



TeeChart Pro 4 VCL version Developer Guide (TeeLoper)

Rev 2. - July 1998



www.teemach.com

*TeeChart Copyright © 1995-98 by David Berneda.
All Rights Reserved.*

Contents

1.....	Foreword & History	32.
43.....	TeeChart Internals / Design Overview	5a) The Charting
TCustomChart.....		5
TDBChart.....		6
TQRChart.....		6
A fourth chart component developed by Borland Intl.....		6
b) The Series components.....		6
ParentChart.....		7
Streaming.....		7
Creating new Series at runtime.....		7
Assigning Series.....		8
c) The Function component.....		8
What is it ?.....		8
How it works.....		8
4.....	Creating custom Chart components	9
Changing default property values.....		10
Installing the new Chart.....		10
Extending TMyChart.....		10
Protected parts.....		11
Zoom and Scroll.....		13
The Draw sequence.....		14
Titles and Legend.....		14
Series Z Order.....		15
The Chart Rectangle.....		16
Painting the Walls.....		17
Drawing the Axis.....		17
Drawing the Series.....		17
The End of the Draw sequence.....		18
User Mouse clicks.....		18
Printing Charts.....		21

Clipboard.....	22
Exporting.....	23
5..... Creating custom Function components	24
Implementing TMyFunction.....	24
Calculate.....	25
CalculateMany.....	26
Registering your new Functions:.....	27
6..... TeeChart Runtime Editor interface	28
Editing a DBChart.....	28
Cloning a Series.....	28
Changing a Series type.....	29
ChangeAllSeriesType.....	29
Naming Series.....	29
The Chart Gallery.....	30
7..... Integrating TeeChart with your VCL	32
Deriving.....	32
Linking or Embedding ?.....	32
8..... Appendixes	35
Appendix A- T	
Appendix B- Unit and Files Organisation.....	37

1. //Foreword & History

TeeChart started as a custom charting component for a special application I developed.

Special because the customer was (is !) a very good friend and because it was my first “big” application made in Delphi 1.0, after being in programming since 1987 in Pascal, C++ and Paradox's ObjectPal.

After finishing the project and seeing the customer was very happy with it, I thought it would be great to upload it to Compuserve Delphi forum.

Having posted some freeware components and tools in the past (under Compuserve address 100115,1155), I decided to upload TeeChart 1.0 as my first shareware component, pricing it quite cheap (\$40), and hoping to get some extra money to force myself to continue developing and improving it.
This was back in March 1996.

The results were quite impressive. In one month there were more than 100 users and I had to rush porting TeeChart to the just arrived Delphi 2.0 and integrate it with QuickReport 1.0 to cover the increasing demand from Borland'ers all around the net.

The rest is just history, hard work and a big special surprise from Borland. They were interested in TeeChart and this successfully ended up with their acceptance to include TeeChart 3.0 standard version in Delphi version 3.0.

I wish to thank publicly to everybody who made TeeChart possible, starting with my wife Marga Bielsa, our friend and “special” customer Cesc Marrugat, our friend Marc Meumann for joining us on this ship, QuickReport's daddy Allan Lochert for helping me that much (“help overflow” I'd say), the whole Delphi team at Borland (Inprise) for their patience with me and my English !, and to all TeeChart'ers who made the product possible with their ideas and their economic support.
Thanks to everybody.

I hope you'll enjoy using TeeChart 4.0 and it's next upcoming versions.

David Berneda (aka “Pio”)
July 1998

2. Introduction

This guide is aimed for the intermediate to advanced Delphi developer, who knows what is inside a component, how to create components and who understands Delphi's Object Pascal terminology.

Please refer to Borland Delphi's Object Pascal Guide and Component Writers Guide for more information on these topics.

This guide is a complement to TeeChart's manual and help file. The main purpose of creating this guide is to explain to you the internals as much as possible, to help you in creating new Chart, Series and Function components and simply to provide a more in-depth documentation about TeeChart.

As usually, all topics documented here are subject to change in subsequent versions, although the goal is to leverage porting your code from TeeChart version 3 to future ones.

This guide is structured with the following sections:

- Chapter 3 describes TeeChart design providing a birds-eye-view of TeeChart components.
- Chapters 4, 5 and 6 document the three main TeeChart groups (Chart, Series and Functions) explaining their methods and properties in more detail to ease creation and installation of your custom charting components.
- Chapter 7 contains the runtime Chart Editor and Gallery interface so you can access them in your projects.
- Chapter 8 is oriented to the Delphi 3rd party VCL developer who might want to integrate TeeChart with their Delphi VCLs.
- Appendix A includes the TeeChart object classes' tree diagram.
- Appendix B enumerates and describes in short all TeeChart files
- Appendix C is a preliminary explanation of Delphi 3 issues.

New information, examples, frequently asked questions and 3rd party tools and source code will be posted regularly at teeMach web site (www.teemach.com).

Please contact us via email (support@teemach.com) if you want to contribute your TeeChart work for all the other TeeChart'ers !

3. TeeChart Internals / Design Overview

TeeChart consist of three families of components:

- The Charting panels (TChart, TDBChart and TQRChart)
- The Series components (TLineSeries, TBarSeries, etc)
- The Function components (TAddFunction, TAverageFunction, etc)

a) The Charting panels

The Charting panels form a hierarchy deriving from the abstract TCustomChart component, which derives from Delphi's TCustomPanel.

The chart is designed to be a “container” for Series and Functions, or in Delphi terms, the chart is the Parent for the Series.

This is very similar to what a TDataSet does with TField components, or what a TPageControl does with TTabSheets. The Chart is not intended to “own” the Series and Functions. Series and Functions are primarily intended to be owned by the Form.

Chart components add special features to the Panel ancestor, for example they add an internal bitmap used as a “buffer” for drawing operations to avoid flickering and to speed drawing calls.

The Chart also has several subproperties which determine the Title, Foot, Axis and Legend.

These are Chart subcomponents, which already have several properties. They are created and destroyed by the chart itself and they derive from TPersistent so you can access them at Delphi's Object Inspector.

3 different chart components

TCustomChart

The TCustomChart component is the main charting component. Deriving from it we have TChart, which simply publishes all TCustomChart properties. TChart adds nothing to TCustomChart.

This is to allow you to derive from TCustomChart and publish the properties you need, while keeping the rest public and unpublished.

Internally, TCustomChart derives from TCustomAxisPanel, which derives from TCustomTeePanel, which finally derives from Delphi's TCustomPanel.

These classes exist for two reasons: first, to avoid the 64 kilobyte maximum unit size in Delphi 16bit, second, to separate TCustomChart functionality in smaller units so preparing for future TeeChart extensions, which may not require any Axis or charting specifically features.

TCustomTeePanel is declared at TEEPROCS.PAS unit.

TCustomAxisPanel is declared at TEENGINE.PAS unit.
TCustomChart and TChart components are declared at CHART.PAS unit.

TDBChart

TDBChart derives from TCustomChart, publishes all properties and overrides a few methods so the Series can be connected to any TDataSet component. It also maintains a “pool” of TDataSource components to know when the datasets are opened or closed so dataset records must be processed and their values should be added to one or more Series.

You can mix Series connected to databases and Series connected to other sources on the same TDBChart component.

TDBChart also provides some methods used by the Chart editor dialog to fill the database field names on Series DataSource editor tabs.

TDBChart source code is totally contained in DBCHART.PAS unit.

TQRChart

TQRChart derives from QuickReport's TQRPrintable component and interfaces to a small special TQRDBChart component. Both are declared and implemented at QRTEE.PAS unit.

See Chapter 7 (Integrating TeeChart) for a description of TQRChart as an example of integration with other VCL publishers.

A fourth chart component developed by Borland Intl.

In Delphi 3 Client-Server version, Borland implemented a fourth Chart component which integrates with Delphi 3 Decision Cube package.

This component (TDecisionGraph), interacts with a DecisionSource and charts the current cube “slice” and the “slice” totals. With Borland TPivotTool toolbar you can “rotate” the cube slice so new data will appear on the chart. The cube is not limited to 3 dimensions. You can have small cubes inside other more big cubes, and so on...

