Controls Demystified

by Glenn Lawrence

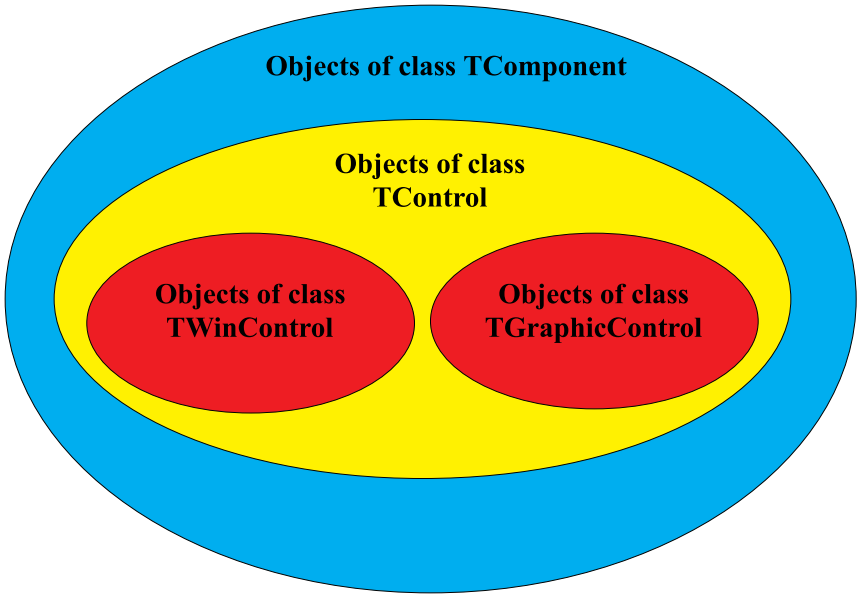
Delphi controls are special types of components whose run time appearance is generally similar to their design time representation. Buttons, labels, grids and check boxes are all examples of controls. When you drop a button on your form, it looks pretty much as it will appear at run time. Contrast this with a main menu component which has an "iconic" representation at design time that looks nothing like how it will appear at run time. Many components, such as timers, don't even have a run time appearance.

Because of their visual nature, controls are sometimes referred to as "visual" (as opposed to "non-visual") components. This terminology can however be slightly misleading as there are some non-visual components, such as the main menu, that have an appearance at run time but are not true controls. To add to the confusion all components, visual or otherwise, reside in the "Visual Component Library."

In user terms the key difference between a control and other components is that all controls share a number of common properties that govern their appearance, notably `Top`, `Left`, `Height` and `Width` (the properties that define the space a control occupies).

In programming terms a control is an object of type `TControl`. By definition this includes any object of a class that is derived from the `TControl` class, such as `TButton`, `TLabel` etcetera. The `TControl` class itself is derived from, or in Borland terminology 'descended from' the `TComponent` class. `TComponent` is the class that represents all component types, including "non-visual" components.

There are two basic types of control, those that have a window of their own and those that use the window of their 'parent.' Those with their own window are called 'windowed' controls and are of type `TWinControl`. Buttons and check boxes fall into this class. The others are called 'graphic' controls and are of type `TGraphicControl`. Label and image controls fall into this class.



➤ *Figure 1*

This class hierarchy is represented by the Venn diagram in Figure 1. The diagram shows that the sets of `TWinControl` and `TGraphicControl` objects are mutually exclusive subsets of the set of `TControl` objects, which is itself a subset of the set of `TComponent` objects.

Notice that this scheme allows for the existence of subclasses of `TControl` other than `TWinControl` and `TGraphicControl`. The VCL supplied by Borland currently has no such classes, but there is nothing to stop third party components being derived direct from `TControl`. There is probably no reason for anyone to do so, but be aware that it could happen.

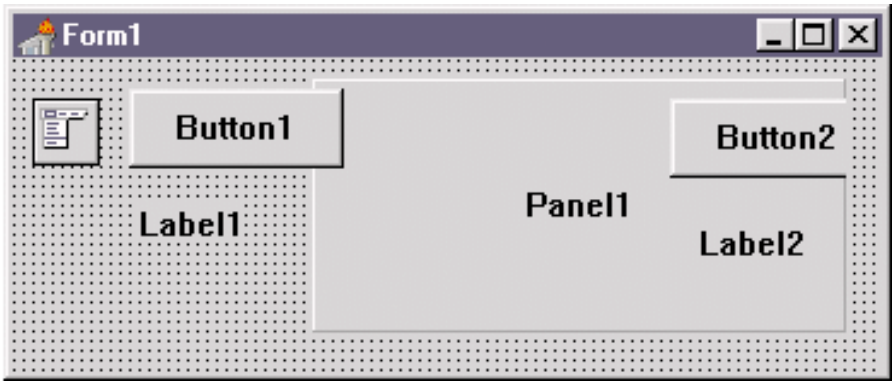
The Control Tree

We have already seen that there are two basic types of control, windowed controls and graphic controls, represented by the classes `TWinControl` and `TGraphicControl`. A control of type `TWinControl` can also be a 'parent' to a number of 'child' controls. Child controls can be any control type, not only graphic controls, but also other windowed controls.

