePath method. This method will take care of creating an instance of the correct class. The result class will be the generic TDgrGraphicsPath, but it can be a TGPGraphicsPath if the drawing library being used is GDI+, or can be TD2DGraphicsPath if the drawing library used is Direct2D.

Example:

```
if PathDrawingMode then
  path := CreatePath;
```

So, for example, inside a method that receives "AInfo: TDiagramDrawInfo" as a parameter you can use this:

```
path := AInfo.DgrDrawer.Canvas.CreatePath;
```

Graphic libraries: GDI, GDI+ and Direct2D

Diagram Studio 4.0 supports GDI, GDI+ and Direct2D libraries for painting of controls, and allows choosing the used library at runtime, by setting a property, either for the whole diagram (TatDiagram.GraphicLib property) or a specific block (TDiagramControl.GraphicLib property).

The property GraphicLib is of TDgrGraphicLib type (declared in unit DgrClasses) and can be one of following values:

- *dglGDI*: GDI
- *dglGDIPlus*: GDI+
- *dglDirect2D*: Direct2D (available for Delphi 2010 and later)

Diagram Studio determines at runtime which library is used by default (see DefaultGraphicLib function in DgrClasses.pas): if application is compiled in Delphi 2010 or later and is running in an environment that supports Direct2D (Windows Vista or later), then Direct2D library is used by default; otherwise diagram uses GDI+ (unless GDI+ has been removed by conditional defines, in this case uses GDI).

WARNING

Existing source code for drawing of diagram blocks using specific GDI+ features will not work when diagram is using Direct2D library.

To force the use of GDI+, set GraphicLib property for dglGDIPlus at startup, in the specific block (TDiagramControl) or globally in diagram (TatDiagram).

To adapt your code to work with both GDI+ and Direct2D, see "Converting specific GDI+ code to generic (GDI+/Direct2D) code" section in "Direct2D support and generic drawer classes" topic.