The implementation of TDecisionGraph is not delivered in Delphi 3, but you can guess it's “copying” a DBGrid cell values to chart Series, creating and destroying the Series again as the data change, among other things.

TDecisionGraph derives from TCustomDecisionGraph, which derives from TCustomChart component.

b) The Series components

Series are invisible components which encapsulate a specific charting type. Series form another hierarchy of components starting from TChartSeries, which derives from Delphi's TComponent.

TChartSeries is the common ancestor for all Series types. You may think of TChartSeries as an abstract component that has four main functions:

- Communicating with the Chart, Chart Axis and Chart Legend.
- Storing point values
- Connecting to a DataSource of point values (another Series or a database dataset)

- Optionally applying a Function to “filter” the source values.

TChartSeries itself does not know how to draw it's points. It does not use any drawing function.

The derived Series components are the ones responsible to draw something on the Chart Canvas. The Series hierarchy tries to reuse functionality, to reduce source code size and to exploit Delphi's Object Oriented capabilities as much as possible.

So, we have for example the TPieSeries and TPolarSeries. Both derive from a common TCircledSeries ancestor which “knows” that points are expressed in angles and vector units instead of “X” and “Y” units.

Specialised Series, for example, add more list of values than the standard X and Y lists. The TSurfaceSeries has a “Z” list of values to complement the original “X” and “Y” lists.

There's no limits on what a Series component can publish as properties. You can for example create a new TBarSeries component and add new properties like: “StockPriceURL”, making your Series to connect to the net to retrieve stock prices itself .

You can then use this Series in applications where this is required, without interfering with the original TBarSeries, or you can distribute this Series to other TeeChart (or Delphi 3) users and let them exploit your work.

ParentChart

Every Series has one and only one ParentChart. That is, you can have Series attached to one Chart and you can then move them to another Chart by just changing the Series ParentChart property:

```
Series1.ParentChart:=DBChart3;
```

The Chart maintains an internal list of their “owned” Series. This list can be accessed throughout the Chart SeriesList property.

Streaming

In TeeChart 1.03, the Form was the owner and parent of the Series. That means the Series were streamed to the DFM file by the Form component.

In TeeChart 3.0 this has changed completely to accomplish more functionality inside Delphi IDE and to overcome some limitations of Delphi's VCL model.

Now the Series parent is the Chart component, and the Chart is the responsible to load and save the Series to the DFM file.

This change has many good advantages. Now when copying, cutting and pasting Chart components the Chart Series are copied and pasted as well. You can now inherit Forms containing Chart components so Form inheritance will inherit Series as well.

Creating new Series at runtime

By default Series are created at design time using the Chart Gallery. You can also create and append new Series to Charts in runtime in one line of code:

```
Chart1.AddSeries( TLineSeries.Create( Self ) );
```

Where “Self” refers to any valid Delphi component to be the owner of the Series, usually the Form.

The Chart component automatically removes all Series when it's destroyed. You can prevent the Chart from removing a Series by “disconnecting” that Series before destroying the Chart:

```
Series5.ParentChart := nil ;
Chart1.Free;
```

Assigning Series

All Series should override the Assign method so you can “copy” one Series to another. This allows copying / pasting Chart components at designtime as well as changing one Series component from one class to another, preserving all common properties and point values.

The Assign method is also used when changing a Series type (for example from Bar to Pie).

You can programatically “clone” any Series using several methods documented below under “Runtime Editor Interface” chapter.

Chapter 4 “Creating Series Components” describes the Series protected parts and virtual methods you can override in your own Series.

c) The Function component

What is it ?

Functions are very simple components designed to make very easy to apply for instance an average to an existing Series.

You can add a new Series to a Chart, set the Series function to be “Average”, and finally connect the Series to another Series where you have the data values.

Functions derive from a common TTreeFunction component class, which derives from Delphi's TComponent.

Functions are designed to be owned by the Form containing the Chart, and they should be attached to one and only one Series component.

How it works

When Series get their point values from “DataSources”, the Function component controls what points are added to the Series, or what calculation must be done to add the points.

For example, the Copy function just “copies” point values from “DataSource” to the Series.

See chapter 5 “Creating Function Components” for an example of creating a custom function.

4. Creating custom Chart components

There are many good reasons for you to create your own derived Chart components. Maybe the most important is to have an intermediary layer between your projects and TeeChart sources.

By creating your own Chart component you are in some way isolating possible future changes in TeeChart so you'll only need to change your custom Chart sources instead of changing all sources of all projects containing charts.

This concept can be applied to all Delphi components, not only to TeeChart, and it's one of the best features of object oriented programming.

Of course this is optional and you can just use TeeChart components directly, but for long term plans it will give you benefits.

Creating a custom Chart is easy. You should only create a unit and type this code:

```
Unit MyChart ;

interface

Uses Chart, DBChart ;

type TMyChart = class ( TChart )
    end;

    TMyDBChart = class ( TDBChart )
    end;

implementation

end.
```

You can create two units instead of only one. One containing TMyChart and another containing TDBChart.

Your Chart components can be extended in many different ways. The following are samples of extensions to the standard Chart component.

Hiding properties

Deriving your Chart components from TChart will make all TChart properties and events to appear at Delphi's Object Inspector and to be saved to the DFM file.

If you want to hide or remove TChart properties and events you should derive your Chart component from TCustomChart or TCustomDBChart instead:

```
type TMyChart = class ( TCustomChart )
    end;

Now you can publish the properties and events you need:

type TMyChart = class ( TCustomChart )
    published
        property Canvas ;
        property LeftAxis ;
        ....
    end;
```

While this is good to simplify things, you should remember all non published properties will never get stored at the DFM file, so if you use the editor dialog at design time to change non published properties you'll see the changes will not be persistent when you run the project.

It can be necessary then to create a new editor dialog to allow changing only the published properties.

Changing default property values

You might want all Chart components to be for example in 2 dimensions and always with a white background colour.

To do so, you can change the default values by re-publishing that properties and by changing the defaults at your own Create constructor:

```
type TMyChart = class ( TChart )
    public
        Constructor Create( AOwner : TComponent ); override ;
    published
        property Color default clWhite ;
        property View3D default False ;
    end;

Constructor TMyChart.Create( AOwner : TComponent );
begin
    inherited Create( AOwner );
    Color:=clWhite ;
    View3D:=False;
end;
```

Installing the new Chart

Following Delphi guidelines on installing components, your unit should have a Register procedure:

```
Unit MyChart ;

interface
...

Procedure Register;

implementation
...

Procedure Register;
begin
    RegisterComponents('Samples', [TMyChart]);
end;
```

Now you can install MyChart.pas unit in Delphi component gallery as usually.

Delphi 3 includes easier ways to create new components. Just click on "File --> New... --> Component" and choose the ancestor class. Clicking "Install" will add your new component to the gallery in just 3 seconds thanks to the new packages technology !

Extending TMyChart

As an example, let's now make TMyChart to automatically create Series. You might want to have a default Chart component with 2 Series, one Bar and one Line, and make this Series to appear with your own default properties.

The Chart constructor is a good place to add the necessary code:

```
Constructor TMyChart.Create( AOwner : TComponent );
begin
    inherited Create( AOwner );
    Color:=clWhite ;
    View3D:=False;

    if not (csLoading in Owner.ComponentState) then
    begin
        AddSeries( TBarSeries.Create( Owner ) );
        Series[0].Name:='Sales' ;
        Series[0].SeriesColor:=clBlue ;

        AddSeries( TLineSeries.Create( Owner ) );
        Series[1].Name:='Expenses' ;
        Series[1].SeriesColor:=clRed ;
    end;
end;
```

The “csLoading” check is very important.

The first time you drop a TMyChart in a Form, csLoading is not present in Owner.ComponentState, so the two Series are created and added to the Chart. When you press F9 to run the project, Delphi first stores the Chart and the two Series onto the DFM file, and then when the project starts the DFM is loaded again. When the DFM is loaded Delphi creates again the stored Series. This is way our TMyChart component should not create the Series when the owner is being loaded, to prevent creating the two Series twice.