This recursive relationship leads to a tree structure with the root and intermediate nodes being of type `TWinControl`, and leaf nodes being either `TWinControl` or `TGraphicControl`.

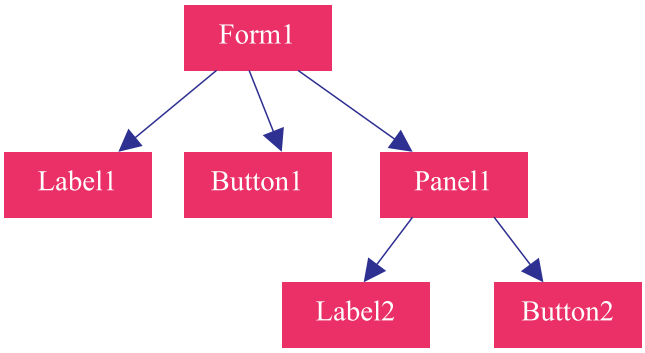
Figure 2 shows an example of a form (ie a `TWinControl`) that contains a panel (`TWinControl`) a number of buttons (`TWinControl`) and some labels (`TGraphicControl`) plus a menu component. The control tree would look something like Figure 3.

In this example `Label1` and `Button1` are child controls of `Form1`, whereas `Label2` and `Button2` are child controls of `Panel1`. Note that the menu component doesn't appear in the control tree as it is a *non-visual* component, and not therefore a control.



➤ *Figure 2*

➤ *Figure 3*



So what does it mean for a control to be a child of a parent?

The parent-child relationship governs the visual appearance of controls such that child controls always appear within their parent. In Windows parlance, they are 'clipped' by their parent as `Button2` is clipped by `Panel1`. One obvious corollary to this is that if a parent control is made invisible, all of its child controls will also be made invisible.

The parent-child relationship also governs the handling of user events such as mouse messages and key presses. See *Disabled Control Surprises* on page 53 for more information.

Seasoned Windows programmers will no doubt have gathered that the parent-child relationship of the windowed controls reflects a similar relationship that occurs between the underlying windows that they represent. Indeed, many Delphi controls are simply 'wrappers' around standard windows controls.

For example, novice Delphi programmers often wonder why it's not possible to change the colour of a `TButton`. The reason is that `TButton` is simply a wrapper around the underlying Windows button control, and in the interests of consistent look and feel the authors of Windows decided that the programmer would not be allowed to change the colour of individual buttons. Although the user can of course change the colours of *all* buttons by changing the local colour scheme in the Control Panel.

Fortunately you don't really need to understand how the underlying Windows system works to use controls as Delphi encapsulates the most important aspects into methods and properties. However, if you want to get really deep understanding of what is going on 'under the covers' the classic reference for this material is *Programming Windows* by Charles Petzold, or the later *Programming Windows 95* by Petzold and Yao.

The parent-child relationship also plays a part in the handling of various properties like fonts, colours and hints at design time. Check out `ParentColor`, `ParentFont`, `ParentCtrl3d` and `ParentShowHint` in Delphi's online help to see how certain properties of child controls can be linked with the same properties of their parent. This is why you can for example change the font of a form and the change conveniently propagates to its children.

Controls that share the same parent are 'sibling' controls. `Button1` and `Panel1` above are examples of siblings. Note that siblings are allowed to visually overlap, subject to the natural restriction that graphic controls cannot overlap windowed controls. When designing a form it is not always easy to tell from just looking whether an overlapping control is a sibling or a child. Fortunately Delphi gives you a very neat way of finding out. Simply select the control and press the `Escape` key and this will cause its parent control to be selected and highlighted. This is also very handy for getting to parent controls that completely filled by children with their `Align` property set to `alClient`.

Unlike the *class hierarchy*, which I touched on above, the *control tree* is a run time relationship between object *instances*. It is even possible for controls to change parents at run time, simply by reassigning their `Parent` property. My `TAimSizerPanel` component does this for example to support its sub-form feature. A child can never have more than one parent control at a given time however.