Protected parts

Protected methods and variables are not documented in TeeChart manual nor in TeeChart help file.

The following sections explain each protected method and variable:

Let's follow the Chart class hierarchy starting at TCustomTeePanel class, located in TeeProcs.pas unit.

TCustomTeePanel protected

Avoiding Flicker

TCustomTeePanel owns an internal bitmap object. This object is created and destroyed by TCustomTeePanel itself.

The purpose of this internal bitmap is to avoid screen flickering when redrawing Chart contents.

All drawing operations are performed against the internal bitmap, and when finished, the bitmap content is transferred to the screen in a single operation.

TCustomTeePanel re-publishes the Panel Canvas property to point to the internal bitmap Canvas instead of pointing to the screen device.

When the Chart panel needs to be redrawn, two situations may arise:

- The internal bitmap is already filled with Chart drawing.
- The internal bitmap is empty or it contains an old Chart.

If the internal bitmap is correct and current, TCustomTeePanel just resends the bitmap to the screen. This is just a single operation so no Charting drawing function gets called. This happens whenever you change from one application to another or when other Form passes over the Chart rectangle.

If the internal bitmap does not reflect the current Chart state, TCustomTeePanel forces a real redraw of Chart and Series to the bitmap and then the bitmap is transferred to the screen.

This happens whenever Chart or Series properties change, or when you zoom or scroll at runtime.

There are several variables to control all this behaviour:

property BufferedDisplay:Boolean default True

BufferedDisplay is used to determine if the internal bitmap is used or not.

This property is public, so you can't access it through the Object Inspector.

When False, the internal bitmap is not used at all, so all drawing functions are directed to the screen canvas. This produces flickering, but it can result in faster drawing speed if you develop in 16bit Windows or with cheap video cards.

Property Canvas :TCanvas;

The Canvas property returns the internal bitmap Canvas if BufferedDisplay is True, or the original Panel Canvas if BufferedDisplay is False.

Property DelphiCanvas:TCanvas;

The DelphiCanvas variable is always the original Panel Canvas, no matter if BufferedDisplay is True or False.

InternalDirty:Boolean;

Whenever you invalidate the Chart panel (by calling Chart1.Repaint for example), TCustomTeePanel sets InternalDirty property to True. Then, when Delphi sends the paint message to the Chart, this Boolean variable is used to determine if the internal bitmap contents is valid or not. If InternalDirty is True, the Chart redraws again everything to the bitmap and the bitmap is transferred to the screen. If InternalDirty is False, no drawing is necessary so it just transfers the bitmap to the screen.

InternalBitmap:TBitmap;

This variable points to the bitmap object.

Procedure TransferBitmap(ALeft,ATop:Integer; ACanvas:TCanvas);

This method sends the bitmap contents to the ACanvas parameter at the ALeft and ATop origin co-ordinates.

Function ReDrawChartBitmap:Boolean;

This function checks if InternalDirty is False, and if so, calls TransferBitmap to draw the bitmap to the screen. It returns True in this case, and False if the bitmap is not valid.

procedure Invalidate; override;

Delphi's Invalidate method is overridden by TCustomTeePanel to set InternalDirty to True, thus forcing the Chart to redraw when Delphi calls the Chart Paint method.

procedure WMERaseBkgnd(var Message: TWmEraseBkgnd); message WM_ERASEBKGND;

If you use Windows 95 + Plus !, or Windows NT 4, with “Drag Full Window” option, Windows continuously send Paint messages to the Chart when you resize or move the Window, at every mouse movement.

Prior to the Paint message, Windows sends an WM_EraseBkgnd message to make the control to paint it's background.

When “Drag Full Window” is active, filling the background at every repaint can produce an extra annoying flickering. TeeChart has a global Boolean variable to control if the background is filled or not at every repaint.

This variable is called TeeEraseBack, and it's True by default at design time and False at runtime.

You can set TeeEraseBack to True when your project starts to eliminate flickering when resizing when “Drag Full Window” is on:

```
TeeEraseBack:=False ;
```

Zoom and Scroll

Zooming and Scrolling a Chart are basically the same thing: changing axis scales.

When you zoom or scroll, either using the mouse or programmatically, TeeChart changes the axis minimum and maximum values and forces the Chart to repaint again completely.

Scrolling increments or decrements both the Minimum and Maximum axis values by the same offsets.

Zooming increments the Minimum and decrements the Maximum, or vice-versa, to “stretch”

or “expand” the axis scales.

Whenever the Chart redraws, all Series points use the axis scales to calculate where to paint every point or bar or area.

Please refer to TeeChart manual to zoom and scroll by code using the public methods and properties.

There are three protected methods and one internal variable related to zoom:

IZoom and IPanning : TZoomPanningRecord

The IZoom and IPanning public variables store the screen co-ordinates bounding the zoom rectangle.

These co-ordinates are updated when dragging the mouse, and then are used to recalculate the new axis scales. TZoomPanningRecord definition is at TeeProcs.pas unit.

Procedure DrawZoomRectangle;

This method draws a rectangle over a Chart during mouse drag. The IZoom co-ordinates are used to draw the rectangle.

Procedure CalcZoomPoints; virtual;

This method takes IZoom values to calculate the new axis scales, and then calls DoZoom method to actually change the axis and redraw the Chart.

Procedure DoZoom(Var ALeft,ARight,ATop,ABottom:BiDouble); virtual;

The DoZoom method changes axis Minimum and Maximum values, and forces chart redrawing.

The Draw sequence

When Delphi (or Windows) calls the Chart *Paint* method, the following happens:

If the internal bitmap is not “dirty”, that means it can be used directly to transfer its contents to the screen. If so, nothing more happens. The Chart is drawn.

If the bitmap is not current or not used, the *TChart.Draw* method gets called to draw everything again and then to transfer the drawing to the screen.

The *Draw* method (located at *Teengine.pas* unit), adjusts the bounding rectangle coordinates using the Chart margin values, and then calls the *InternalDraw* method, which is the final responsible to draw everything.

InternalDraw starts the drawing sequence by filling the Chart background. Chart background is painted by *PanelPaint* method.

PanelPaint (located at *Chart.pas* unit), does the following:

If the Chart *Gradient* is used, it calls *GradientFill*. If *Gradient* is not used it fills the Chart background with the *Panel Color*.

Note: the *Panel Color* is not used when printing to avoid printing a grey chart instead of the more suitable white background.

The *PrintTeePanel* global variable can be set to True to use the *Panel Color* also when printing.

After checking the *Gradient* and *Panel Color*, *PanelPaint* checks if it's necessary to draw a background image (the *Chart BackImage* property).

PanelPaint finally draws the Panel edges (*BevelInner* and *BevelOuter* properties).

Titles and Legend

The next step after painting the Chart background is to draw the Chart *Title*, *Foot* and *Legend* subcomponents.

The *DrawTitlesAndLegend* method (located at *chart.pas* unit) is the responsible of this.

There's a special case with drawing *Titles* and *Legend*.

If the *Legend* is located at Right or Left positions, the *Legend* should be drawn before the *Titles*. This is because *Title* and *Foot* should be centred on the Chart without including the *Legend* dimensions.

The *Legend* is drawn first so it's dimensions can be used to recalculate the remaining rectangle used to centre the *Title* and *Foot*.

But, if *Legend* position is Top or Bottom, we should draw first the *Title* and *Foot* to calculate the remaining rectangle for the *Legend*.

Title and *Foot* are the same object class, so they are drawn using the same method: *TCustomChart.DrawTitle*

DrawTitle starts by calculating the *Title* rectangle dimensions. The rectangle size is determined by the *Title.Font* used and the *Title.Pen* used to draw a frame around the *Title*.

The frame, if visible, is drawn, and the *Title* text is then drawn one line per line.

After drawing the Title and Foot, the remaining Chart rectangle is recalculated.

The Legend has it's own method to draw: *TChartLegend.Draw* (located at chart.pas unit).

The Legend.Draw method does a similar thing that DrawTitle, although more complex as Legend can be positioned at Left, Right, Top or Bottom, and as Legend displays a coloured rectangle representing each Series properties.

The Legend has two modes of operation:

- One Legend item for each point in the first Series on the Chart
- One Legend item for each Series on the Chart

The Legend.LegendStyle property controls these two modes.

Legend.Draw first calculates how many Legend items should be displayed taking care of the LegendStyle mode.

Then it calculates if it's possible to draw all these items with the current Chart size and Legend Font size. It calculates the maximum possible number of Legend items with the available space.

Then it draw the Legend background, the Legend frame, and starts drawing each Legend item, taking care of Legend Inverted property to start from top or bottom in the Legend items list.

To draw every item, the Legend first asks the Series to draw the coloured rectangle. This is done at TChartSeries DrawLegend virtual method. Every Series class is the responsible of drawing that small rectangle in the appropriate way, according to its formatting properties.

After drawing all items, the Legend draws the Legend Shadow around the Legend rectangle and then recalculates the remaining chart space.

Series Z Order

When Chart View3D property is True, Series are drawn one behind the other to achieve the 3D effect.

Before drawing the Chart Axis and the Series, the Chart component calculates the MaxZOrder value.

Every Series has a Z Order, an integer representing the Series position (the order) on which the Series are drawn.

After drawing the Panel, Titles and Legend, the drawing sequence calculates the corresponding Z Order value for each Series.

This calculation is a little bit complex. For instance, Bar Series in Stacked or Stacked 100% mode share the same Z Order. This means a virtual method in all Series components is the responsible to calculate the Z order and take care of the specific situations regarding Z order, like stacked Bar Series.

All Series have a CalcZOrder virtual method, which by default increments the Chart MaxZOrder property.

The Chart uses MaxZOrder to calculate the 3D chart dimensions. So, 3D dimensions are directly proportional to the number of Series in the chart.

When Chart View3D is False (2D charting), all Series have the same ZOrder: zero.

Also, the Chart has a global Boolean variable that determines if Series ZOrder is used or not.

The teedemo project has an example of this at LastValu.pas unit.

The Chart Rectangle

The Chart rectangle defines the XY space in pixels where Series points are displayed. This rectangle has been actually calculated after drawing the Chart Titles and Legend, and now it should be recalculated again to reflect the extra space axis use and the extra space Series need to, for example, Series Marks.

Another very important issue during the drawing sequence is to calculate where the axis should be positioned, and most important, how much margins should be applied to the axis scales.

How Axis affect Chart rectangle

For example, when a Bar Series has its Marks visible (the default), the axis should increase their Maximum scale value so there's enough space to draw Series Marks. If not, Series Marks will be drawn outside the Chart rectangle, which is a very bad effect !

TCustomChart has the CalcAxisRect virtual method to calculate the correct axis rectangle.

The logic of CalcAxisRect is as follows:

First, all four axis are forced to calculate their Minimum and Maximum scale values. This is necessary at this point as Axis Minimum and Maximum can be Automatic, meaning the axis will "guess" the right Minimum and Maximum based on how many Series are Active and what are the minimum and maximum Series values both for the Series horizontal axis and the Series vertical axis.

The *TChartAxis.AdjustMaxMin* method is called for each axis (LeftAxis, TopAxis, RightAxis and BottomAxis).

TChartAxis.AdjustMaxMin calls *TChartAxis.CalcMinMax* and sets the axis internal variables for Minimum and Maximum values.

TChartAxis.CalcMinMax checks if Minimum and Maximum are Automatic, and if so, axis call the Chart component to traverse all Series and return the Minimum and Maximum for the axis.

Once the axis Minimum and Maximum values are obtained, the Chart calls again all the axis to change the available rectangle space.

Axis make this rectangle space smaller to accommodate space for the axis Title, for the axis Labels, for the axis Ticks and for the axis line itself.

How much space depends on what Font size is used for axis Labels, if the axis Labels are visible or not, what length the axis Ticks have, which pen size is used for the axis Ticks and so on.

TChartAxis.CalcRect (located at teengine.pas unit) is the method that calculates the axis space.

How Series affect Chart rectangle

Now the axis have calculated their space, Series are give a chance to obtain more space for the axis.

The *TCustomChart.CalcSeriesRect* method (located at chart.pas unit) is called at this point to re-adjust axis Minimum and Maximum.

For each Active Series associated to each of the four axis, the Series is requested to calculated the “margins” space. This “margins” are used then to re-adjust axis Minimum and Maximum.

To calculate Series “margins”, the Series has two virtual methods:
TChartSeries.CalcHorizMargins and TChartSeries.CalcVertMargins

Every Series that need extra margin space override the above virtual methods.

For example, Bar Series override these methods so there's an extra space from the side of the Chart to the first Bar and from the last Bar to the other side of the Chart.

Some other Series perform different margin calculations based on their needs.

Painting the Walls

Now we have, finally ! calculated the Chart rectangle, the Chart draws the axis walls. *TCustomAxisPanel.BackWallsPaint* method, located at teengine.pas unit, sets the Canvas attributes and calls *TCustomChart.DrawWalls* virtual method, which finally draws the left Wall and Bottom wall.

Drawing the Axis

After drawing the Chart walls the axis are requested to draw.
TChartAxis.Draw method gets called for all visible axis.

TChartAxis.Draw is quite complex, as it takes care of labels, grid lines, ticks, etc.

The most complicated thing in axis drawing are how to draw axis labels.
Axis labels have two “modes”, which an specific logic for each “mode”.

One axis labelling mode is to draw one label at each axis Increment position.
By default Increment is zero, so the axis firsts needs to calculate the most suitable increment value. This value depends if the axis draw DateTime labels or non-DateTime labels, and it depends on axis labels font size to avoid axis label overlapping.
To complicate things more, axis labels can be drawn with rotated fonts. This should be take in consideration when calculating the axis Increment.

For DateTime axis, there's also extra functionality. The Increment property can be set to ExactDateTime, so OneMonth increment is not 30 days, it's an exact month increment.

The other label mode is to draw one label for each point in the Series, no matter where each point is positioned along the axis. In this case the Increment property is unused.
This mode is simpler than the other mode, but as axis labels have different styles, different methods are used to retrieve the axis labels (the Series point labels, the Series marks texts, etc).

A big effort has been done to make axis work with all kind of scale ranges, but there are some cases where axis Increment should be set manually (with very small or with very

big axis scales for example), as the generic “automatic” calculation can not cover everything.

Drawing the Series

Let's get the real work done. Series points are our final goal.

Series points are first pre-processed to determine which should be the first and last point in the Series point list that gets inside axis scales.

The *TChartSeries.CalcFirstLastVisibleIndex* virtual method is the responsible to calculate for each Series class the first and last visible point. Series do so using the current axis minimum and maximum.

At this point the Chart auto-paging feature takes place.

When Chart *MaxPointsPerPage* property is used, Series use the current *Chart.Page* property to calculate which should be the first and the last point in the Series to display. Series points are drawn using the *TChartSeries.DrawAllValues* virtual method.

This method uses the pre-calculated first and last visible point indexes, and for all points calls to the Series *DrawValue* virtual method, which should be the final responsible of drawing an individual point.

Every Series overrides *DrawValue* to display one point.

An special case happens with Series sharing the same Z Order (as stacked Bars for example).

In this case the logic is inverted. Instead of drawing all points for each Series, one point at a time is drawn for all Series.

Instead of calling *Series.DrawValues* method, the Chart calls *Series.DrawValue* method for each point.

After Series points are displayed, the Chart calls all *Series.DrawMarks* method, which in turn calls the virtual *Series.DrawMark* method for each point mark, calculating the rectangle co-ordinates for each mark.

Both Series points and Series marks are affected by clipping. Clipping is done calling Windows API *SelectClipRgn* function.

Several lines of code take care of clipping, as it involves 2 and 3 dimensions.

The End of the Draw sequence

Now we only need two more things to finish all drawing.

If, for example, we have a real-time chart with automatic live scrolling, showing new points, and at that time we would like to zoom a chart region, the Chart shows the drag rectangle to identify the region to zoom.

When the Chart redraw, it needs to remember we are in the process of zooming to draw the zoom rectangle again.

This is done at this point calling the *DrawZoomRectangle* method.