Most controls, whether windowed or graphic, need a parent. If you have ever tried to create a control on the fly you may have noticed that it will refuse to appear until its `Parent` property is set. The notable exception to this rule is the `TForm` control which can be a 'top level window' control and therefore does not need a parent.

The child-to-parent relationship is expressed through the child's Parent property. In the case of a root control (usually a form) the `Parent` property will be `Nil`.

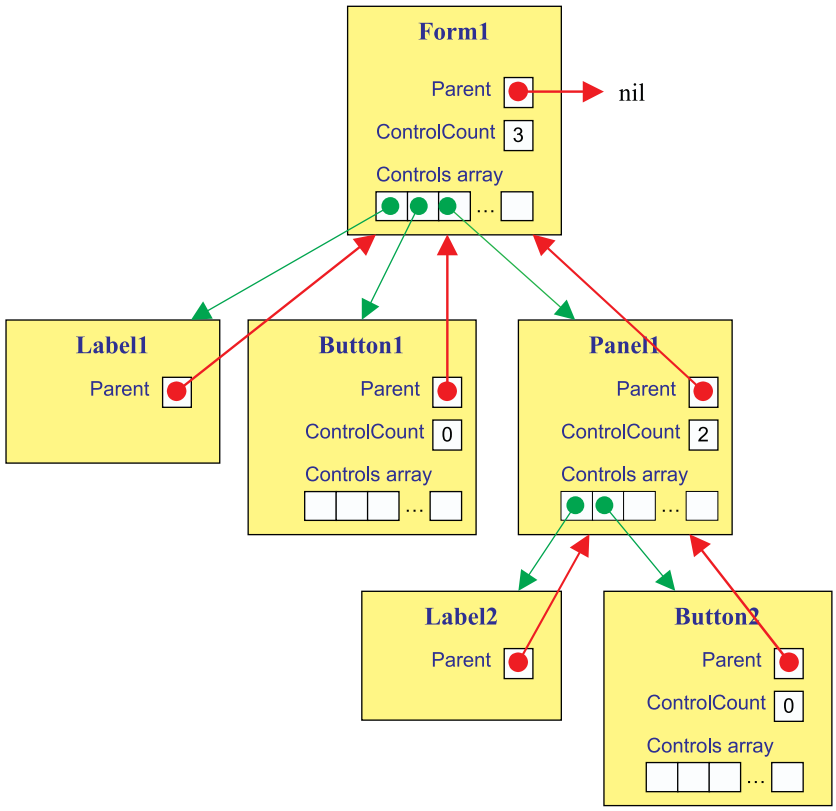
The parent-to-child relationship is expressed through the parent's `ControlCount` and `Controls` (array) property through which you can find the current set of children for

a given parent. These properties are however *read only* properties so you can't use them to modify this set. If you want to change the parentage of a control you must set the `Parent` property of the child. The parents' `ControlCount` and `Controls` properties will be updated automatically.

The parent's `Controls` array contains all controls that are currently its children, both windowed and graphic. The `Controls` array also represents the Z-order of the child controls, with lower indexes being furthest to the back, and higher indexes closest to the front. Because they are rendered directly on the parent window, graphic controls can never appear in front of any windowed siblings, and therefore will always come first in the `Controls` array. Figure 4 shows in detail how the control tree of the above example is implemented.

Note that labels, being derived from `TGraphicControl` are the only controls in this example that can't be parents. All other controls are derived from `TWinControl` so they each have a `Controls` array containing the references to any children they may have. It doesn't make a lot of sense for buttons to have child controls but they *are* capable of it and the effect is quite weird. Usually though, as in this example, their `Controls` array will be empty.

You can traverse the complete control tree by visiting each element of the `Controls` array of the root control (usually, but not necessarily, a form) and then recursively visit the elements of the `Controls` array for any windowed controls encountered.



➤ *Figure 4*

Let's apply this knowledge to a practical example.

"Hit" Testing That Really Works

Although Delphi's automatic fly-over hints and the event driven drag and drop mechanism reduces the need for it, there are still many occasions where it is useful to be able to tell when the mouse is over a certain control. For example I have a component `TAimHelpButton` that needs to do this.

The idea is that the user clicks a button or presses `Alt-F1` to enter help mode. The screen cursor changes to the arrow or question mark and the user then clicks on a control and a pop-up help box appears. Neil Rubenking has an example of something like this in his excellent book *Delphi Programming Problem Solver* in which he uses the Delphi function `FindDragTarget` to detect the underlying control.