And, to finish, we should call the Chart *OnAfterDraw* event so you can custom draw new things onto Chart canvas.

User Mouse clicks.

Another big part of TChart component is to handle user mouse clicks.

All mouse click handling is done at Chart.pas unit, and it basically starts with the following three overridden methods:

```
procedure MouseDown(Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
override;
procedure MouseMove(Shift: TShiftState; X, Y: Integer); override;
procedure MouseUp(Button: TMouseButton; Shift: TShiftState; X, Y: Integer); override;
```

Using these methods the Chart provides more high level events like Series OnClick, OnDbClick and several Chart related events like OnClickSeries, OnClickAxis, etc.

These methods are also used to control zoom and scroll.

MouseDown

Whenever you press a mouse button inside a Chart component, the MouseDown method is called by Delphi.

TChart does the following at MouseDown:

- Calculate which Chart or Series is the mouse pointer at.
- For each different part perform the adequate thing (see below)
- If no specific part is clicked, start zooming or scrolling depending which mouse button is pressed.

The Clicked part

TChart has a public method that calculates which Chart section is at a specific screen pixel position.

This method is:

```
Procedure CalcClickedPart( Pos:TPoint; Var Part:TChartClickedPart );
```

The TChartClickedPart parameter has this declaration:

```
TChartClickedPart=Record
    Part          : TChartClickedPartStyle;
    PointIndex    : Longint;
    ASeries       : TChartSeries;
    AAxis         : TChartAxis;
end;
```

Depending on “Part”, this record is filled with the appropriate Series, Series point index or Axis.

“Part” can be one of the following values:

```
TChartClickedPartStyle=( cpNone,
                        cpLegend,
                        cpAxis,
                        cpSeries,
                        cpTitle,
                        cpFoot,
                        cpChartRect );
```

The logic on calculating which part is clicked is:

cpLegend

Legend is Clicked if mouse is inside Legend.RectLegend rectangle.

If True, the OnClickLegend event is triggered.

cpAxis

An Axis is Clicked if the Axis.Clicked method returns True.
If True, the OnClickAxis event is triggered.

cpSeries

A Series is Clicked if mouse is inside any Series point.
The Series.Clicked function returns which Series point is the mouse cursor over.
Every Series override Clicked virtual method in their appropriate way.
If True, the Series.DoSeriesClick virtual method gets called, and then the Chart.OnClickSeries event is triggered.

cpTitle

The Title is Clicked if the Title.Clicked method returns True.
No event exists to notify the Title has been clicked. You can call CalcClickedPart method or the Title.Clicked function programmatically if necessary.

cpFoot

The Foot is Clicked if the Foot.Clicked method returns True.
No event exists to notify the Foot has been clicked. You can call CalcClickedPart method or the Foot.Clicked function programmatically if necessary.

cpChartRect

The Chart axis rectangle is Clicked if the mouse is inside Chart.ChartRect.
If True, the OnClickBackground event gets triggered.

No clicked part

After calling the appropriate event, the Chart starts zooming or scrolling.
You can stop the Chart to start zooming or scrolling by setting a special Boolean variable:

CancelMouse

When CancelMouse is True, the Chart does not start zoom or scroll after your code has handled the click event.
When CancelMouse is False, the Chart will start zooming or scrolling after your code has handled the click event.

For zoom and scroll to start, the AllowZoom and AllowPanning properties should be positive.

Mouse buttons

To control which mouse button should be pressed to zoom or scroll, the Chart.pas unit contains four global variables:

```
TeeZoomMouseButton    : TMouseButton; { button used to zoom }
TeeScrollMouseButton  : TMouseButton; { button used to scroll }
TeeZoomKeyShift       : TShiftState;   { keys that should be pressed to start zoom }
TeeScrollKeyShift     : TShiftState;   { keys that should be pressed to start scroll }
```

You can set the above variables to your desired values.
Be careful not to assign the same mouse button both to zoom and scroll !

Zoom is performed when pressing the Left mouse button (Right on left-handed Windows).

Scroll is performed when pressing the Right mouse button (Left on left-handed Windows).

Printing Charts

Printing is another big part inside TeeChart.

For “what-you-see-is-what-you-get” printing, Windows provides a special Canvas mode, called “Anisotropic”.

This mode is the same when creating a metafile image of Chart contents.

Metafiles are simply a sequence of drawing instructions, as opposed to bitmaps which are an array of pixels.

Metafiles are more powerful as they can be resized, they are a vectorial graphic format.

Metafiles are generally smaller than bitmaps for an equal Chart size.

In the “old” Window times, metafiles were used to transfer drawing instructions from one application to another via the Windows clipboard.

During these years metafiles have been designed as defacto standard for printer drivers, and with Windows NT and Windows 95 they now play a bigger role with an improved “Enhanced Metafile” mode.

Windows 3.1 has some known bugs in GDI when handling metafiles. TeeChart tries to suit standard and enhanced metafiles both in Delphi 16bit and 32bit, using Delphi TMetafile and TMetafileCanvas objects when possible.

Please refer to TeeChart manual for more information about metafiles and printing issues.

We'll describe here the TChart organisation for printing and creating metafiles.
All printing and metafile methods are located in chart.pas unit.

Printing methods

Procedure Print;

The Print procedure simply outputs a Chart to the current selected printer using the default printing margins and default printer paper orientation.

The Printer page is ejected after printing.

Procedure PrintLandscape;

Procedure PrintPortrait;

Same as Print method but changing the paper orientation if necessary.

Procedure PrintOrientation(AOrientation:TPrinterOrientation);

Same as Print method but changing the paper orientation to AOrientaton parameter.

Procedure PrintRect(Const R:TRect);

Same as Print method but outputting the Chart to the desired R rectangle expressed in Printer Canvas units.

Function ChartPrintRect:TRect;

This function returns the rectangle co-ordinates after applying the Printing margins.
See below for an explanation of printing margins.

Procedure PrintPartial(Const PrinterRect:TRect);

Procedure PrintPartialCanvas(PrintCanvas:TCanvas; Const PrinterRect:TRect);

These two methods output a Chart to the passed Canvas of the passed rectangle size, but they will NOT eject the printer page after printing.

Printing variables and properties

PrintTeePanel : Boolean; (global variable)

Set it to True before printing to get the Chart background on paper. By default it's False.

PrintMargins:TRect;

This Chart property is used to calculate the paper margins. Margins are expressed in percent of paper width and height. You can set the Left,Right,Top and Bottom margin using this property. Default margins are 15%.

PrintResolution:Integer;

This property plays a big role when printing.

It defines the ratio (or proportion), between the actual screen resolution and the printed resolution. By default it's zero, meaning screen pixels will be scaled to printer pixels to obtain as much "wysiwyg" as possible.

Setting PrintResolution to -100 (or any negative number), will make all Font sizes and printed lines to appear thinner and smaller. This can produce better results on old printers.

Clipboard

Clipboard exporting is supported by the following methods:

Procedure CopyToClipboardBitmap:

The implementation of CopyToClipboardBitmap is:

```
Procedure TCustomChart.CopyToClipboardBitmap;
var tmpBitmap:TBitmap;
    R:TRect;
begin
    tmpBitmap:=TBitmap.Create;
    with tmpBitmap do
    try
        R:=GetRectangle;
        Width := R.Right-R.Left;
        Height:= R.Bottom-R.Top;
        Self.Draw(Canvas,R);
        Clipboard.Assign(tmpBitmap);
    finally
        tmpBitmap.Free;
    end;
end;
```

Procedure CopyToClipboardMetafile(Enhanced:Boolean);

And the implementation of CopyToClipboardMetafile is:

```

Procedure TCustomChart.CopyToClipboardMetafile(Enhanced:Boolean);
Var Meta:TMetaFile;
begin
  Meta:=TeeCreateMetafile(Enhanced,GetRectangle);
  try
    ClipBoard.Assign(Meta);
  finally
    Meta.Free;
  end;
end;

```

The “Enhanced” parameter determines (in 32bit only), if the metafile is standard (WMF with Aldus header) or extended (EMF format).

Exporting

As well as copying to the clipboard, the TChart component includes these methods to export Chart images to files:

Procedure SaveToBitmapFile(Const FileName:String);
Creates a new file and stores a bitmap image of a Chart.

Procedure SaveToMetafile(Const FileName:String);
Procedure SaveToMetafileEnh(Const FileName:String);
Create a new file and store a metafile image of a Chart. (needs WMF or EMF file extensions).

Procedure SaveToMetafileRect(Enhanced:Boolean; Const FileName:String; Const Rect:TRect);
Creates a new file of metafile format of a Chart with Rect size.

Function TeeCreateMetafile(Enhanced:Boolean; Const Rect:TRect):TMetafile;
This function is used by all metafile methods.
It returns a Delphi's TMetafile object. You should take care of freeing this object:

```

Var tmp:TMetafile;
tmp:=Chart1.TeeCreateMetafile( True, Rect );
try
  .... do here whatever you want with "tmp"....
finally
  tmp.Free;
end;

```


5. **Creating custom Function components**

Type declaration

Creating a new Function component is simply creating a new component derived from TTeefunction (it also can be derived from an existing function):

```
type
  TMyFunction = class( TTeefunction )
  end;
```

TTeefunction source code is located at Teengine.pas unit.

Standard functions like add, subtract, etc, source code is located at TeeFunci.pas unit.

Example:

Create a new Project.

Drop a TChart.

Add a Bar Series (Series1)

Add a Line Series (Series2)

Place a TButton and type this code on Button1Click:

```
Series2.SetFunction( TAddTeeFunction.Create(Self) );
Series2.DataSource:=Series1;
```

The above 2 lines are not necessary (it can be done with Chart Editor dialog), and are shown here as how to do it programmatically.

Now Series2 is ready to calculate the sum of Series1 values. We just need to add values to Series1 to see how Series2 shows the total sum:

```
With Series1 do
begin
  Clear;
  Add( 10, 'Sample 1', clTeeColor);
  Add( 20, 'Sample 2', clTeeColor);
  Add( 5, 'Sample 3', clTeeColor);
  RefreshSeries;
end;
```

The “RefreshSeries” method forces the recalculation after finished adding points.

Now you should see Series2 (the Line) in the vertical position of Series1 sum of values:

10 + 20 + 5 = 35

“RefreshSeries” is not necessary when connecting Series to databases. In that case it's called automatically.

Implementing TMyFunction

Let's decide we need TMyFunction to return the “sum of squares”.

Change the previous code to use the TMyFunction instead of TAddTeeFunction:

```
Series2.SetFunction( TMyFunction.Create(Self) );
```

There are 2 important virtual methods in TTeefunction that can be overridden to create a new Function type.

- 1) Function TTreeFunction.Calculate(SourceSeries:TChartSeries; First,Last:Longint):Double;
- 2) Function TTreeFunction.CalculateMany(SourceSeriesList:TList; ValueIndex:Longint):Double;

Then “register” the new function so it can be used at design-time in the editor and gallery.
See below “Registering Functions”.

Calculate

Let’s override TMyFunction “Calculate” method to do a very simple thing.

The “Calculate” method looks like:

```
Function TMyFunction.Calculate(SourceSeries:TChartSeries;
First,Last:Longint):Double;
Var StartPoint, EndPoint, t : Longint;
begin
  StartPoint:=0;
  EndPoint:=SourceSeries.Count-1;
  if First <> -1 then StartPoint:=First;
  if Last <> -1 then EndPoint:=Last;

  result:=0;
  for t:=StartPoint to EndPoint do
  begin
    result:= result + Sqr( SourceSeries.MandatoryValueList [ t ] );
  end;
end;
```

In the above example the “Calculate” method gets called by Series2, passing Series1 as “SourceSeries” parameter, and “First” and “Last” both equalling “-1” meaning all points should be considered for calculation.

The StartPoint and EndPoint variables are used to “loop” all SourceSeries points to calculate the sum of squares.

The “MandatoryValueList” property is used instead of “YValues” property just to make this function to work with Series types like HorizBarSeries where “XValues” holds the point values and not “YValues”.

If you run the previous project with TMyFunction, you should see Series2 (the Line) showing the sum of square of Series1 point values:

$10*10 + 20*20 + 5*5 = 525$

The “Calculate” method is used when the Series has **only one** Series as DataSource. When Series have more than one Series as datasources, the “CalculateMany” method is called.

CalculateMany

You can also override TMyFunction “CalculateMany” to allow TMyFunction to work with Series which have more than one Series as datasources.

Programmatically, Series can be added as datasources of other Series.

In the previous example, just add a new Series3 (Bar) and substitute the "Series2.DataSource:=Series1" with this code:

```
Series2.DataSources.Clear;
Series2.AddDataSource(Series1);
Series2.AddDataSource(Series3);
```

Add sample points to Series3 with this code:

```
With Series3 do
begin
  Clear;
  Add( 40, 'Sample 4', clTeeColor);
  Add( 20, 'Sample 5', clTeeColor);
  Add( 10, 'Sample 6', clTeeColor);
  RefreshSeries;
end;
```

If you run this project now, you'll see TMyFunction has calculated nothing. This is because there are two datasources now, and we haven't overridden "CalculateMany" to support more than one datasource.

So let's override CalculateMany and run the project again:

```
Function TMyFunction.CalculateMany(SourceSeriesList:TList;
ValueIndex:Longint):Double;
var t : Longint;
begin
  result:=0 ;
  for t:= 0 to SourceSeriesList.Count -1 do
  begin
    result := result + sqr( SourceSeriesList[ t ].MandatoryValueList
[ ValueIndex ] );
  end;
end;
```

"CalculateMany" will get called once for each point in the source Series, starting from zero and ending with the minimum point count of all datasources.

We had the following values:

```
Series1:  10 20 5
Series3:  40 20 10
```

The results are:

```
first point:  10*10 + 40*40 = 1700
second:      20*20 + 20*20 = 800
third:       5*5 + 10*10 = 125
```

It is very important to understand the difference between Calculate and CalculateMany. "Calculate" is called when there only one datasource and it's called only once. "CalculateMany" is called several times (one for each point) when there are more than one Series as datasources.

Registering your new Functions:

Like Series types, Functions can also be registered in TeeChart gallery.

The way to do it is very similar as registering Series.

You should place this code at the bottom of you unit and in the “initialization” part:

```
{ Functions Registration }
Procedure MyFunctionExitProc; far;
begin
    UnRegisterTeeFunctions([ TMyFunction ]);
end;

initialization
    RegisterTeeFunction( TMyFunction, 'SumSqr', 'David', 1 );
{IFDEF WIN32}
    AddExitProc(MyFunctionExitProc);
{ENDIF}
{IFDEF WIN32}
finalization
    MyFunctionExitProc;
{ENDIF}
end.
```

The “RegisterTeeFunction” takes four parameters:

- The function class to be registered
- The function name to appear at gallery
- The gallery tab where the function appears
- The number of random series for the gallery (usually 1)

Just adding your Function units to Delphi's component library will make them magically appear at TeeChart Gallery !

6. TeeChart Runtime Editor interface

The Chart Editor Dialog is a big and complex dialog. It manages Chart and Series properties and allows you to connect Series to databases and Series to Functions with just mouse clicks.

You might want to include the editor in your projects as a fast way for end users to control Chart properties.

At design-time under Delphi IDE, the Chart and Series components access the Editor dialog through a set of methods located in several units.

We'll see here which methods are available, what they do and how to call them from our projects.

Editing a Chart

To start the Editor dialog with your TChart component, simply type this:

```
Uses EditChar ;

EditChart( Self, Chart1 );
```

Where "Self" will be the owner of the Editor dialog. You can also pass "nil" instead of "Self".

The Editor Dialog will show the same dialog you see when double clicking a Chart component at design-time under Delphi.

Editing a DBChart

If you want to edit a TDBChart component, you need then to do this:

```
Uses DBEditCh ;

EditDBChart( Self, DBChart1 );
```

The reason of EditChart and EditDBChart is the same reason there are a TChart and TDBChart components.

Using EditChart will not link the BDE units in your project, while using EditDBChart will do.