Unfortunately there is a problem with this approach in that `FindDragTarget` does not detect disabled windowed controls. Even though the Delphi documentation implies that it does. Disabled graphic controls are fine, but disabled windowed controls and their children are ignored. For a more detailed explanation see *Disabled Control Surprises* on page 53.

The solution to this problem is given by the functions shown in Listing 1. This function recursively descends the control tree starting from the given 'parent' control and

➤ *Listing 1*

```
function FindTopMostWinControlAtPos(
  parent: TWinControl;     { The parent at the top of the control tree }
  pt: TPoint               { The "hit" test point in screen coordinates }
  ): TWinControl;
var
  i: integer;
  c: TControl;
begin
  Result := nil;
  if parent.Visible then begin
    i := parent.ControlCount -1;
    while (i >= 0) and (Result = nil) do begin   { Check children first }
      c := parent.Controls[i];
      if c is TWinControl then  { Recursively descend }
        Result := FindTopMostWinControlAtPos(c as TWinControl, pt);
      i := i -1;
    end;
    if Result = nil then begin  { Check parent control last }
      pt := parent.ScreenToClient(pt);  { Convert point to local coords }
      if (pt.X >= 0) and (pt.X < parent.Width)
        and (pt.Y >= 0) and (pt.Y < parent.Height) then
        Result := parent; { Found it! }
    end;
  end;
end;
```

```
function GetTopMostControlAtScreenCoords(
  pt: TPoint;              { Screen coords for "hit" test }
  allow_disabled : boolean { Indicates if disabled controls are included }
  ): TControl;
var wc : TWinControl;
begin
  Result := FindDragTarget(pt, allow_disabled);
  if (Result <> nil) and (Result is TWinControl) and allow_disabled then begin
    { Check for disabled child windowed controls }
    wc := FindTopMostWinControlAtPos((Result as TWinControl), pt);
    if wc = nil then abort; { Can't happen - honest guv! }
    pt := wc.ScreenToClient(pt);   { Convert to local coords }
    Result := wc.ControlAtPos(pt, True);
    if Result = nil then Result := wc;
  end;
end;
```

➤ *Listing 2*

Disabled Control Surprises

You would expect that when a control is disabled it would not respond to mouse events, but less obvious is that its child controls will also not respond. Even though its Enabled property is set True a control will not respond to mouse events if its parent, grandparent or any control higher in the control tree is disabled.

This can sometimes be useful, to disable a whole panel full of controls for example, but be aware that the child controls so disabled will *not* be greyed out and this could confuse your user.

Even more disconcertingly, if you have a disabled control sitting on top of an enabled sibling control mouse events will pass through the disabled control to be caught by the control beneath it. Your user may think he is clicking one button, but may actually be clicking one he can't even see!

The FindDragTarget function and other parts of the VCL fail to detect disabled windows because they rely on the Windows API function WindowFromPoint which even according to the documentation "does not retrieve the handle of a hidden, *disabled*, or transparent window."

tests each visible windowed control to see if it contains the given screen coordinates point.

It first checks all of the leaf controls in the control tree as these will be the top most controls. It also traverses each `Controls` array from back to front, so that the top most siblings are checked first.

The function then uses `FindTopMostWinControlAtPos` to complete the hit test (Listing 2). This function first uses the standard Delphi function `FindDragTarget` to obtain the top most control. Even though this function takes a boolean value to indicate that disabled controls are wanted, it will not find disabled windowed controls. It will therefore return a graphic control (disabled or enabled) or an enabled windowed control, or nil to indicate no control found.

If `FindDragTarget` returns a windowed control and we are also interested in disabled controls then we need to call our new function `FindTopMostWinControlAtPos` to see if there might be a disabled child windowed control (wc) that is at the given screen coordinates.

We then use the `TWinControl` method `ControlAtPos` to check for any graphic controls that might be at the given screen co-ordinates.

The DRAGRACE project on this month's disk is a novelty application that shows this code put to use in a simple game involving drag and drop onto disabled components.

Conclusion

As Delphi programmers we use controls every day. Delphi makes it so easy that we often don't think about it what is happening. However there are times when a deeper understanding of Delphi controls is very useful. Hopefully this article has helped you to acquire some of that understanding.

Acknowledgment

I would like to thank the members of the Australian Delphi User Group whose questions prompted me to write this article and whose critical attention has helped to improve it.

Glenn Lawrence is a busy guy: as well as working with AIMtec P/L in Bundoora, Victoria, he helps run the Australian Borland User Group. He can be contacted on CompuServe at 101463,3252.