Cloning a Series

You can create a copy of any Series by calling this function:

```
Uses Chart;
Var tmp:TChartSeries ;
    tmp:=CloneChartSeries( Series1 );
```

A new Series will be created of the same class as Series1. All formatting properties and values from Series1 will be copied to the new Series and finally the new Series is returned.

Changing a Series type

You can change a Series from one type to another calling this function:

```
Uses Chart;
Var tmp :TChartSeries;
    tmp:=Series1;
    ChangeSeriesType( tmp, TBarSeries );
```

The original Series is DESTROYED and the “tmp” parameter is returned with a newly created instance of the type specified (TBarSeries). Series properties, point values and events are copied from the old Series to the new. Note: Only the common properties from one type to another are copied. As this happens at runtime, you should use runtime castings in your code to avoid runtime crashes and general protection errors:

```
if ( tmp is TBarSeries ) then
    ( tmp as TBarSeries).MultiBar:=mbStacked;
```

ChangeAllSeriesType

This method calls ChangeSeriesType for all Series on a Chart:

```
ChangeAllSeriesType( Chart1, TBarSeries );
```

Naming Series

When creating a new Series component at design-time or runtime programmatically, it's a good practice (and sometimes mandatory) to name the Series component. To give a name to a Series you can simply set the name property:

```
Var tmp:TLineSeries ;
    tmp:=TLineSeries.Create( Self );
    tmp.ParentChart:=Chart1;
    tmp.Name:='Peter';
```

However, if you plan to support Form inheritance, and to avoid duplicating component names, you might want to use the following method:

```
{ Returns a valid component name starting with AStartName, for Delphi
1, 2 & 3
}
Function TeeGetUniqueName(AOwner:TComponent; Const
AStartName:String):String;
```

This method takes care of non duplicating Series names, assigning a sequence number like Delphi IDE does:

```
Var tmp:TLineSeries ;
    tmp:=TLineSeries.Create( Self );
    tmp.ParentChart:=Chart1;
    tmp.Name:=TeeGetUniqueName(Self, 'Peter');
```

The returned Name will be “Peter1”, “Peter2”, and so on...

The Chart Gallery

TeeGally.pas unit contains several methods to interface with the Chart Gallery.

You can use these methods in your projects too.

All gallery methods are used by the Chart editor dialog (see IEditCha or IEdit16 units).

```
{ Shows the gallery and asks the user a Series type.
  If user double clicks a chart or presses Ok, a new Series
  is created and returned. The new Series is owned by AOwner
  parameter (usually the Form).
}
Function CreateNewSeriesGallery( AOwner:TComponent;
                                OldSeries:TChartSeries;
                                tmpChart:TCustomChart;
                                AllowSameType,
                                ShowFunctions:Boolean
                                ):TChartSeries;
```

Example:

```
Var tmp:TChartSeries ;
    tmp:=CreateNewSeriesGallery( Self, nil, Chart1, True, True );
```

“OldSeries” parameter can be set to an existing Series component.

If so, the Gallery will check for each Series type in the gallery if they are compatible with “OldSeries”, disabling the non compatible types.

“AllowSameType” can be set to False when using “OldSeries” and not wanting the user to choose the same Series type for the new Series.

“ShowFunctions” simply hides the “Functions” gallery tab when False.

```
{ Shows the gallery and asks the user a Series type. Then
  changes tmpSeries to the new type.
}
procedure ChangeSeriesTypeGallery(AOwner:TComponent; Var
tmpSeries:TChartSeries);
```

Very similar to ChangeSeriesType except the gallery is used to ask the user the new Series type.

```
{ Shows the Gallery Dialog and lets the user choose a Series type.
  Returns True if user pressed OK.
  The "tmpClass" parameter returns the choosen type.
}
Function GetChartGalleryClass( AOwner:TComponent;
                                OldSeries:TChartSeries;
                                ShowFunctions:Boolean;
                                Var tmpClass:TChartSeriesClass;
                                Var Show3D:Boolean;
                                Var
tmpFunctionClass:TTEEFunctionClass
                                ):Boolean;
```

This low level method shows the gallery and returns the user selection at tmpClass, Show3D and tmpFunctionClass parameters. The function returns False if user choose Cancel.

```
{ Returns the name of a Series in it's "gallery" style:
  TLineSeries returns "Line"
  TPieSeries returns "Pie"
  etc.
}
Function GetGallerySeriesName (ASeries:TChartSeries):String;
```

This function returns the title the gallery shows for the ASeries Series class object.

```
{ Shows the gallery and asks the user a Series type. Then
  changes all Series in AChart to the new type.
}
procedure ChangeAllSeriesGallery( AOwner:TComponent;
AChart:TCustomChart );
```

Very similar to ChangeAllSeriesType except the gallery is used to ask the user the new Series type.

7. Integrating TeeChart with your VCL

VCL Developers can consider worth upgrading their components to support TeeChart controls.
This chapter describes the different approaches to integrate TeeChart with your classes.

There are different approaches to do this.

Doing nothing

If your VCL components are visual controls, maybe it's not necessary to do anything new to TChart or TDBChart components.
You can insert any control inside a TChart (because it is a TPanel), and you can insert a TChart inside any control capable of having "child" controls (as a TPanel for example).

If you create custom database components (derived TDataSet components), it's almost sure TDBChart will automatically integrate with them.
You'll need to do a good testing of connecting Series to your DataSets and checking everything works fine both at design-time and run-time.

Deriving

If you have non-visible components which contain or adquire data, one way to create a Chart to display these data can be creating a new derived TChart class:

```
type
  TSuperChart = class ( TChart )
    { or TCustomChart, TCustomDBChart, TDBChart, etc }
  end;
```

You start then adding new properties, methods and events to TSuperChart to extend or automate your charting needs.

This is the easier and more natural way to extend your objects, and this is very well documented at Delphi's Component Writer Guide.

Linking or Embedding ?

Let's say you have a generic class for which you can derive new controls compatible with your VCL. We can name this class "TMyAncestor".

You can create a new class of your own hierarchy with the purpose of "talking" to a Chart component.

There are two ways (or more !) to do this:

Linking

One way could be creating a new property in your component to "link" any TChart to it.

Your component can then use this property to control the Chart, to create or remove Series and to add or remove points from the Series, just with the same code as if the Chart was standalone in a form.

```
Type TMyControl = class ( TMyAncestor )
  private
    FChart : TCustomChart;
  published
    property Chart : TCustomChart read FChart write FChart ;
end;
```

This is maybe the simplest way to “merge” your components with TChart components. The disadvantage is users will need to drop both a TChart and one of your TMyControl components on the Form, and then use the Object Inspector to “link” the two components.

Embedding

A more advanced need can be to “embed” a Chart inside your own component creating it in your Constructor so it’s not necessary for the user to drop a TChart on the Form. That is, just having your own component in the Form should automatically create and display a TChart component.

This can be represented with this code:

```
type
  TEmbeddedChart = class ( TChart )
  end;

  TMyChart = class( TMyAncestor )
  private
    FChart : TEmbeddedChart;
  published
    property Chart : TEmbeddedChart read FChart;
  end;
```

The Chart property has no “set” method as the Chart shouldn’t be deleted by the user.

Your Constructor should take care of Form inheritance (on Delphi 2 and 3) to avoid creating more than one “embedded” Chart if your control is in an inherited Form. This can be done checking for “csLoading” in your Owner.ComponentState before creating the “embedded” Chart variable.

```
Constructor TMyChart.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  if (csDesigning in ComponentState) and
    (not (csLoading in Owner.ComponentState)) then
  begin
    FChart:=TEmbeddedChart.Create(AOwner);
    With FChart do
    begin
      Parent:=TWinControl(Self);
      Name:=TeeGetUniqueName(AOwner,'MyChart');
    end;
  end;
end;
```

The embedded Chart is assigned a Name calling the “TeeGetUniqueName” method, located in Chart.pas unit.

This method returns a valid and unique Name also when in inherited Forms.

There’s no need to “Free” the embedded Chart if your control is it’s Parent. The default TWinControl Destructor will take care of destroying all child Controls.

You might need the Chart to notify you when it should be displayed, as for example when some Chart property has been changed and the Chart needs to repaint.

This can be done overriding the Invalidate method of TEmbeddedChart, calling the Invalidate method of the Parent (your control) :

```
procedure TEmbeddedChart.Invalidate;
begin
    if Parent<>nil then Parent.Invalidate;
end;
```

Depending the nature of your VCL you might decide to draw the Chart contents using a temporary bitmap or metafile object.

This approach is used in QRTEE.PAS unit to integrate TeeChart with QuickReport reporting tool. The Paint method is overridden and the Chart is converted to bitmap or metafile:

Metafile

```
Meta:=FChart.TeeCreateMetafile(True,Rect(0,0,Width,Height);
try
    Canvas.StretchDraw(GetClientRect,Meta);
finally
    Meta.Free;
end;
```

Bitmap

```
Bitmap:=TBitmap.Create;
with Bitmap do
try
    Width := FChart.Width;
    Height:= FChart.Height;
    FChart.Draw(Canvas,Rect( 0,0, Width, Height ));
    { now you can use here the bitmap as you like }
finally
    Free;
end;
```

It isn't easy to "preview" all possible integration ways as it depends very much on each particular case.

Please email us any integration work you want us to post at TeeChart web site for use of all other TeeChart'ers and 3rd party developers.

8. Appendixes

Appendix A- TeeChart Class Hierarchy

Chart classes:

```

TCustomPanel
  TCustomTeePanel
    TCustomAxisPanel
      TCustomChart
        TChart
          TCustomDBChart
            TDBChart
              TQRDBChart
                TDecisionGraph

```

QuickReport classes:

```

TQRPrintable
  TQRChart

```

Stand-alone classes:

```

TZoomPanningRecord

TBrush
  TChartBrush

TPen
  TChartPen
    TChartArrowPen
    TChartHiddenPen
    TChartAxisPen
    TChartTickPen
    TChartGridPen

TPersistent
  TSeriesPointer
  TChartValueList
  TChartFontObject
    TChartTitle
    TChartAxisTitle
    TChartAxis
    TCustomChartLegend
    TChartLegend
  TSeriesMarks
  TChartGradient
  TChartWall

```

- TList
 - TChartSeriesList
- Exception
 - BarException
 - ChartException
 - LegendException
 - DBChartException
- TDataSource
 - TDBChartDataSource
- TGraphicControl
 - TTeePreviewPage
- TForm
 - TChartPreview
 - TTeeExportForm
- TPanel
 - TTeeGalleryPanel

Series Hierarchy:

- TComponent
 - TChartSeries
 - TCustomSeries
 - TLineSeries
 - TVolumeSeries
 - TOHLCSeries
 - TCandleSeries
 - TAreaSeries
 - TPointSeries
 - TArrowSeries
 - TGanttSeries
 - TBubbleSeries
 - TCustomImagePointSeries
 - TImagePointSeries
 - TDeltaPointSeries
 - TCustomBarSeries
 - TBarSeries
 - TErrorBarSeries
 - TImageBarSeries
 - TBar3DSeries
 - THorizBarSeries
 - TCircledSeries
 - TPieSeries
 - TPolarSeries
 - TFastLineSeries
 - TSurfaceSeries
 - TChartShape

Functions Hierarchy:

- TComponent
 - TTeeFunction
 - TAddTeeFunction
 - TSubtractTeeFunction

```

TMultiplyTeeFunction
TDivideTeeFunction
THighTeeFunction
TLowTeeFunction
TAverageTeeFunction
TCountTeeFunction
TMovingTeeFunction
    TMovingAverageFunction
    TRSIFunction
    TExpAverageFunction
    TMomemtumFunction
TCustomFittingFunction
    TCurveFittingFunction
    TTrendFunction

```

Appendix B- Unit and Files Organisation

The source code files, and their meanings, are documented below:

File name	Description
Core files	
TeeProcs.pas	TCustomTeePanel definition and "tool" routines
TeEngine.pas	TCustomAxisPanel definition. Axis "engine".
TeeFunci.pas	Definition for all standard Functions (Add, Subtract, etc)
Chart.pas	TCustomChart definition. Legend, metafiles, gradient.
DBChart.pas	TDBChart definition
Series files	
Series.pas	Definition for all standard Series (Line, Bar, Area, Pie, etc)
ArrowCha.pas	TArrowSeries definition
BubbleCh.pas	TBubbleSeries definition
GanttCh.pas	TGanttSeries definition
TeeShape.pas	TChartShape definition
OHLChart.pas	TOHLCSeries definition
CandleCh.pas	TCandleSeries definition
Statchar.pas	Extended Functions definitions
Curvfit.pas	TCurveFittingFunction and TTrendFunction definition
ErrorBar.pas	TErrorBarSeries definition
TeeSurfa.pas	TSurfaceSeries definition
TeePolar.pas	TPolarSeries definition
POLY.pas	Polynomial fitting algorithm. Used by CurvFitt.pas
Chart Dialogs (with corresponding DFM file)	
TeeGally.pas	Chart Gallery
Teeprevi.pas	Dialog to Print Preview a Chart
Teexport.pas	Dialog to export & clipboard a Chart to Bitmap or Metafile
TeeAbout.pas	TeeChart About box
Chart Editor Dialog files	
EditChar.pas	Interface functions to access Standard Editor Dialogs
EditPro.pas	Interface methods to access Extended Editor Dialogs

DBEditCh.pas	Interface functions to access DBChart Editor Dialog
IEditCha.pas	32-bit Editor Dialog (Delphi 2, 3 and C++ 1)
IEdit16.pas	16-bit Editor Dialog (Delphi 1)
teeLibs.pas	Double ListBox used at editor dialog

Series Dialogs (with corresponding DFM file)

Axmaxmin.pas	Dialog to change Axis Scales
Axisincr.pas	Dialog to change Axis Increment property.
Pendlg.pas	Dialog to edit a Pen
Brushdlg.pas	Dialog to edit a Brush
Areaedit.pas	Dialog for TAreaSeries
Arrowedi.pas	Dialog for TArrowSeries
Baredit.pas	Dialog for TBarSeries and THorizBarSeries
Bubbledi.pas	Dialog for TBubbleSeries
Flinedi.pas	Dialog for TFastLineSeries
Ganttedi.pas	Dialog for TGanttSeries
Pieedit.pas	Dialog for TPieSeries
Shapeedi.pas	Dialog for TChartShape
Custedit.pas	Dialog for TLineSeries and TPointSeries
Candlei.pas	Dialog for TCandleSeries
Errbared.pas	Dialog for TErrorBarSeries
Surfedit.pas	Dialog for TSurfaceSeries
Polaredi.pas	Dialog for TPolarSeries

Samples

MYPOINT.pas	Sample unit with TMyPointSeries definition
BAR3D.pas	Sample unit with TBar3DSeries definition
BIGCANDL.pas	Sample unit with TBigCandleSeries definition
TEECOUNT.pas	Sample unit with TCountFunction definition
ImaPoint.pas	Sample unit with TImagePointSeries definition
ImageBar.pas	Sample unit with TImageBarSeries definition

Message files (for Internationalization)

teeconst.pas	String constants for Standard Series and Functions
teeProCo.pas	String messages for Extended Series and Functions

Registration files

teechart.pas	Registration unit for Standard, Extended and Samples
teechart.dcr	Registration icons for Delphi component palette
Chartreg.pas	Registration unit for Standard Series and Functions
Chartpro.pas	Registration unit for Extended Series and Functions
teeloper.pas	Registration unit for Sample Series and Functions

QuickReport files

Qrteereg.pas	Registration unit for QuickReport (TQRChart)
QrTee.pas	TQRChart definition

Resource files (sources at *.rc files)

teebmps.res	Bitmaps used by Chart Editor Dialog for standard Series
teeimaba.res	Bitmap used by TImageBarSeries
teeimapo.res	Bitmaps used by TImagePointSeries

teeprbm.res
teeresou.res

Bitmaps used by Chart Editor Dialog for extended Series
Contains the "hand" cursor (crTeeHand)